TECHNICAL REPORT 00-02

Exploiting Segmentation Mechanism for Protecting against Malicious Mobile Code

Takahiro Shinagawa Kenji Kono Takashi Masuda

May 17, 2000

DEPARTMENT OF INFORMATION SCIENCE
FACULTY OF SCIENCE, UNIVERSITY OF TOKYO

7–3–1 HONGO, BUNKYO-KU TOKYO, 113 JAPAN

TITLE
Exploiting Segmentation Mechanism for Protecting against Malicious Mobile Code

AUTHORS
Takahiro Shinagawa, Kenji Kono, and Takashi Masuda

KEY WORDS AND PHRASES
operating system, fine-grained protection domain, virtual memory, mobile code, security

ABSTRACT
This paper describes a mechanism for protecting against malicious mobile code. As mobile code is linked with a hosting application and executed in the same process, a fine-grained protection domain providing an intra-process protection is required to prevent a malicious mobile code from unauthorized access. This paper introduces a multi-protection page table: a mechanism of virtual memory that enables fine-grained protection domains to be supported at the kernel level. A fine-grained protection domain (1) confines the memory accesses by mobile code in authorized areas, (2) restricts the system calls issued by mobile code, and (3) enables efficient cross-domain calls among mobile codes and a hosting application. Efficiency of cross-domain calls encourages the use of fine-grained protection domains. This paper demonstrates that a multi-protection page table can be implemented efficiently on the most widely used architecture; that is, Intel x86 family. The presented implementation achieves reasonable performance for practical use; one round-trip cross-domain call requires 226 to 608 cycles. Experimental results show that the protection overhead is only 6.1% to 15.8% in a real application.

REPORT DATE	**WRITTEN LANGUAGE**
May 17, 2000	English
TOTAL NO. OF PAGES	**NO. OF REFERENCES**
16	15

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT
Submitted for publication

DISTRIBUTION STATEMENT
This technical report is available ONLY through `http://www.is.s.u-tokyo.ac.jp/tech-reports/FILES.html`.

SUPPLEMENTARY NOTES

DEPARTMENT OF INFORMATION SCIENCE
Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

Exploiting Segmentation Mechanism for Protecting against Malicious Mobile Code

Takahiro Shinagawa[†] Kenji Kono[††,†††] Takashi Masuda[††]

†Department of Information Science,
Graduate School of Science,
University of Tokyo
7-3-1 Hongo Bunkyo-ku, Tokyo 113-0033 Japan
Email:shina@is.s.u-tokyo.ac.jp

††Department of Computer Science,
University of Electro-Communications
1-5-1 Chofugaoka Chofu-shi, Tokyo 182-8585 Japan
Email:{kono, masuda}@cs.uec.ac.jp
†††Japan Science and Technology Corporation

Abstract

This paper describes a mechanism for protecting against malicious mobile code. As mobile code is linked with a hosting application and executed in the same process, a fine-grained protection domain providing an intra-process protection is required to prevent a malicious mobile code from unauthorized access. This paper introduces a multi-protection page table: a mechanism of virtual memory that enables fine-grained protection domains to be supported at the kernel level. A fine-grained protection domain (1) confines the memory accesses by mobile code in authorized areas, (2) restricts the system calls issued by mobile code, and (3) enables efficient cross-domain calls among mobile codes and a hosting application. Efficiency of cross-domain calls encourages the use of fine-grained protection domains. This paper demonstrates that a multi-protection page table can be implemented efficiently on the most widely used architecture; that is, Intel x86 family. The presented implementation achieves reasonable performance for practical use; one round-trip cross-domain call requires 226 to 608 cycles. Experimental results show that the protection overhead is only 6.1% to 15.8% in a real application.

1 Introduction

Code mobility is indispensable to Internet computing. Code mobility provides extensive flexibility for distributed computing because any fragment of program code can be added dynamically both to clients and to servers in order to enhance their functionalities on demand. More specifically, we define *mobile code* as a code fragment that is downloaded from Internet, is linked with a hosting application, and runs as a part of the application. Once a mobile code is linked with a hosting application, it runs in the same process as the hosting application runs in. For example, a web browser such as Netscape Navigator allows the user to download a mobile code called a plug-in to enhance the ability of browsing multimedia contents. Code mobility is now gaining widespread acceptance; servlets, applets, and ActiveX are all examples of mobile code.

Mobile code raises critical security concerns. It provides not only a flexible and powerful platform for Internet computing, but also a convenient carrier of computer virus. Anyone can develop a mobile code and make it available to everyone through Internet. The developer of mobile code is often anonymous; a code may be developed by an adversary. Even if the developer is known and trustful, mobile code may be buggy and damage the local computing systems.

Protecting against malicious mobile code is challenging because it exposes critical design flaws in operating systems. Traditionally, operating systems provide only inter-process protection; that is, a process coincides with a protection domain. This coincidence incurs two critical design flaws, resulting in inadequate security support for mobile code systems. First, no intra-process memory protection is supported. Since a mobile code can access any location of a hosting application's memory, it can steal sensitive information such as passwords stored in the application's memory. Second, all execution entities in the same process are given the same authority to issue system calls. Since even a mobile code is given the same authority as a hosting application, it can access user's files, perform network communications, etc.

This paper introduces a new abstraction called *fine-grained* protection domain to reduce security risks caused by mobile code. Fine-grained protection domains can co-exist within a traditional protection domain (i.e, a pro-

cess), and confines all accesses by mobile code in a fine-grained protection domain. By using fine-grained protection domains, the memory access by mobile code is restricted to a certain memory area, and each mobile code is given different authorities to issue system calls. The notion of fine-grained protection domain solves the design flaws in traditional operating systems, and is expected to facilitate the development of various security mechanisms for mobile code.

The notion of fine-grained protection domain is simple but the implementation is not straightforward if the overhead of protection is considered. Since a mobile code is a part of a hosting application, it shares computing resources extensively with the hosting application and frequently invokes APIs provided by the application. For example, Java applet (an example of mobile code) invokes such APIs 30,000 times per second [15]. The goal of the implementation is to provide intra-process protection, while achieving efficient cross-domain calls between fine-grained protection domains.

To implement fine-grained protection domains efficiently, the authors' previous paper [12] described a novel mechanism called *multi-protection page table*. A multi-protection page table extends a conventional page table so that the intra-process protection can be implemented without degrading the performance of cross-domain calls between fine-grained protection domains. For efficiency reasons, we exploited processor-specific features to implement multi-protection page tables. The implementation described in the previous paper exploited the features specific to SPARC processors such as register windows, tagged TLBs, and physical address caching. It was an open problem whether a multi-protection page table can be implemented efficiently on Intel x86 processors because Intel x86 lacks such features. Since over 90% of desktop computers are equipped with Intel x86 processors, it has a great impact on the practicality of multi-protection page tables.

