Fine-grained Protection Domain based on Segmentation Mechanism

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, masuda}@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@cs.uec.ac.jp
†††Japan Science and Technology Corporation

Abstract

Extensibility is a vital property of modern applications. An extension component is downloaded from Internet and loaded into an application. However, an extension component may be malicious. Thus there is a risk of the application being illegally accessed. To protect the application from malicious components, this paper proposes a *multi-protection page table*: a mechanism for providing protection among an application and components residing in the same address space. A multi-protection page table provides kernel-level fine-grained protection domains, thereby enabling efficient cross-domain calls between fine-grained protection domains. To prove that a multi-protection page table can be implemented on stock hardware, the paper shows the implementation on IA-32 (32-bit Intel Architectures). Experimental results show that a cross-domain call requires only 267-700 cycles and the performance in a real application is also good enough.

1 Introduction

Extensibility is a vital property of modern applications. An extensible application allows the user to enhance application functionalities by incorporating components independently developed by third-party vendors. These components are frequently downloaded from Internet. The user chooses a component providing the necessary functionality, downloads it from Internet, and incorporates it in the application. For example, a web-browser such as Netscape Navigator allows the user to incorporate components called plug-ins. A plug-in enables the user to browse a new kind of multimedia contents.

Unfortunately, a component downloaded from Internet may be *malicious*. Since Internet is filled with anonymous users, we can not trust all components downloaded from Internet; there may exist a malicious component. Since the component runs within the same virtual address space that the application runs, the component can access to the application's private memory. It is thus possible for the component to steal the users passwords stored in the memory. It is indispensable to provide a protection domain *within* a process so as to prevent malicious components from accessing the application's private memory. We refer to a protection domain within a process as a *fine-grained* protection domain since a fine-grained protection domain is finer than a conventional protection domain; that is, a process.

To incorporate the notion of fine-grained protection domains, the previous paper [8] of the authors has proposed a new abstraction of virtual memory, called a *multi-protection page table*. A multi-protection page table enhances a traditional page table so that each virtual memory page can have multiple protection modes at the same time. A multi-protection page table enables each fine-grained protection domain to have its own protection modes while sharing a single address space. If components are given the protection modes different from those of the application, we can prevent even a malicious component from accessing the application's private memory. In addition, since fine-grained protection domains co-exist within the same process, a cross-domain call does not require to switch any resources attached to the process. This results in a cross-domain call more efficient than a traditional interprocess call.

In the previous paper, we have shown that a multi-protection page table can be implemented on RISC processors such as SPARC, MIPS, and Alpha. The present paper shows that a multi-protection page table can be implemented on IA-32 (32-bit Intel Architecture). IA-32 is the most widely used architecture in the world but the technique shown in the previous paper can not be applied to IA-32. That technique extensively exploits modern features of RISC processors such as tagged TLB but IA-32 lacks such modern features. Therefore it poses many challenging problems to develop an implementation technique of a multi-protection page table on IA-32. Since IA-32 is used in 90% of desktop computers, developing the technique is significant for practical reasons.

Experimental results indicate that a multi-protection page table on IA-32 is efficient. In a micro benchmark, it has been shown that a cross-domain call requires only 267–700 cycles. To measure the performance of a real application, we modified an Apache web-server so that each component of the Apache (a default Apache consists of 31 components) is placed in different fine-grained protection domains. In the macro benchmark using the modified Apache, the overhead incurred by multi-protection page table is at most 14.3% when the size of a html file is 64KB.

The rest of the paper is organized as follows. Section 2 describes the protection model based on multi-protection page tables. Section 3 shows the implementation of multi-protection page tables on IA-32. Section 4 reports experimental results. Section 5 relates the paper with other work. Section 6 concludes the paper.

2 Fine-grained Protection Domain

2.1 Multi-Protection Page Table

In our protection model, a process holds more than one fine-grained protection domain within itself (Figure 1). All the fine-grained protection domains in the same process share the virtual address space of the process.

