A HIERARCHICAL PROTECTION MODEL FOR PROTECTING AGAINST EXECUTABLE CONTENT

Takahiro Shinagawa
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

Kenji Kono† and Takashi Masuda
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
† PRESTO, Japan Science and Technology Corporation

ABSTRACT
Executable content poses a threat of unauthorized access because it contains program code running on the user's machine. Protecting against executable content is difficult because of the inevitable flaws in the implementation of protection mechanisms. This paper introduces a *hierarchical protection model* to tolerate flaws in protection mechanisms. This model improves both the *granularity* and the *robustness* of protection mechanisms by nesting two protection domains: a *level-1 protection domain* to provide fine-grained access control on executable content, and a *level-2 protection domain* to act as a fail-safe mechanism. We achieved an efficient implementation of the hierarchical protection model that incorporated the *fine-grained protection domains* proposed in our previous paper.

KEY WORDS
Executable Content, Security, Operating System, Fine-grained Protection Domain

1 Introduction

Executable content is becoming a popular form of Internet content. They provide a greater power of expression, such as animation and interaction, by allowing the dynamic creation of content. Executable content contains some program code that is executed on a user's machine to dynamically produce the actual content on demand. Examples of executable content include Java applets, ActiveX, and JavaScript. Unfortunately, executable content carries the threat of unauthorized access. Executing such programs can also give crackers attempting unauthorized access the power to cause critical damage — such as destroying files — to the user's machine. In fact, many security threats involving executable content have been widely reported [1, 2, 3, 4].

Protecting against the malicious use of executable content is difficult because of the inevitable flaws in the implementation of the protection mechanisms. According to software engineering research, statistically speaking all programs are likely to contain an amount of bugs proportional to the size of the program code: 1000 lines is estimated to contain 5-30 bugs [5], while another estimate states that even the products of leading-edge software companies contain 0.2 bugs per 1000 lines of code [6]. Moreover, the code size of the protection mechanisms tends to be large because these mechanisms need the complex program code to implement fine-grained access control for the resources of the user's machine. Fine-grained access control is necessary for properly protecting sensitive resources, while at the same time allowing access to the necessary resources for the executable content code to be run. The need for finer-grained access control makes it more difficult to implement the protection mechainsm without bugs. This means that there is a tradeoff between the *granularity* and the *robustness* of the protection mechanisms.

This paper introduces a *hierarchical protection model* that achieves a fail-safe protection mechanism to tolerate flaws in the implementation. In this model, multiple protection domains are nested to balance the tradeoff between the granularity and robustness. As shown in Figure 1, the inner protection domain, called the *L1 (level-1) protection domain*, is assigned to executable content, performing fine-grained access control. Since it is difficult to implement a robust L1 protection domain, the outer protection domain, called the *L2 (level-2) protection domain*, supplements the inner protection domain to reduce the risk of unauthorized access. The L2 protection domain is assigned to the host application and suppress the damage of the attacks, even if the flaws in the L1 protection domains is exploited in an attack. The L2 protection domain only provides coarse-grained access control to make it easy to reduce bugs. This is a easier and more cost-effective approach than implementing a perfect protection mechanism with both finer granularity and robustness. Note that this paper only deals with a two-level hierarchy aiming at protection against executable content.

For efficient implementation of the hierarchical protection model, we exploit the *fine-grained protection domains*, a protection mechanism we proposed in previous papers [7, 8, 9]. The fine-grained protection domain allows multiple protection domains to coexist with others inside a single process. Fine-grained protection domains achieve lightweight cross-domain calls by avoiding context

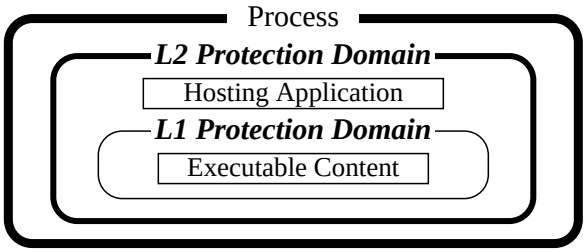

Figure 1. The hierarchical protection model

switches. We have implemented the hierarchical protection model exploiting fine-grained protection domains and applied it to protecting against several executable contents.

The rest of this paper is organized as follows. Section 2 explains the threat of unauthorized access through running executable contents. Section 3 describes the implementation of the model and Section 4 shows example applications of the model. Section 5 reports the experimental results. Section 6 describes related work and Section 7 summarizes the paper.