This paper demonstrates that a multi-protection page table can be implemented efficiently on Intel x86 family. This implies that almost all computers, i.e., over 90% of desktop computers, can employ a multi-protection page table as the protection basis. Thus, the implementation on Intel x86 family provides a good testbed to investigate the usefulness of multi-protection page tables. In addition, if it has been proven to be useful in practical, the implementation can be directly ported to almost all computers. To implement multi-protection page tables, we have newly devised a method that utilizes the segmentation and ring protection mechanism of Intel x86. Since SPARC processor lacks segmentation and ring protection and Intel x86 lacks register windows and tagged TLBs exploited in the SPARC implementation, the Intel x86 implementation is quite different from the SPARC implementation and it was a real effort to reach the Intel x86 implementation.

Experimental results show that the Intel x86 implementation is efficient enough to be used in real applications. One cross-domain call between fine-grained protection domains requires only 226 cycles or 608 cycles, depending on the configuration. In most cases, a cross-domain call requires the less cycles, i.e., 226 cycles. To investigate the protection overhead in a real application, we configured an Apache web server to use fine-grained protection domains. An Apache web server allows external modules to be linked with, and by default it contains 30 external modules. These modules can be thought of as mobile code because they can be downloaded from Internet, be linked with the server, and run as a apart of the server. In the experimental configuration, all 30 modules are put in different fine-grained protection domains. In the experiments, the throughput of the web server degrades only 6.1% to 15.8%. The latency of one HTTP request is increased by only 0.11ms to 0.25ms, which is negligible compared with network latency.

The rest of the paper is organized as follows. Section 2 describes the protection model of fine-grained protection domains. Section 3 describes the implementation of fine-grained protection domains on Intel x86 family. Section 4 reports the experimental results. Section 5 describes related work. Section 6 concludes the paper.

2 Protection Model

2.1 Fine-grained Protection Domain

In conventional operating systems, a process coincide with a protection domain. If a mobile code is placed in the same process as a hosting application, no protection is provided between the application and the mobile code. If a mobile code is placed in a process other than a hosting application, all interactions between the mobile code and the hosting application require interprocess communications. Since a mobile code frequently invokes APIs provided by a hosting application, the overhead incurred by interprocess communication is unacceptably high. For example, Java applet invokes 30,000 times per second the APIs provided by a hosting application. In a conventional operating system like Linux, if interprocess communications are performed at such high frequency, 30% of computation time is spent in interprocess communications [11].

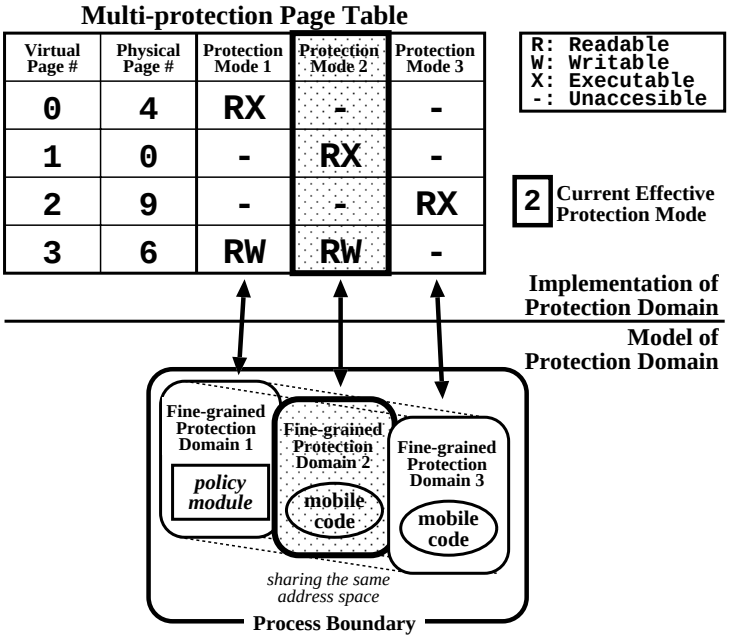

Figure 1: Protection Model. A process consists of multiple fine-grained protection domains, each of which has its own memory protection. A policy module determines a protection policy of the process in which it resides.

To address this issue, this paper introduces the notion of fine-grained protection domains. For brevity, we may simply refer to a fine-grained protection domain as a protection domain. In the protection model based on fine-grained protection domains, a process holds more than one protection domain within itself (Figure 1). For each process, there is exactly one *policy module* that determines a protection policy of the process, and the kernel obeys the policy supplied by the module. Each fine-grained protection domain has different authorities to access memory and issue system calls. As shown in the figure, the policy module is placed in one of the fine-grained protection domains and is protected from mobile codes.

Note that all fine-grained protection domains in the same process share a single address space. The sharing of an address space facilitates pointer-passing amongst fine-grained protection domains because no pointer conversions are required.

2.2 Multi-Protection Page Table

When loading a mobile code, the policy module creates a new fine-grained protection domain and sets up a protection policy for the mobile code. The policy module determines which memory page is readable, writable, and executable by the mobile code.

A *multi-protection page table* extends a traditional page table so that different protections can be assigned to each fine-grained protection domain (see Figure 1). Each row of a multi-protection page table consists of more than one protection mode in addition to the mapping of the virtual and a physical memory page. Each column of protection mode corresponds to one fine-grained protection domain. Namely, each fine-grained protection domain has its own column of protection mode. For example, the data segment of a policy module is set to be unaccessible by mobile codes, whereas the data segments of mobile codes are set to be readable and writable by the policy module.

For efficient data sharing, fine-grained protection domains can share memory pages with one another. In Figure 1, the policy module and one mobile code share the virtual page #3. Since all fine-grained protection domains in the same process share a single process, complex data structures including pointers (virtual addresses) can be shared without converting pointers.

2.3 Cross-Domain Calls

The protection model based on fine-grained protection domains can provide efficient cross-domain calls. The notable feature of fine-grained protection domains is that all computing resources are shared between protection domains, while the authorities to access those resources are different from protection domain to protection domain. Therefore a cross-domain call does not need to switch computing resources; it has only to switch the authorities to access resources. This feature of cross-domain calls contrasts with conventional interprocess communications. An interprocess communication involves the overhead of switching computing resources such as scheduling, switching page tables, flushing TLB, and statistical housekeeping. A cross-domain call does not require those time-consuming operations; thus a cross-domain call is expected to be much more efficient than conventional interprocess communications.

2.4 Restricting System Calls

After setting up the memory protection policy, the policy module determines which system calls the mobile code can issue. The policy module plays the role of reference monitor. When a mobile code issues a system call, it is hooked by the policy module. Then the policy module checks if the mobile code has a permission to issue that system call. If it has, the policy module directs the kernel to execute the system call. Otherwise, the mobile code is determined to have attempted an illegal access, and the hosting application is notified. Since the policy module is an ordinary user program, the designer of security policy can implement any policy as needed.