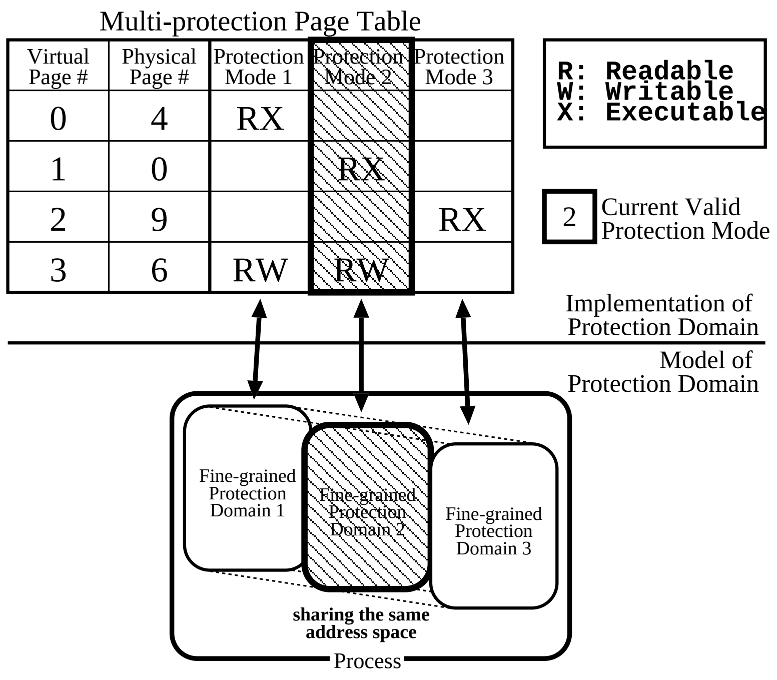
A *multi-protection page table* is an extension of a traditional page table. Each row of the multi-protection page table consists of more than one protection mode of a virtual memory page, in addition to the mapping of the virtual memory page and a physical memory page. Each column of the protection mode represents the protection mode of each fine-grained protection domain. Namely, each fine-grained protection domain has its own column of the protection mode (see Fig. 1). At any instance of time, only the protection-mode column corresponding to the currently executing domain is effective.

Using this abstraction of virtual memory, we can prevent malicious components from accessing the application's private memory. For example, Figure 1 shows that the virtual page #0 is readable and executable in the protection domain #1, whereas it is unaccessible in the other protection domains. In this way, the application's pages are protected from malicious components.

2.2 Cross-domain Calls between Fine-Grained Protection Domains

Our system provides the facility of cross-domain calls by which an entry point of one fine-grained protection domain is invoked from another fine-grained protection domain. To call an entry point in another fine-grained protection domain, a calling thread presents to the kernel an identifier of the called component and an identifier of the entry point. The kernel assigns an identifier (a small integer) to every component when the component is loaded into the application.

When the calling thread traps into the kernel, the kernel switches the effective column of the protection modes. Then the kernel upcalls the called component, passing the identifier of the calling component


Multi-protection Page Table
R: Readable
W: Writable
X: Executable
2 Current Valid Protection Mode
Implementation of Protection Domain
Model of Protection Domain
Fine-grained Protection Domain 1
Fine-grained Protection Domain 2
Fine-grained Protection Domain 3
sharing the same address space
Process

Virtual Page #	Physical Page #	Protection Mode 1	Protection Mode 2	Protection Mode 3
0	4	RX		
1	0		RX	
2	9			RX
3	6	RW	RW	

Figure 1: Multi-protection page table

and the identifier of the entry point. The called component checks whether the calling component has the permission to invoke the specified entry point. If it has, the thread jumps into the entry point. Otherwise, the component returns an error code to the kernel.

Since all fine-grained protection domains share all computing resources such as time slice, memory mappings, and so on, the kernel has only to switch the protection modes of memory pages, without any complicated operations such as scheduling, TLB flush, and so on. As a result, our mechanism of protection provides a much more efficient cross-domain call compared with traditional interprocess communications.

2.3 Protection Policy

Each process has exactly one *policy component* that determines a protection policy of the process. A policy component determines which memory page is accessible from which fine-grained protection domains. The kernel obeys the policy supplied by the policy component. The policy component itself is in one of the fine-grained protection domains and is protected from malicious components.

2.4 Example of Web-Browsers

As a concrete example of our protection model, consider the case where a web browser loads a plug-in from Internet. The browser and the plug-in are put in different fine-grained protection domains. The web-browser determines which memory page is accessible from the plug-in and thus plays the role of a policy component. In most cases the memory protection is set so that the plug-in can execute its own code, read and write its own data and stack segments while the memory areas of the browser are set to be unaccessible from the plug-in. Thereafter the plug-in can invoke the browser's API by using cross-domain calls.