2 Executable Content

In this section, we first discuss the threat and difficulty in protecting against executable content. Then, we show the classification of executable content used in the following sections.

2.1 Threat

Executable content is convenient for malicious attackers because of the anonymity it affords. Since executable contents pass through the Internet, identifying the author is difficult. Malicious attackers can distribute executable content without the risk of being identified. Furthermore, executable content is technically convenient for attacks. Firstly, getting unauthorized access when using it is easy. It directly runs on a user's computer so doesn't need an attempt to intrude into the computer from the outside to gain unauthorized access. Secondly, it has many opportunities to run. It runs automatically whenever the user browses the content without any installation steps. Therefore, it will run many times and on many computers on the Internet. Thirdly, it can be easily and rapidly spread all over the world. By being distributed via the Internet and acting as a virus or a worm, it can achieve extensive unauthorized accesses.

Malicious executable content can get unauthorized access by exploiting flaws in the protection mechanism. It attacks the flaws by using various techniques, such as buffer overflow attacks [10], and *hijack* the host application. Then, malicious executable content can indirectly get unauthorized access by controlling the host application.

2.2 Difficulty

The protection mechanism for protecting against executable content must be able to selectively allow or deny access to a resource in a user's machine. Since any executable content requires a certain amount of resources on the host machine, executable content is unable to run if all resource access is denied. However, executable content may cause harms to the host machine if unnecessary resources are allowed to access. Therefore, the protection mechanism needs fine-grained access control on resources. In fact, the protection mechainsm of Java, for example, now provides a finer-grained access control than the initial sandbox mechanism [11]. Unfortunately, the fine-grained access control mechiansm is difficult to be implemented without flaws because it needs complex and large program code.

There is a tradeoff between the granularity and the robustness of the protection mechanisms. To improve the granularity of the protection mechiansm, large implementation is required which degrades the robustness of the protection mechiansm. Therefore, it is diffcult to provide the both fine-grained and robust protection mechiansm by single implementation. We address this problem by using multiple protection mechiansms each of which have different granularity and robustness.

2.3 Classification

We classify executable content into two types: the *binary-type* and the *script-type*. We consider the mechanism and concrete examples of unauthorized access for each type.

Binary-Type The binary-type is a type of executable content that contains machine-dependent code. This type of content is linked with the host application and runs in the same process. The binary-type content get access to resources by calling the API provided by the host application, or directly issuing system calls. Preventing the binary-type content from making unauthorized access is difficult in existing operating systems. Since binary-type content runs in the same process as the host application, it also has the same authority to access resources. Therefore, controlling the access of binary-type executable content is difficult.

ActiveX is an example of the binary-type executable content. ActiveX depends on a digital signature and/or host-based access control which determine whether the executable content is trustworthy or not, thereby ensuring the safety of the host computer. However, an attack has been reported that exploits flaws in ActiveX [3].

Script-Type The script-type is a type of executable content that contains machine-independent code that is interpreted by an interpreter program. The code is written in a specific programming language and interpreted to run by the relevant interpreter. Examples of the script-type content