2.5 An Example

As a concrete example, consider the case where a web browser attempts to load a plug-in from Internet. In this case, the web-browser plays the role of a policy module and determines protection policies. When loading a plug-in, the browser creates a new fine-grained protection domain and sets up the memory protection of the plug-in. In most cases the memory protection is set so that the plug-in can execute its own code, and read and write its own data and stack segments, while the memory area of the browser is set to be unaccessible by the plug-in. Then the plug-in starts executing and performs cross-fine-grained protection domain calls to invoke the browsers APIs. When the plug-in issues a system call, it is hooked and checked by the web-browser.

3 Implementation

3.1 Segmentation-Based Implementation

Our previous paper[12] has already shown the implementation technique of multi-protection page tables on SPARC processors. This technique fully exploits SPARC-specific features such as tagged TLBs and register windows. Unfortunately, since Intel x86 lacks such features, the technique can not be applied to the Intel x86 family. Intel x86 is used on over 90% of desktop computers. It is significant for practical reasons to develop an implementation technique on Intel x86 family.

3.1.1 Segmentation Mechanism

The segmentation mechanism of Intel x86 provides a programmer with the *segmented* memory model. With this memory model, memory consists of independent address spaces called *segments*. To address a byte in memory, a program issues a *logical address* consisting of a 16-bit *segment selector* and a 32-bit *offset*. A segment selector identifies the segment and an offset specifies a byte in the address space of the segment. A segment selector must be stored in a *segment register* of the processor and a program issues a 32-bit offset alone as a *virtual address*. The relationship between logical address, segment, and virtual address is shown in Figure 2. Loading a segment selector into a segment register can be done in user mode.

Segments are protected from each other, as each segment corresponds to a single address space. An access beyond the segment limit or without legal permission, is detected by the processor and causes a trap to the kernel. Access permissions of each segment can be changed only in kernel mode.

Internally, all the segments are mapped into the processor's 4GB flat address space called *linear address space*. as shown in Figure 2. The processor translates a logical address into a linear address. This translation is transparent

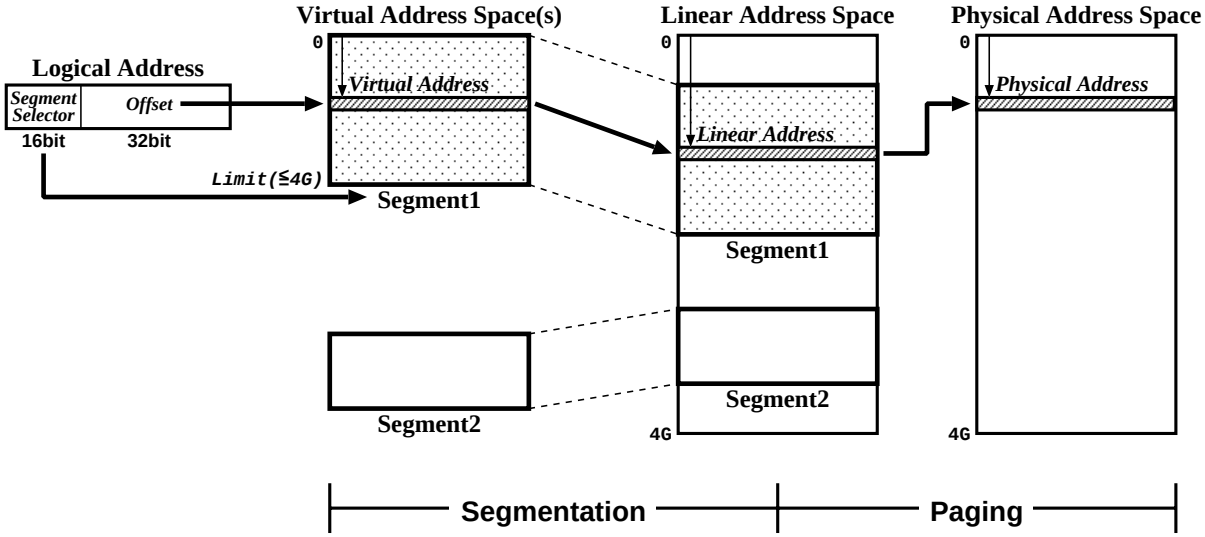

Figure 2: Segmentation mechanism

to application programs. That is, a linear address is different from a virtual address and invisible to application programs. Then the processor translates a linear address into a physical address by *paging* in the same way as ordinary processors.

3.1.2 Segmentation Based Implementation

With our abstraction of the multi-protection page table, a fine-grained protection domain has two primary features. One is that all protection domains share the same address space, and the other is that each protection domain holds its own page-protection mode. We have implemented a multi-protection page table using segmentation mechanism in the following three steps (see Figure 3).

First, a segment is assigned to each fine-grained protection domain for protection purpose. The segment prevents a mobile code from accessing other protection domains, as segments are protected from each other. However, it is not safe enough because a *malicious* mobile code may violate a hosting application's protection domain by loading the segment selector associated with the application. To prevent this, all segments are disabled by the kernel except for the one associated with the currently effective protection domain. Thereby it becomes impossible even for a malicious mobile code to access other domains because a user program can not enable any segments.

Second, we arrange each segment in the processor's linear address space so that the segments do not overlap each other. In Figure 3, three segments are arranged without overlapped in the linear address space. Thereby, each fine-grained protection domain holds its own entries of the page table because there is one entry of the page table for each linear page. Hence the page-protection mode can be determined independent of other protection domains. If segments were mapped into the same region in the linear address space, they would have to share the same page-protection mode.

Third, the mapping from a virtual address to a physical address is set up so that the mapping becomes identical among all the fine-grained protection domains. In this mapping, all protection domains share the same address space, while a protection mode of each page may be different.

A cross-domain call is implemented efficiently using a software trap. We prepared a dedicated software trap to handle cross-domain calls. This software trap switches effective protection domains. When a thread issues the software trap, the kernel disables the caller's segment and enables the callee's segment. Then the kernel loads into the segment registers selectors corresponding to the callee's segments . Finally, the kernel sets up the callee's stack pointer and transfers control to the entry point of the callee. This is all to be done: the kernel need not schedule, flush TLBs, and switch resources because all protection domains share the computing resources of the same process.