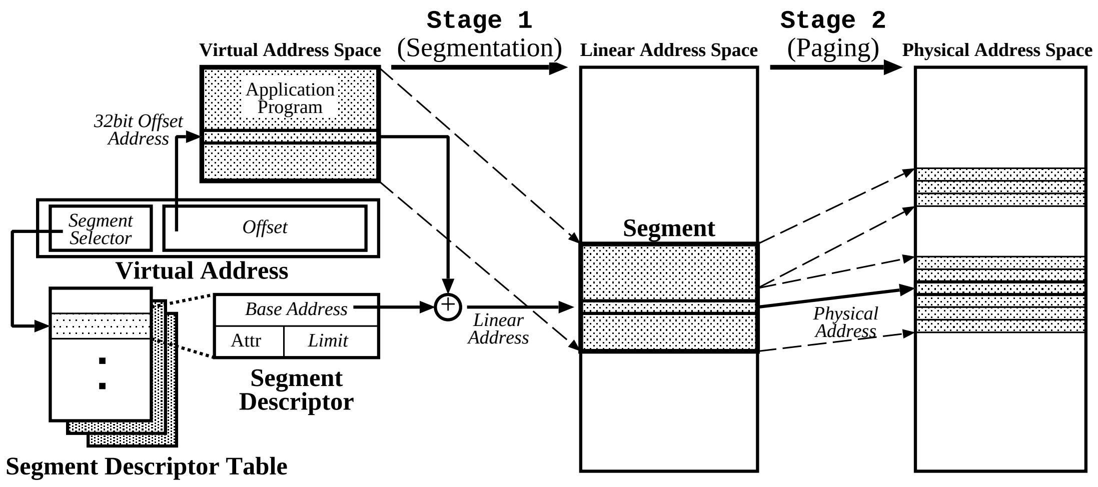


Figure 2: Segmentation mechanism

3 Implementation on IA-32

3.1 Implementation of Multi-protection Page Table

A multi-protection page table can be implemented on modern RISC processors without any special hardware support. Actually, we have already shown the implementation technique on RISC processors such as SPARC, Alpha, and MIPS in the previous paper [8]. Modern RISC processors are all equipped with tagged TLBs, and the technique exploits tagged TLBs as a central part of the implementation. Unfortunately, IA-32 lacks tagged TLBs and thus the technique can not be applied to the IA-32 family. IA-32 is the most widely used architecture in the world and thus it is significant for practical reason to show the implementation technique on IA-32.

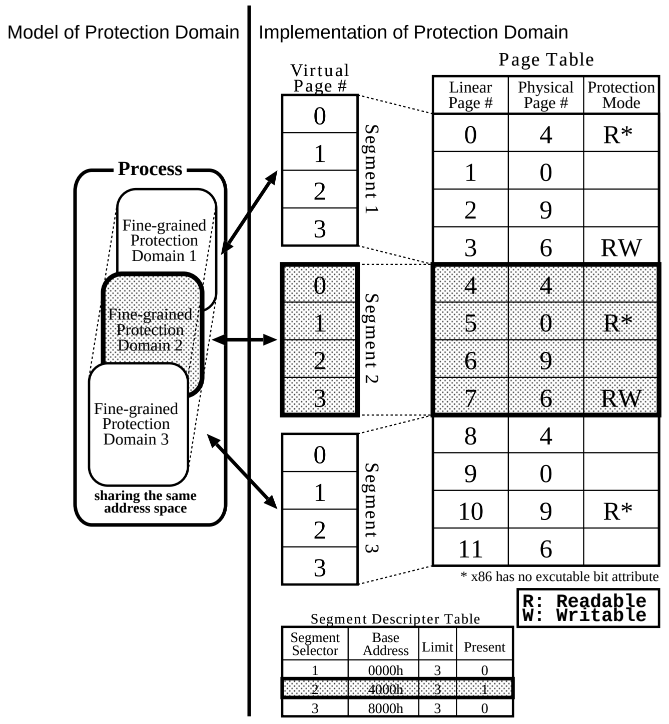
3.1.1 IA-32 Segmentation

Segments are protected from each other by hardware. An illegal access, either beyond the segment limit or without legal permission, generates a general-protection exception. A virtual address of IA-32 consists of two integers: a 16-bit *segment selector* and a 32-bit *offset*. A virtual address is translated into a physical address in the two stages: (1) the segmentation stage and (2) the paging stage (see Figure 2).