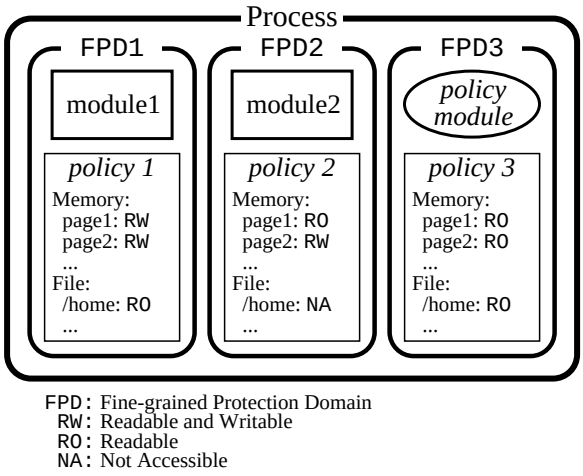

Figure 2. Fine-grained protection domains

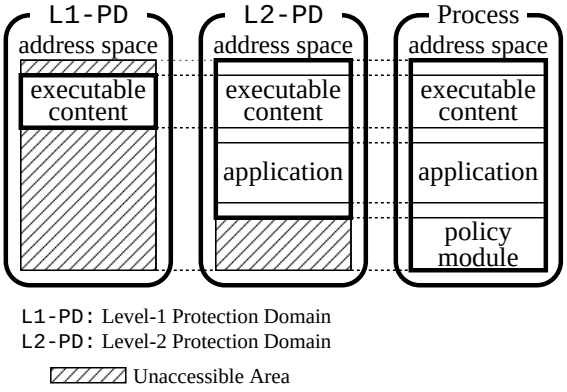

Figure 3. Memory protection

include JavaScript and VBScript. Furthermore, PostScript and PDF, popular formats for electrical documents, can also be classified into the script-type, since they are written in a language that has, to some extent, characteristics of programming languages. The script-type content is relatively more difficult to use for unauthorized access than the binary-type content. Directly accessing memory is usually difficult because the interpreter ensures type-safety at the language level. It is also difficult to directly access resources because the interpreter controls the access.

However, the script-type content can get unauthorized access by exploiting flaws in interpreters. For example, ghostscript, an interpreter for PostScript, had bugs that enabled PostScript to access any files on the host computer [1]. Adobe Acrobat Reader, a PDF reader, was also vulnerable to buffer overflow attacks [4].

3 Implementation

To implement the hierarchical protection model, we used the *fine-grained protection domain*, a protection mechanism that we proposed in previous papers [7, 9]. In this section, we first give a brief description of the fine-grained protection domain. Then, we show an implementation of the L1 and L2 protection domains that exploit fine-grained protection domains.

3.1 Fine-grained protection domain

The fine-grained protection domain is a kernel-level protection mechanism that allows multiple protection domains to co-exist within a single process, thereby enabling multiple modules composing the process to have different levels of access authorities for the system resources. Each module is assigned one fine-grained protection domain that grants that module's authority to access system resources such as memory, files, and so on. Therefore, fine-grained protection domains enable fine-grained access control for every system resource, including memory, file, network, and other resources.

Each fine-grained protection domain is associated with a *protection policy*, which defines each domain's access right to resources (see Figure 2). For example, we can control memory access at the granularity of the page. If there are two fine-grained protection domains, we can set a memory page to be readable and writable by one domain while read-only for the other. For other resources such as files and network communications, each fine-grained protection domain is granted different access rights.

The protection policies of fine-grained protection domains are implemented by a single module called the *policy module*. A policy module cooperates with the kernel to enforce a specific policy on each fine-grained protection domain. It configures the permissions to access memory pages, and directs the kernel to upcall the policy module when a fine-grained protection domain issues a system call. On the upcall, the policy module determines whether the issued system call is permitted or not, by inspecting the system call arguments.

The fine-grained protection domain has two notable features. The first is that all the fine-grained protection domains share the same name space. The second is that it is designed to considerably reduce the cost of cross-domain calls. Therefore, the overhead of using fine-grained protection domains is suppressed

3.2 L1 protection domain

We use a fine-grained protection domain for implementing the L1 protection domain. Fine-grained protection domains are suitable for L1 protection domains because of their features, such as fine-grained access control. We assign a fine-grained protection domain to an executable content, allowing fine-grained access control over the executable content.

The fine-grained protection domain assigned to an ex-

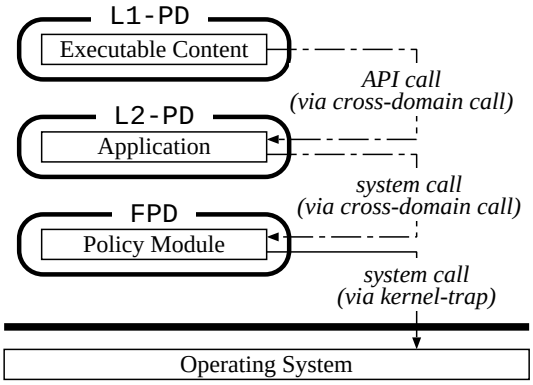

Figure 4. The path to access resources

ecutable content performs memory protection and controls access to system resources such as files and network communications. Fine-grained access control over memory is performed by a fine-grained protection domain at the granularity of the page. As shown in Figure 3, an L1 protection domain (abbreviated to L1-PD) only restricts memory access only to the pages owned by the executable content itself. It prevents the executable content from attacking the memory data of the host application.

Fine-grained access control over system resources is performed in co-operation with the host application. This host application checks the resource accesses of the executable content so that the content can not obtain unauthorized access. As shown in Figure 4, the content is forced to call the application via a cross-domain call to access any system resource. Directly issuing system-calls is prohibited. The application needs to provide a proper set of APIs to safely access system resources and ensure that unauthorized access is prevented. Calling the APIs, and not issuing system calls, is forced by the L1 protection domain by using the features of fine-grained protection domains.