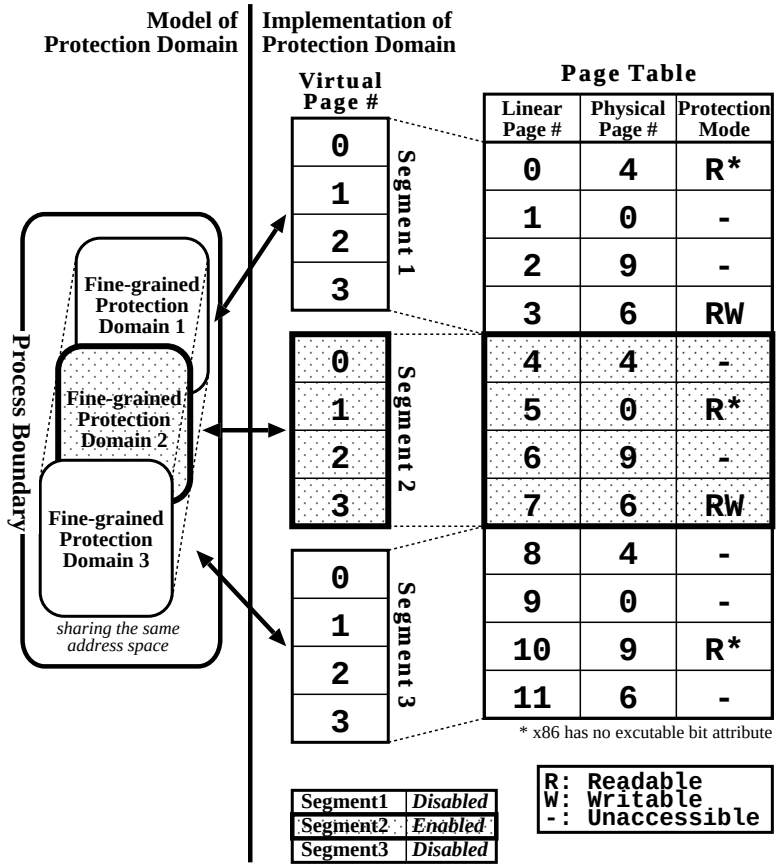

Figure 3: Implementation of a multi-protection page table

The entry point of each protection domain is registered in the kernel. The return address is stored on the stack of the caller to deal with recursive cross-domain calls. The stack pointer of the calling thread is registered in the kernel and saved/restored at the time of cross-domain calls. The arguments of cross-domain calls are placed on registers. If the number of arguments is larger than that of registers, extra arguments are placed on memory page shared by the caller's and the callee's protection domain.

Intel x86 supports 8192 segments for one process and one protection domain requires at least two segments (for code and data). Therefore, the maximum number of mobile codes is 4096, which will be sufficient for practical use. However, the approach shown in this section may require the linear address space as large as the sum of the sizes of all segments because each segment is arranged on different areas in the linear address space. Consequently, an application may exhaust the 4GB(2^{32}) linear address space. To reduce the size of segments, the hosting application's memory pages unaccessible from a mobile code are dropped from the segment corresponding to the mobile code. In addition, shared memory areas are allocated at the address as low as possible. If shared memory areas are allocated at a higher address, a segment must be expanded to include those shared memory areas.

3.2 Optimizing Cross-domain Calls between Application and Mobile Codes

A cross-domain call between a hosting application and a mobile code can be optimized by exploiting the ring protection of Intel x86. This optimization is effective because cross-domain calls between a hosting application and mobile codes will be performed more frequently than between mobile codes. We refer to the mechanism described in Section 3.1.2 as segment switch, and to the mechanism described in this section as ring switch. As shown in Figure 4, ring switch implements the cross-domain calls between a hosting application and mobiles codes, and segment switch implements the calls between mobile codes.

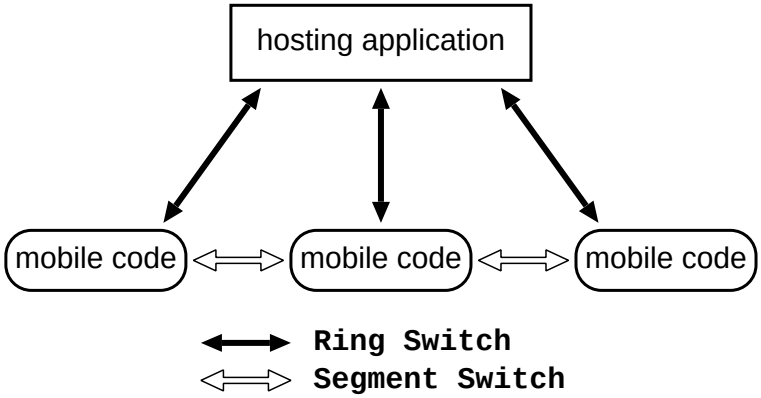

Figure 4: Suitable situation for two mechanisms

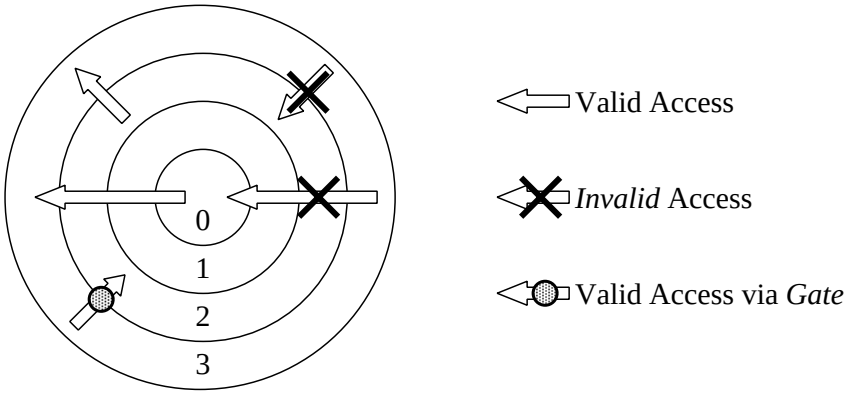

Figure 5: Ring Protection Mechanism

3.2.1 Ring Protection

The segment-protection mechanism of Intel x86 recognizes 4 privilege levels, numbered from 0 to 3. The privilege level 0 is the most privileged and the privilege level 3 is the least privileged. Normally, a kernel resides in the segment with privilege level 0, and a user program resides in the segment with privilege level 3. Figure 5 shows how these levels of privilege can be interpreted as rings of protection.

To call a procedure operating at higher privilege segments, a program must do it by means of a *gate*. A gate is tightly controlled and protected interface to communicate safely with procedures in higher privilege segments. An attempt to access higher privilege segments without going through a gate or without having appropriate access rights raises a protection fault.

3.2.2 Optimization by Ring Protection

We have optimized cross-domain calls between a hosting application and mobile codes using the ring protection. The point of this optimization is that the application have more privileges than mobile codes. As described in Section 2.5, a hosting application usually plays the role of policy module. Thus mobile codes do not necessarily have to be protected from the hosting application.

Taking this point into consideration, the privilege level of a hosting application is raised from 3 to 2, while the mobile codes remain at the privilege level 3. Then, the ring protection mechanism prevents a malicious mobile code from accessing unauthorized data of the application, because the privilege levels of mobile codes are lower than that of the application. Mobile codes can not access to the application's segment even if the application's

segment is not disabled. Therefore, it is unnecessary to disable and enable segments when cross-domain calls are performed between a hosting application and mobile codes. As a result, we can avoid the cost of entering the kernel; a cross-domain call is performed by two instructions: the instructions for inter-segment call/return by way of the gate.