In the first stage, the segmentation mechanism translates a virtual address into a *linear address*. A linear address is different from a virtual address. It is invisible from the user program and appears only in the process of the translation. A virtual address is translated into a linear address by adding the *base* address of the segment and the offset embedded in the virtual address. The base address of the segment is obtained from the segment descriptor, which is specified by the segment selector embedded in the virtual address. A segment descriptor specifies the base address, the size, and the access rights, and the present flag of the segment.

In the second stage, the paging mechanism translates a linear address into a physical address. This translation is performed in the same way as ordinary paging mechanism except that IA-32 translates a linear address, not a virtual address, into a physical address.

A segment is enabled/disabled by setting/clearing the present flag bit in the segment descriptor. Only a valid segment is accessible in the user mode and an access to an invalid segment generates a segment-not-present exception. It is impossible for the user program to enable an invalid segment because rewrit-


Model of Protection Domain
Implementation of Protection Domain
Process
Fine-grained Protection Domain 1
Fine-grained Protection Domain 2
Fine-grained Protection Domain 3
sharing the same address space
Virtual Page #
Segment 1
Segment 2
Segment 3
R: Readable
W: Writable

Page Table

Linear Page #	Physical Page #	Protection Mode
0	4	R*
1	0	
2	9	
3	6	RW
4	4	
5	0	R*
6	9	
7	6	RW
8	4	
9	0	
10	9	R*
11	6	

* x86 has no excutable bit attribute

Segment Descripter Table

Segment Selector	Base Address	Limit	Present
1	0000h	3	0
2	4000h	3	1
3	8000h	3	0

Figure 3: Implementation of a multi-protection page table

ing the present flag in the segment descriptor is possible only in the kernel mode.

3.1.2 Segmentation-based Implementation

To prevent a thread in one fine-grained protection domain from accessing the memory areas of other domains, each fine-grained protection domain is assigned to one segment, as shown in Figure 3. By doing this, a component can not access to the memory areas of other components because a segment is isolated from each other by hardware. But a malicious component can access illegally to other segments by changing the segment selector to another domain's selector.

To prevent the malicious component from accessing illegally, all segments are disabled by the kernel except for the segment associated to the currently valid domain. Thereby it becomes impossible for the malicious component to access other components because a user program can not enable any segments.

A cross-domain call can be implemented as follows. The kernel provides a software interrupt for a cross-domain call. When a thread interrupts the kernel, the kernel rewrites the present flags in the segment descriptors of the caller and the callee segments. The caller's segment is disabled and the callee's segment is enabled. Then the kernel saves the context of the caller and restores the context of the callee. This is all to be done: the kernel need not to schedule, flush TLBs, and other complicated things because all domains in the same process share the computing resources of the process.

To enable each fine-grained protection domain to have different page protection modes, each segment is mapped to a different area in the linear address space. Figure 3 shows that the segment #1 is mapped to the linear pages #0 to #3, the segment #2 is mapped to pages #4 to #7, and so on. In this way each page in the segments can have its own page protection modes. For example, Figure 3 shows that page #0 is read-only in the segment #1, while it is unaccessible in the segment #2.

The advantage of this approach is that we can avoid the cost of TLB miss and cache miss when switching protection domains because all segments share a single page table. Also, we can avoid the cost of switching the resources assigned to the processes because all components and application can share it.

3.2 Optimizing cross-domain calls

A cross-domain call between an application and extension components can be optimized by exploiting the ring protection of IA-32. This optimization is effective because the cross-domain calls between an application and extension components will be performed more frequently than between the extension components themselves. This section describes the optimization method of cross-domain calls after explaining the ring protection.

3.2.1 Ring protection

The ring protection limits the accessibility of a segment according to its privilege level. A segment descriptor specifies the privilege level of the segment. IA-32 provides four privilege levels, numbered from 0 to 3. The ring 0 is the most privileged and the ring 3 is the least privileged. The processor prohibits direct accesses to the segment with greater privilege level. Normally, a kernel resides in the segment with privilege level 0, and a user program resides in the segment with privilege level 3.

To call procedures with more privileged level such as system calls, a program calls a gate. A gate is the mechanism to communicate safely with the procedures with different privilege levels. A gate of IA-32 specifies the entry point (a segment selector and offset) of the callee procedure. A call to the gate causes the shift to the more privilege level and switching the stack.