The specification of the APIs must be designed and implemented properly so that it can ensure safety. It is important to design a proper set of APIs because these decide the granularity of protection. The application can design any APIs in order to perform resource protection at the desired granularity. It is also important to properly implement the APIs. Because the implementation flaws may enable unauthorized access, the APIs must be implemented as carefully as possible. However, the implementation complexity of the L1 protection domain is not specific to our approach. As discussed in the previous section, implementations of fine-grained access control tend to be complex. We argue that an L2 protection domain, described in the next section, should be prepared to lessen the damage caused by the implementation flaws in the L1 protection domain.

3.3 L2 protection domain

We use another fine-grained protection domain as the L2 protection domain and assign it to the host application. It confines the host application to making accesses only to the intended operation resources.

Resource access by the host application is controlled by a policy module. The policy module acts as a reference monitor and checks resource accesses by the host application at the system-call level. As shown in Figure 4, the system-calls issued by the host application are intercepted by the module. It then checks whether the access is allowed or not. If it is allowed, the policy module performs the system call on behalf of the host application. If not, the host application is killed because it violates the security policy.

4 Example Application

In this section, we show two examples that apply a hierarchical protection model to existing applications. Both examples incorporate a hierarchical protection model without the need for modification to the applications. They deal with the two types of executable content, the *binary-type* and the *script-type*, described in Section 2.3. The first example is a web-browser that handles binary-type content. We used a plug-in to incorporate the protection model without modifying the browser. The second example is a document viewer that handles script-type content used in an electronic document. We modified an ELF loader to incorporate the protection model without modifying the viewer.

4.1 A web-browser

The first example is a web-browser that incorporates the L1 protection domain to protect against binary-type executable content. We achieved protection by using the technique described in Section 3.2 that exploits the fine-grained protection domain for an L1 protection domain.

We defined an original MIME type as the binary-type content stored in Intel x86 ELF format. We assume that this type of executable content is used like Java applets. It is downloaded with HTML documents and executed by the browser to draw images embedded in the document. Unlike Java applet, though, this type of executable content is run directly on the processor of the host machine.

We used a plug-in to incorporate the L1 protection domain without modifying the browser. The plug-in itself loaded the executable content into the address space of the browser and assigned the content an L1 protection domain. The plug-in configured the L1 protection domain to confine memory access and to perform access control over the system resources.

The plug-in mediated between the executable content and the browser to perform access control on resources (see Figure 5). When the browser needed to display an image in the document, it requested the plug-in to draw the image.

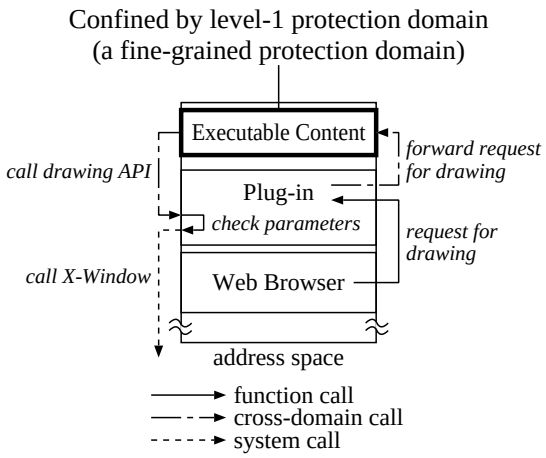

Figure 5. L1 protection domain incorporated in a web-browser

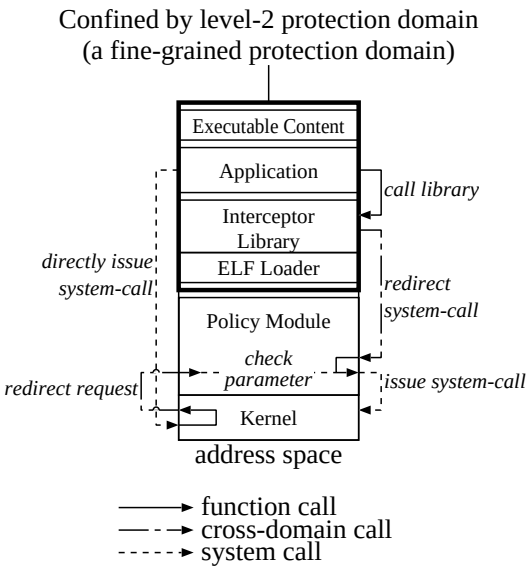

Figure 6. L2 protection domain incorporated in a document viewer

The plug-in then forwarded the request to the executable content via a cross-domain call. To draw the requested image, the executable content called the drawing APIs provided by the plug-in via cross-domain calls. The drawing APIs, implemented by the plug-in, performed access control. The plug-in strictly checked the parameter of every API call so that the content did not violate the protection policy. After ensuring that the API call was safe, the plug-in invoked the proper function of X Window to draw the image.