We must ensure that there is no security hole when the privilege level of a hosting application is raised from 3 to 2. First, the privileged instructions are not problem because they can be executed only in the privilege level 0. Second, a program in a privilege level higher than or equal to *IOPL* (I/O privilege level) can make an I/O access. IOPL is stored in the processor's system register and can be modified in the privilege level 0. Again this is not problem because IOPL is normally 0. Finally, a program in the privilege level 2 can access a memory page whose page-level protection is marked as *supervisor pages*. A program in the privilege level 3 can not access to supervisor pages. Supervisor pages are intended to be used for protecting the kernel from user programs but Linux, on which our implementation is based, does not need to utilize supervisor pages to protect the kernel. Instead, the kernel is protected by the segmentation mechanism. Again, this is not the problem. Only the problem is that Intel 386 processor, an early implementation of Intel x86, has a bug that the page-level write-protection in the privilege level 2 is always ignored. There is not such a bug in later processors, such as Intel 486, Pentium, and P6 family. Consequently, there is no problem even if the privilege level of a hosting application is raised to 2.

This approach may slightly increase risk of attacks from a malicious mobile code. For example, a malicious mobile code may be able to cause the application to jump to and execute code within the mobile code by means of buffer overflow attacks. This sort of attacks is not unique to our approach. Several techniques are available to defend against buffer overflow attacks[4].

3.3 Restricting System Calls

Our mechanism provides the policy module with a mechanism to prevent a mobile code from issuing arbitrary system calls. This mechanism retains backward compatibility with existing systems at the binary level since the interfaces of system calls need not changed. The policy module registers a callback routine in the kernel. When a mobile code issues a system call, the kernel upcalls the callback routine. At the same time, the kernel passes the ID of the mobile code issuing the system call, type of the system call, and arguments of the system call. Based on the protection policy, the policy module determines whether the system call is granted permission to execute, or not. As a callback routine is a normal procedure, various protection policies can be implemented using capability, access control list, and so on.

This mechanism exploits the features of fine-grained protection domains. First, the policy module can check a system call argument without extra costs even if it contains a pointer. No pointer conversions are required because the policy module shares the same address space with mobile codes. Second, the callback routine in the policy module is invoked by way of our cross-domain call. Since our cross-domain call is efficient as shown in Section 4, the overhead of calling the callback routine is not high. Moreover, as a user program is normally designed not to issue system calls so frequently, the cost of checking system calls is relatively low.

3.4 Library

For ease of use, we have implemented a library for loading mobile codes. This library creates a segment in the linear address space for the fine-grained protection domain of the code. Then, the library loads the mobile code into the address space of the created segment. Finally, the library registers the entry point and initial stack address of the mobile code in the kernel.

This library manages dynamic linking information for every protection domain. Shared libraries are linked against each mobile code and are *not shared* among mobile codes. It is necessary for efficiency and security reasons. If a shared library is put in another fine-grained protection domain, a shared library's function such as `memcpy()` is called frequently by way of a cross-domain call and the performance degrades significantly. If mobile codes share a shared library, static data areas used by a library function such as `printf()` might be destroyed by a malicious mobile code.

4 Experiments

To validate our approach to fine-grained protection domains, we have implemented a prototype system by altering Linux 2.2.14 kernel. The prototype system is running on a PC with Intel PentiumII 400MHz processor, and

Table 1: Cycles for one cross-domain call

Method	Cycles	Time
Segment switch	608	1.52 μs
Ring switch	226	0.57 μs
IPC	3975	9.94 μs

Table 2: Breakdown of segment switch (one-way)

Operation	# of Instructions	Cycles
Enter kernel	1	86
Parameter check	16	16
Context setup	21	12
Segment enable /disable	12	12
Segment register reload	6	30
misc.	20	11
Leave kernel	1	137
Total	77	304

128Mbytes of main memory. As described in Section 3, we have proposed two approaches to implementing fine-grained protection domains on Intel x86 family: Segment switch exploiting the segmentation mechanism (described in Section 3.1.2) and ring switch exploiting the ring protection mechanism (described in Section 3.2.2).

4.1 Micro Benchmark

To estimate the cost of cross-domain calls, we measured processor cycles spent in a cross-domain call and compared it with those of a conventional interprocess communication. From this comparison, we can verify that cross-domain calls between fine-grained protection domains are more efficient than interprocess communications. In the experiment, a null procedure was called repeatedly in a tight loop to reduce the effect of cache misses. The number of cycles was obtained from the cycle counter, called time-stamp counter, of PentiumII processor. We also measured the average cycles spent in an interprocess communication (Hereinafter, IPC) of Linux. In this experiment, two processes communicate via a semaphore.

Table 1 shows the result of measuring cycles for one cross-domain call. Segment switch costs 608 cycles, and ring switch costs 226 cycles for one cross-domain call. In both cases, a cross-domain call between fine-grained protection domains is faster than IPC; 6.5 times faster in segment switch and 17.6 times faster in ring switch. Comparing segment switch and ring switch, ring switch is 2.7 times faster than segment switch. Thus ring switch successfully optimizes cross-domain calls in special but most frequent cases between a hosting application and mobile codes.

To analyze the cost of cross-domain calls, Table 2, Table 3, and Table 4 list the breakdown of cycles for one cross-domain call in segment switch, ring switch, and IPC respectively. Table 2 shows the one-way cost of a cross-domain call because the return sequence is completely the same as the call sequence. In this table, Enter kernel is the software trap to enter the kernel mode. Parameter check is the instructions to check the validity of a destination protection domain. Context setup is the instructions for setting up the stack pointer and the instruction pointer for the destination . Segment enable /disable and Segment register reload are instructions to switch segments. Leave kernel is the instruction to return to the user mode. This breakdown indicates that the bare hardware cost for switching segments requires 265 cycles — 87% of all — (entering and leaving the kernel, and switching segments). Thus our implementation is almost optimal in that the software overheads such as parameter checking occupies only 13% of processor cycles. From this breakdown, the optimization that eliminates the cost of entering and leaving the kernel is expected to be very effective. This is the reason we launched the development of ring switch, which does not need kernel entry.

Table 3 shows the breakdown of ring switch. Inter-segment return is the instruction to call a mobile code

Table 3: Breakdown of ring switch (round-trip)

Operation	Instructions	Cycles
Context setup	4	11
Segment register save & reload	6	22
Inter-segment return	1	86
Inter-segment call	1	80
Segment register restore	3	18
misc.	9	9
Total	24	226

Table 4: Breakdown of IPC (one-way)

Operation	Cycles
Enter kernel	86
Prepare for system call	113
Semaphore operation	208
Scheduling	770 – 1500
Return from system call	101
Leave kernel	137
Total	1415 – 2145

from a hosting application and Inter-segment call is the instruction to return from a mobile code to a hosting application. Note that control transfer to a segment with less privilege requires the inter-segment return instruction as mentioned in Section 3.2.2. Since all operations are performed in the user mode in ring switch, the cost of a round-trip cross-domain call is much smaller than segment switch. In segment switch, one round-trip cross-domain call requires two kernel entries, totally costing 446 cycles. In ring switch, calling and returning from a mobile code costs 166 cycles (86 for calling, 80 for returning) but this cost is much smaller than the cost of two kernel entries.