3.2.2 Optimization

The key idea of this optimization is that an application is trusted and thus there is no need to prevent an application from making accesses to extension components. To prevent the extension components from accessing the application's segment, the kernel sets the privilege level of the application to the ring 2 and those of the components to the ring 3. By doing this, an application can call a component without interrupting the kernel but it is protected from the components.

Note that this optimization is applied only to cross-domain calls between an application and components. A cross-domain call between a component and a component, the software interrupt is required as mentioned in Section 3.1.2.

4 Experiments

4.1 Set up

All experiments are performed on the following system. The hardware is PC with Intel PentiumII 400MHz processor and 128MB memory. The kernel is our prototype implementation based on the Linux 2.2.12 kernel.

As described in Section 3, we implemented two mechanisms: (1) the segmentation-based mechanism and (2) the ring-protection-based mechanism. Hereinafter, we call the former `segment` and the latter `ring`.

4.2 A micro benchmark

In order to estimate the baseline cost of a cross-domain call, we measured the cycles of a null cross-domain call. A call was made repeatedly in a tight loop to reduce the influence of cache misses. The

Table 1: Cycles for a null cross-domain call

Method	Time	Cycles
Segment	1.75 μs	700
Ring	0.67 μs	267
IPC	15.75 μs	5600

Table 2: Cycle breakdown of `segment`

Operation	Cycles	
	User mode	Kernel mode
Caller stub	51	
Callee stub	41	
Software Interrupt(kernel trap)		86 * 2
Parameter check		16 * 2
Context setup		12 * 2
Segment enable/disable		12 * 2
Segment register reload		15 * 2
misc		26 * 2
Interrupt return(kernel return)		137 * 2
Total	92	304 * 2

number of cycles was obtained using the performance counter of PentiumII. We also measured the average cycles of a traditional interprocess communication (IPC) for comparison.

As shown in Table 1, `segment` is about 8 times faster than `IPC` and `ring` is about 2.6 times faster than `segment`.

Table 2 shows the breakdown of the cycles for a null cross-domain call by `segment`. Since `segment` requires two kernel entries for every one cross-domain call, the operation in the kernel mode is executed twice (denoted by “*2”). The cost of a null cross-domain call is 304 cycles. A stub code for interrupting the kernel is not optimized, so we can expect this cost is much more reduced.

Table 3 shows the breakdown of cycles for a null cross-domain call by `ring`. This approach requires no kernel entry. The cost of a cross-domain call is 86 cycles for calling a module and 80 cycles for returning to an application. Some additional operations such as “context setup” and “segment register save & reload” are performed by the application.

4.3 A Macro Benchmark

In order to estimate the performance of a real application, we measured the performance of the *modified* Apache web server. The Apache allows extension components to be loaded at runtime to extend its functionalities. By default, it consists of 31 components. We regard these components as distrusted, as if they came from Internet and were loaded dynamically. In this scenario, we put them in fine-grained protection domains for protection purpose.

To make the porting of the Apache easier, the standard C library is put in the protection domain of the Apache body. Therefore, all library calls from the components are made via cross-domain calls, although some functions such as `memcpy()` are inline-expanded. This setup causes many unnecessary cross-domain calls because some functions are apparently safe to be in the protection domains of the components. Thus the performance shown in this section is the worst case; we can expect the better performance.

Table 3: Breakdown of cycles for Privilege Level 2

Operation	Cycles
Context setup	11
Segment register save & reload	18
Inter-Segment return(upcall)	86
Inter-Segment call	80
Segment register restore	18
misc	54
Total	267