As the protection policy of the L1 protection domain, we defined a policy that follows the details described in Section 3.2. Memory access was confined only to the pages owned by the executable content. Resource accesses are allowed only through the APIs provided by the plug-in: directly using system calls were denied. These policy was enforced by the mechanism of fine-grained protection domains. In addition, we implemented the drawing APIs carefully so that it could ensure safety. In this way, we could incorporate an L1 protection domain into a web-browser.

Now we will show the implementation of an L2 protection domain. Since implementation of the L2 protection domain is the same as that used in the document viewer example, we show it in the next section.

4.2 A document viewer

The second example is a document viewer that handles script-type executable content used in an electronic document. Since the document viewer is an interpreter, as discussed in Section 2.3, it already performs some access control on executable content. We regard the access control mechanism of the viewer is that for an L1 protection domain. We threfore only show the technique to incorporate an L2 protection domain. This technique also use the fine-grained protection domain to implement the L2 protection domain.

We modified an ELF loader to incorporate an L2 protection domain without modifying the viewer. The loader assigns an L2 protection domain to the viewer just after loading it. It also loads a policy module that determines the protection policy of the L2 protection domain. For protection of the policy module itself, the loader creates and assigns another fine-grained protection domain to the policy module. In addition, it loads an *interceptor library* that intercepts system calls and forwards them to the policy module. This library allows the policy module to efficiently check system calls.

The policy module performs access control on resources by checking the system calls issued by the viewer. As shown in the right-side of Figure 6, a system call issued by the viewer is intercepted by the interceptor library and forwarded to the policy module. The module checks the parameters of the system call and determines whether the system call is allowed. As shown in the left-side of Figure 6, the system calls directly issued by the viewer are redirected to, and checked by, the policy module.

We implemented a sample policy module for the L2 protection domain. In the policy of the module, all accesses to memory except for that of the policy module were allowed. File accesses had to be read-only and write accesses to files were denied except for the temporary files specified in the policy. Network accesses were also denied. Defining a proper application-specific policy is a very difficult issue and out of the scope of this paper.

```
1  > ls
2  acroread shell.pdf
3  > ./acroread shell.pdf
4  $ ls
5  acroread shell.pdf
6  $
```

Figure 7. An example of an attack

```
1  > ls
2  acroread shell.pdf
3  > /lib/ld-pd.so -pm /lib/pm.so \
4  ? ./acroread shell.pdf
5  pm.so: exec: permission denied
6  pm.so: program terminated
7  >
```

Figure 8. An Example of protection

4.3 Demonstration

We demonstrate that a hierarchical protection model successfully suppresses the damage of attack from malicious executable content. We show an example of an attack against Adobe Acrobat Reader. As reported in SPS Advisory[4], Acrobat Reader 4.05 contains a flaw vulnerable to a buffer-overflow attack. An attacker can exploit the flaw to make the reader execute arbitrary code injected by the attacker. To demonstrate the attack, we made a PDF file exploiting the flaw. When the PDF file is loaded by the Acrobat Reader, a shell is executed. We used the code from the Aleph One's article[12] to execute a shell after causing the buffer-overflow.

Figure 7 shows an example of an attack against the reader. The PDF file named `shell.pdf` is processed by the reader at the third line, and at the fourth line a command interpreter is invoked unexpectedly. The command prompt changes from `'>'` to `'$'`, which means a `/bin/sh` is executed.

We now show an example that successfully suppresses the damage of the attack in Figure 8. We used the ELF loader described above to protect against the malicious PDF. At the third and fourth lines in Figure 8, the reader is invoked through the command `/lib/ld-pd.so`, which uses our modified ELF loader to invoke the reader. The command line option `-pm` specifies a policy file. In this case, at the fifth and sixth lines, the L2 protection domain detects unexpected behavior by the reader and immediately terminates it.