Table 4 lists the breakdown of cycles in IPC. The major overhead of IPC is scheduling. It takes 54% – 70% of all the cycles. Even if the kernel supports hand-off scheduling, a conventional interprocess communication can not compete with a cross-domain call between fine-grained protection domains because Intel x86 requires TLB flush in conventional interprocess communications. Once TLB is flushed, reconstructing TLB entries causes TLB miss interrupts, accessing a page table, and restarting a process. This secondary cost is not involved in Table 4 since the call was made in a tight loop.

In fine-grained protection domains, all protection domains share computing resources such as time slice and a virtual address space. Therefore, a cross-domain call requires no TLB flush as well as no scheduling. This design effectively eliminates the overhead involved in interprocess communications, even if compared with hand-off scheduling.

4.2 Macro Benchmark

This section investigates the protection overhead in a real application. We modified the Apache web-server to use fine-grained protection domains. Apache allows external modules to be dynamically loaded for extending its functionalities. These modules handles access control, authentication, CGI, SSI, logging, URL rewriting, and so on. These modules can be thought of as mobile code because they can be downloaded from Internet. We configured Apache so that each module is put in a different fine-grained protection domain.

4.2.1 Porting Apache

Incorporating fine-grained protection domains in Apache is straightforward but setting up protection policy is difficult. This is because Apache is not designed supposing that the modules may be malicious. In particular,

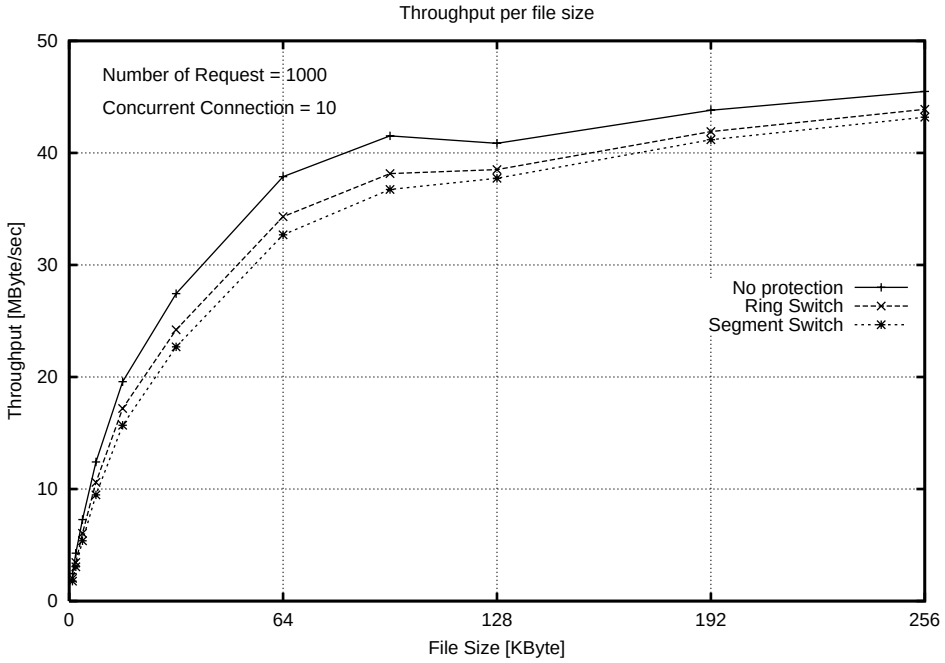

Figure 6: Throughput

setting up memory protection is difficult because Apache's private data is mixed up with modules' private data.

In this experiment, we set the protection of Apache's memory pages shared with modules to be readable and writable from modules. In practice, this will cause security problems. However, it does not matter here whether page protections are appropriate or not, because the goal of this experiment is to measure the performance. In the experimental configuration, the stack and heap areas of Apache are allocated in shared pages and made accessible from modules. Because our prototype loader lacks the facility to map static data areas to modules, library functions that return a pointer to a static data area must be wrapped so that they return a pointer to a shared memory page by copying the referent. For example, `gethostbyname()` is wrapped in the modified Apache.

As our module loader does not support dynamic linking yet, shared libraries are linked with a hosting application. All modules must call library functions via cross-domain calls except for inline functions such as `memcpy()`. This is the restriction of the current implementation and will be modified with modest efforts. Calling all library functions via cross-domain calls degrades performance of Apache significantly. Therefore, the following results must be interpreted, taking this point into account.

4.3 Results

We measured the throughput and the latency of the modified Apache, and compared them with those of the original Apache (version 1.3.9). we used ApacheBench, a benchmark program distributed with Apache server. A client requests the same file for 1000 times with 10 simultaneous connections, changing the file size from 1Kbyte to 256Kbyte. HTTP KeepAlive feature is turned on. The file size ranges from 1KByte to 256KByte. As a client, we used an Alpha 667MHz machine with 512MB memory and a PentiumII 400MHz machine with 128MB memory as a server. The server machine is equipped with 4GB Ultra-Wide SCSI disk. These machines are connected to by Gigabit Ethernet. The configuration of Apache is the same as default except that logging is disabled to avoid I /O bottleneck.

Figure 6 shows the measured throughput of three Apache servers. No protection denotes the original Apache in which no protection is provided. Segment switch and ring switch denote the Apache modified to use fine-grained protection domains. Segment switch and ring switch are degraded compared with no protection. This is because cross-domain calls are used in segment switch and ring switch. By the optimization effects, ring switch shows slightly better throughput than segment switch.

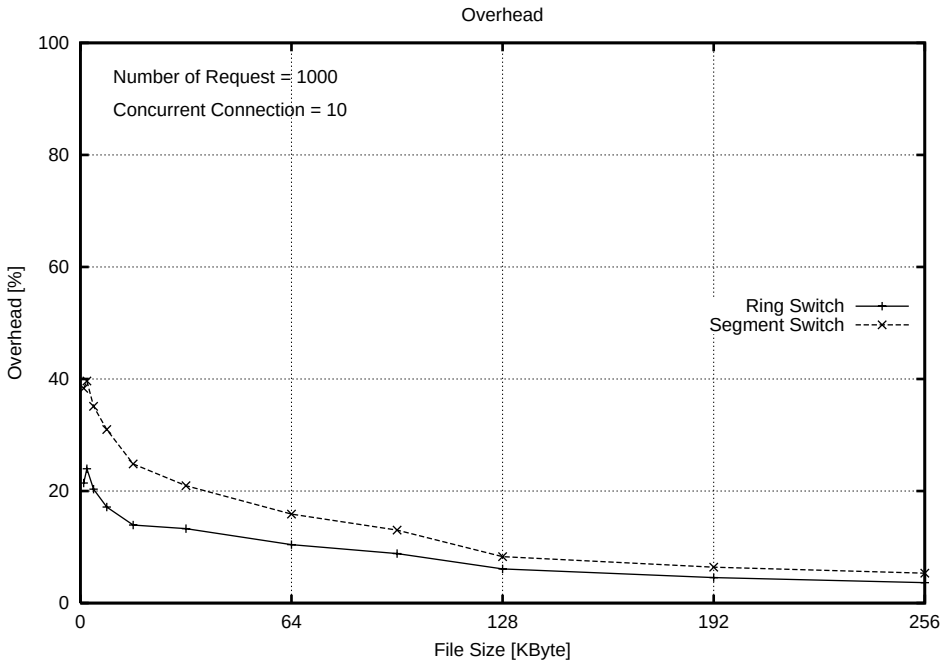

Figure 7: Overhead

Table 5: Number of cross-domain calls

The number of calls from modules	71
The number of calls from Apache	59
Total number of calls	130