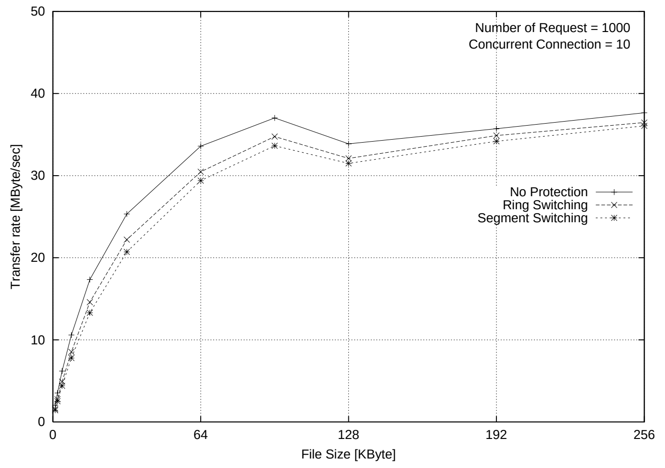


Figure 4: Throughput

The protection policy used in this experiment is naive. All the application data, including a stack, are shared among the components. This allows to pass a pointer to a stack beyond the protection boundaries. Some functions such as `gethostbyname()`, which return a pointer to the static region, is wrapped to return a pointer to the shared region by copying the result data. This slightly degrades the measured performance.

As a benchmark program, the experiments used the ApacheBench, a benchmark that comes with the Apache distribution. The server is based on the Apache 1.3.9. All components of the Apache are enabled by configuration and put into different protection domains. The configuration of the Apache is default except that the logging is off to avoid the I/O becoming bottleneck.

The client requests the same file 1000 times, with 10 simultaneous connections, using the HTTP KeepAlive feature. The file size ranges from 1KByte to 256KByte. The results are obtained from the output of the ApacheBench. We run the client on the same machine with the server to avoid the network becoming bottleneck.

Figure 4 shows the results of the experiments. The performance of our approach is constantly worse than "No Protection", but the degradation by using our approach is at most in the order of 3.5MByte/sec at a file size of 96KB. Our approach of `ring` shows a little better performance than `segment`.

Figure 5 shows the overhead estimated from Figure 4. As the file size grows, the overhead becomes the lower. For example, at the file size of 64KB, the overhead is 14.3% for `segment` and 10.3% for ring, while at the file size of 128KB, 7.6% for `segment` and 5.5% for `ring`.

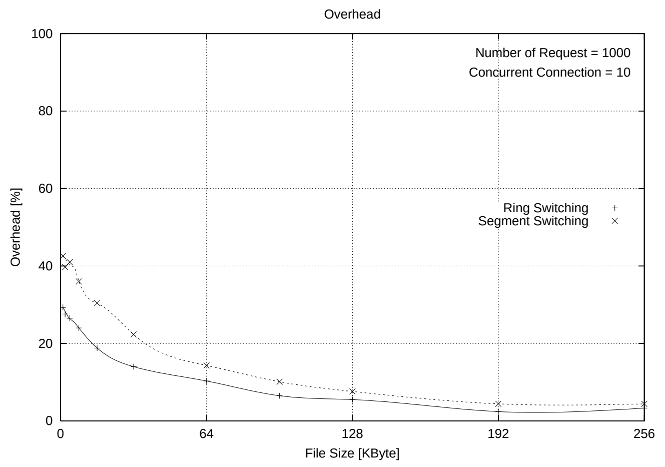


Figure 5: Overhead

Table 4: Number of cross-domain calls

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

This is due to the characteristic of HTTP. Most cross-domain calls between components are made when a negotiation is performed at the beginning of a connection. After the negotiation, the server simply transmits the requested file to the client, without any complicated operations. Then the cost of the cross-domain call does not depend on the file size. Consequently, the overhead of a HTTP request becomes relatively lower as the file size grows.

Figure 6 shows the latency of each request. The average overhead is about 0.26ms for `segment` and 0.17ms for `ring`. This result also shows that the overhead of cross-domain calls is hardly influenced by the file size.

Table 4 shows that how many cross-domain calls are made for each HTTP request. The Apache calls components 71 times and these components call the functions the Apache provides 59 times. Thus, 130 cross-domain calls are made per one connection. This means that 227.5 μs for `segment` and 87.1 μs for `ring` is spent to perform cross-domain calls. In comparison with the results of Figure 6, this result shows that the overhead of our approach is mainly due to the cost of cross-domain calls.