To discern the overhead caused by protection, we calculated the overhead from the following expression.Figure 7 shows the overhead of segment switch and ring switch.

$$overhead = \frac{Throughput\ with\ No\ Protection}{Throughput\ with\ Protection} - 1$$

As the file size grows, the overhead becomes lower. At the file size of 64KB, the overhead is 15.8% for segment switch and 10.4% for ring switch. At the file size of 128KB, the overhead is 8.3% for segment switch and 6.1% for ring switch. This is due to the characteristics of HTTP. Almost all calls of modules are made when a negotiation is performed at the beginning of the connection. After the negotiation, the server simply transmits the requested file to the client without calling modules. Therefore, the cost of cross-domain calls does not depend on the file size. As a result, the overhead of HTTP request becomes relatively lower as the file size grows.

Figure 8 shows the results of measured latency of each HTTP request. On average, latency is increased by 0.25ms for segment switch and by 0.11ms for ring switch. At the file size of 128KB, the overhead is 2.8% for segment switch and 1.2% for ring switch. This is negligible compared with network latency.

Table 5 shows how many cross-domain calls are made in each HTTP request. The hosting application calls modules for 71 times and the called modules calls back the Apache for 59 times. Totally, 130 cross-domain calls are performed per HTTP connection. Calculating from Table 1, 130 calls amount to 198μs in segment switch and 74μ in ring switch.

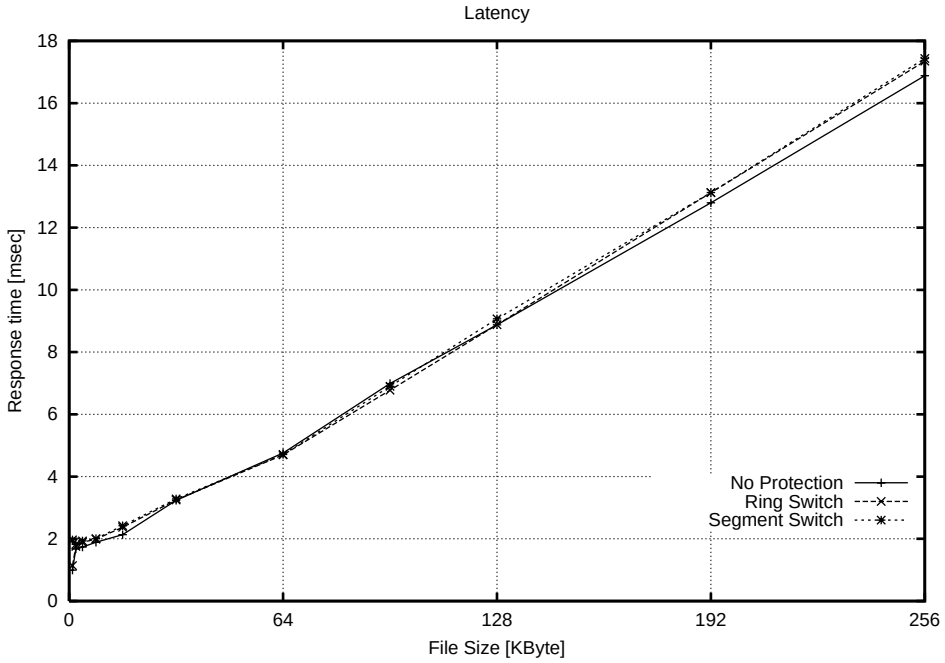

Figure 8: Latency

Table 6: Comparison of memory consumption

	No Protection Domain		With Protection Domain	
sum of virtual pages	2692KB	673 pages	54596KB	13649 pages
sum of physical pages	2648KB	662 pages	2792KB	698 pages
page table size	16KB	4 pages	68KB	17 pages

4.4 Memory Consumption

Our approach consumes memory resources than usual because it requires many entries of the page table to implement a multi-protection page table. To investigate the memory consumption, we obtained the information on memory consumption from the kernel. The target application is Apache used in the above experiments. Table 6 shows the comparison of memory consumption between Apache with and without fine-grained protection domains. The data listed in Table 6 is obtained after processing one HTTP connection. In this experiment, Apache is configured to place 30 modules in different fine-grained protection domains respectively. Among these modules, 23 modules are actually invoked per HTTP connection. The size of each module ranges from 8KB to 56KB (16KB on average). In this experiment, each fine-grained protection domain contains 1024KB virtual pages to access the stack and heap of the hosting application, and 20KB virtual pages for its own stack. When these modules are put in fine-grained protection domains, each module consumes 1056KB to 2120KB virtual pages. Totally, the 48292KB virtual pages are consumed.

As shown in Table 6, the Apache with protection consumes about 54MB virtual memory, which is 20 times larger than that of the original Apache without protection. The increased number of virtual pages has an impact on the size of the page table. The size of the page table is increased from 16KB to 68KB. This is negligible compared with the size of Apache (2.6MB). The increased size of virtual memory has an impact on TLB misses but the experimental results in Section 4.2 implies the protection overhead is modest. The total number of physical pages is increased only by 5.4% (from 2648KB to 2792KB). As fine-grained protection domains share almost all physical pages, the increase of physical pages is negligible. The reason of the increase is that each protection

domain holds its own stack pages.

4.5 Summary

We have implemented fine-grained protection domains and achieved efficient cross-domain calls. Segment switch exploiting the segmentation mechanism costs only 608 cycles, which is 6.5 times faster than IPC. And ring switch exploiting the ring protection mechanism successfully reduced the cost of cross-domain calls between a hosting application and mobile codes to 226 cycles, which is 2.7 times faster than segment switch.

Also, we have modified an Apache to use fine-grained protection domains and measured performance in practice. The overhead of throughput is only 6.1% to 15.8%, which becomes lower as the file size grows. The latency of one HTTP request is increased by only 0.11ms to 0.25ms, which is negligible compared with network latency.

From the analysis of memory consumption of the Apache, the increase of the page table was found to be negligible compared with the size of the Apache. As fine-grained protection domains share almost all physical pages, the increase of physical pages was only 5.4%.