5 Related Work

The protection mechanisms developed so far are classified into the kernel-based approach and the language-based approach. The kernel-based approach often utilizes the hardware for managing virtual memory (MMU). The conventional operating systems only support a *coarse*-grained protection domain that is intended for protecting a process. As a result, the traditional protection domains are too heavyweight to be used for components. To the contrary, the language-based approach provides a *fine*-grained protection domain by using different techniques for language processing. The language-based techniques developed so far can be classified into the three approaches: (1) virtual machine, (2) code modification, and

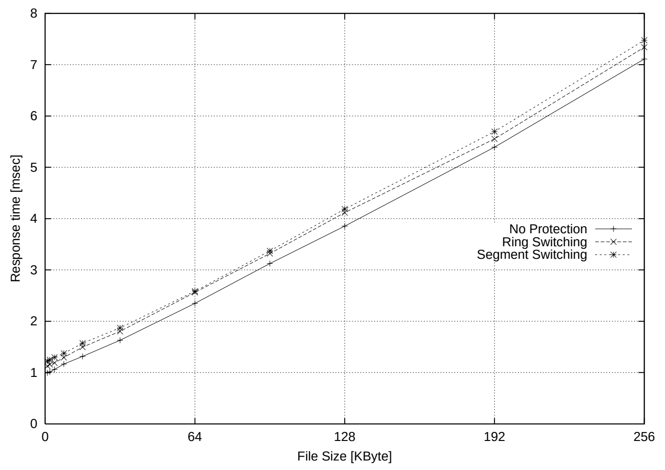


Figure 6: Latency

(3) proof carrying code.

In the virtual-machine approach, a component is expressed in byte-code and a virtual machine interprets instructions one by one. Since the virtual machine can check, if necessary, the validity of all the byte-code instructions, a fine-grained protection domain can be implemented easily. The apparent disadvantage of this approach is inefficiency due to interpretation overhead.

The second approach, code modification, inserts a code sequence into the component code in order to check the safety of the code. The SFI approach proposed by Wahbe et al. [10, 1] checks memory accesses of components by inserting a set of instructions before all the memory access instructions. This approach incurs constant overhead independently of the number of cross-domain calls.

The third approach, proof carrying code [7], provides a mechanism by which a kernel can determine that it is safe to execute a component. A special compiler called a *certificating compiler* generates binary codes in a special form called proof-carrying code or PCC. Each PCC binary contains, in addition to the native code, a formal proof that the code obeys the protection policy determined by the kernel. Then the kernel validates the proof before executing the code. The main difficulty of this approach is in generating the safety proof automatically. To reduce this difficulty, the expressive power of protection policies is limited so that automatic proof is feasible.

Many literatures [9, 11] report that the kernel-based approach is too heavyweight, but we claim that this is due to the lack of abstraction for fine-grained protection domains in conventional operating systems. Recent results of research on interprocess communication are encouraging since even the context switch between conventional coarse-grained protection domains can be implemented efficiently (as reported in LRPC [2], Spring [5, 4], and L4 [6]). In fact, recent research shows a efficient implementation of intra-address space protection mechanism [3].

6 Conclusion

We have developed a mechanism of fine-grained protection domains. Fine-grained protection domains co-exist within a single address space and prevent malicious components from accessing unauthorized memory areas. 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 memory page can have multiple protection modes at the same time. This paper showed an implementation of multi-protection page tables on the IA-32. By exploiting segmentation mechanism, we achieved a cross-domain call that requires only 267 - 700 cycles in a null cross-domain call. We also demonstrated that the cross-domain calls are efficient enough in a real application using an Apache web server.

References

[1] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proc. of ACM Conf. on Programming Language Design and Implementations*, pages 127–136, May 1996.

[2] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1), Feb. 1990.

[3] Tzi cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of the 17th ACM Symposium on Operating System Principles*, pages 140–153, Dec. 1999.

[4] Graham Hamilton and Panos Kougiouris. The Spring nucleus: A microkernel for objects. In *Proceedings of the Summer 1993 USENIX Conference*, pages 147–159, June 1993.

[5] Graham Hamilton, Michael L. Powell, and James G. Mitchell. Subcontract: A flexible base for distributed programming. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 69–79, December 1993.

[6] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 237–250, Dec. 1995.

[7] G. C. Necula and P. Lee. Safe kernel extensions without runtime checking. In *Proc. of USENIX Symp. on Operating System Design and Implementation*, pages 229–243, Oct. 1996.

[8] M. Takahashi, K. Kono, and T. Masuda. Efficient kernel support of fine-grained protection domains for mobile code. In *Proc. of IEEE 19th Int. Conf. on Distributed Computing Systems*, pages 64–73, 1999.

[9] Tommy Thron. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, September 1997.

[10] R. Wahbe, S. Lucco, E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proc. of ACM Symp. on Operating System Principles*, pages 203–216, 1993.

[11] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architecture for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.