5 Related Work

Many research activities have been done to provide intra-process protection in various contexts. The work most related to ours is Palladium [3]. Palladium exploits the features specific to Intel x86 family to provide the memory protection between a hosting application and its extensions. Palladium is different from ours in three significant points. First, Palladium does not provide any mechanisms of the protection between extensions. In our terms, Palladium does not allow more than two fine-grained protection domains to co-exist within a process. Thus all extensions must be put in the same fine-grained protection domain. If there is one malicious extension, all the other extensions are infected. Second, the protection policy is restricted in Palladium; each memory page of a hosting application is either accessible (readable and writable) or unaccessible from extensions. Our approach allows each page to be any combination of readable and writable modes. Read-only sharing is useful in various contexts but Palladium does not allow read-only sharing. Third, Palladium does not provide any mechanisms to restrict system calls. Even if such mechanisms are introduced, all extensions are given the same authority to issue system calls. Thus we can not insist on the "principle of least privilege": a mobile code should be granted the most restrictive collection of capabilities required to perform its legitimate duties, and no more [5]. Designing a protection policy is much more complicated in Palladium than in our approach.

LRPC [1] and L4 [6] attempt to minimize the overhead of interprocess communication in traditional operating systems. Since interprocess communication inherently requires to switch computing resources, our model based on fine-grained protection domains will over-perform interprocess communication. If a more efficient technology of interprocess communication is developed, we can use that technology to implement fine-grained protection domains, eliminating the overhead of switching computing resources.

Java [13] ensures code safety by the bytecode verifier and by the runtime checking. The verifier checks code integrity at the loading time, and Java VM (virtual machine) checks code safety at runtime. The safety of Java heavily relies on the correctness of Java VM. If Java VM has a security bug, the safety of the system is lost. Unfortunately, large applications are scarcely bug-free (for example, "bug of the month of `sendmail`"). We believe that an operating system should provide a final barrier using hardware-level protection.

SPIN [2] is an extensible operating system that allows an extension to be loaded and executed in the kernel. SPIN relies on the safety of Modula-3 to ensure the safety of extensions. This approach suffers from the same weakness with Java VM; if the Modula-3 compiler has a bug, the generated code may be able to corrupt the kernel.

SFI [14] ensures code safety by inserting a code sequence that confines all memory accesses in a certain memory area. In SFI, a set of instructions is inserted before all memory access instructions such as `load`, `store`, and `jump`. VINO [10], an extensible operating system, uses SFI to ensure the safety of extensions. Seltzer *et al.* [10] report that the overhead due to SFI is 5% to 200%.

Proof carrying code (PCC) [9, 8] can ensure code safety without any runtime overheads. In PCC, a mobile code is compiled by a special compiler, and a formal proof is attached to the generated code. This proof proves that the mobile code obeys the protection policy. When loading the mobile code, the proof is validated before execution. If the validation fails, the mobile code is rejected. The primary difficulty lies in automatic proof. To keep the automatic proof feasible, the expressive power of protection policy is considerably limited.

To prevent untrusted programs from issuing arbitrary system calls, Janus [5] uses the Solaris process tracing facilities. In Janus, a reference monitor is placed in a process different from the traced process. Thus the monitoring overhead is larger than our approach.

Loscocco *et al.* [7] reports that modern operating systems lack sufficient support for secure computing environments. Our attempt is an answer to them; operating systems should provide fine-grained protection domains to protect against malicious mobile codes.

6 Conclusion

We have developed a mechanism of fine-grained protection domains to protect against malicious mobile code. A fine-grained protection domain prevents a malicious code from unauthorized access to a hosting application's memory and local computing resources. To support fine-grained protection domains, we have proposed a new abstraction called a multi-protection page table. A multi-protection page table extends traditional page tables so that each virtual page can hold multiple protection modes at the same time. This abstraction allows each fine-grained protection domain to have its own memory protection policy while sharing the same single address space. It also provides an efficient cross-domain call by avoiding TLB flush and scheduling. An efficient cross-domain call also realized an efficient mechanism to check unauthorized system calls from a mobile code. This paper showed an implementation of multi-protection page tables on the most widely used Intel x86 architecture. By exploiting the segmentation mechanism and the ring protection mechanism, we achieved a cross-domain call that requires only 226 - 608 cycles in a null cross-domain call. We also demonstrated that the overall performance of the Apache using fine-grained protection domains is good enough for practical use: the overhead of throughput is 6.1 - 15.8% and the overhead of latency is 1.2 - 2.8%.

References

[1] Brian N. Bershad, Thomas E. Anderson, Edward D. Lanzowska, and Henry M. Levy. Lightweight Remote Procedure Call. *ACM Transactions on Comuputer Systems*, 8(1):37–55, February 1990.

[2] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, and Susan Chambers, Craig an Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proc. of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, pages 267–284, December 1995.

[3] Tzi-cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions. In *Proc. of the 17th ACM Symposium on Operating System Principles (SOSP '99)*, pages 140–153, December 1999.

[4] Crispin Cowan, Perry Wagle an Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer Overflows: attacks and defenses for the vulnerability of the decade. In *Proc. of the DARPA Information Survivability Conference and Expo*, 1999.

[5] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Enviroment for Unstrusted Helper Applications. In *Proc. of the 6th USENIX Security Symposium*, July 1996.

[6] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The Performance of μ-Kernel-Based Systems. In *Proc. of the 16th ACM Symposium on Operating System Principles (SOSP '97)*, pages 66–77, October 1997.

[7] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Enviroments. In *Proc. of the 21th National Information Systems Security Conference*, October 1998.

[8] George C. Necula. Proof-Carrying Code. In *Proc. of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programing Languages (POPL '97)*, pages 106–119, January 1997.

[9] George C. Necula and Peter Lee. Safe Kernel Extensions without Runtime Checking. In *Proc. of the 2nd Symposium on Operating System Design and Implementation (OSDI '96)*, pages 229–243, October 1996.

[10] Margo Seltzer, Yasuhiro Endo, Christopher Small, and Keith Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of the 2nd Symposium on Operating System Design and Implementation (OSDI '96)*, pages 213–227, 1996.

[11] Takahiro Shinagawa, Kenji Kono, Masahiko Takahashi, and Takashi Masuda. Kernel support of fine-grained protection domains for extension components. *Journal of Information Processing Society of Japan*, 40(6):2596–2606, June 1999.

[12] M. Takahashi, K. Kono, and T. Masuda. Efficient Kernel Support of Fine-Grained Protection Domains for Mobile Code. In *Proc. of the 19th IEEE International Conference on Distributed Computing Systems*, pages 64–73, May 1999.

[13] Java Team, James Gosling, Bill Joy, and Guy Steele. *The Java[tm] Language Spectication*. Addison Wesley Longman, 1996. ISBN 0-201-6345-1.

[14] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proc. of the 14th ACM Symposium on Operating System Principles (SOSP '93)*, pages 203–216, December 1993.

[15] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible Security Architecture for Java. In *Proc. of the 16th ACM Symposium on Operating System Principles (SOSP '97)*, pages 116–128, October 1997.