5 Performance

In this section, we show the experimental results that estimate the overhead of L1 and L2 protection domains. A low-overhead protection mechanism is important for practical use. Although nesting multiple protection domains is expected to incur high protection costs, we show that it is possible to achieve an efficient implementation of a hierarchical protection model. Since a primary cost factor is that for the cross-domain calls, we first show this cost. Then, we estimate the overhead of L1 and L2 protection domains respectively for the experiments.

We used a PC equipped with Intel Pentium III 1 GHz processor, 128 MB RAM, 75 GB hard-disk (IBM DTLA-307030), and Matrox Millennium G450 graphics card. The kernel was Linux 2.2.18 that we modified to implement fine-grained protection domains. The version of X Window was 4.0.1.

Table 1. Cycles for one procedure call

Method	Cycle	Time
cross-domain call	378	0.38μs
IPC (via pipe)	4,046	4.05μs

5.1 Cross-domain call

We show the measured processor cycles for a cross-domain call. We obtained the number of cycles from the time-stamp counter of the Pentium III. We repeatedly called the null procedure in a tight loop to reduce the effect of cache misses. For comparison, we measured the cycles of traditional IPC (via pipe). Table 1 shows the experimental results. A cross-domain call between fine-grained protection domains costs 378 cycles, or 0.38μs on a Pentium III 1 GHz machine, which is about 11 times faster than that for an IPC.

5.2 L1 protection domain

To estimate the cost of the L1 protection domain, we used the implementation described in Section 4.1. As an executable content, we implemented one that draws a Mandelbrot set image in the browser's window. We measured the time to draw various sizes of a square image of the Mandelbrot set. We changed the length of the square's side from 1 pixel to 150 pixels. Since the executable content calls the drawing APIs via cross-domain calls, it performs a number of cross-domain calls.

For comparison, we prepared three implementations of executable content: "No protection" used a normal function call to invoke the drawing APIs, "Process" used an IPC to invoke the API, and "JavaVM" was a Java applet. We used two versions of JavaVM 1.1.5 with a JIT compiler and 1.3 with HotSpot. We used Netscape 4.76 as the web-browser. Note that we did not use L2 protection domains because the purpose of this experiment was to estimate the cost of the L1 protection domain.

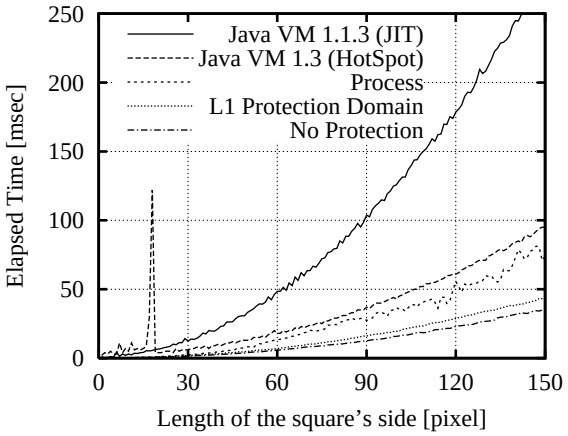

Figure 9. Overhead of L1 protection domain

Figure 9 shows the results. The horizontal axis denotes the length of the square's side in pixels, and the vertical axis denotes the elapsed time to draw the image in milliseconds. Since this executable content called the drawing APIs for every drawn pixel, the elapsed time increased proportional to the size of the square pixels. The average overhead of the L1 protection domain was about 22.6%; the executable content protected by the L1 protection domain cost 43ms to draw a square 150 pixels on a side, while "No Protection" type cost 35ms. As for other types, the average overhead of the "Process" type was 101% and JavaVM 1.1.5 was 755%. The overhead of JavaVM 1.3 is 199% on average for 31–150 pixels. Note that the peak around 20 pixels seems to due the JIT compilation.

5.3 L2 protection domain

To perform an experiment that estimates the cost of the L2 protection domain, we used two applications; Adobe Acrobat Reader and Ghostscript. Acrobat Reader is a PDF viewer and Ghostscript is a PostScript viewer. As discussed in Section 2.3, both PDF and PostScript can be regarded as executable content. We used the modified ELF loader described in Section 4.2 to assign an L2 protection domain to the viewers.

In the experiment for Ghostscript, we measured the time taken to interpret and display various sizes of PostScript files. Ghostscript processes files non-interactively. We used 163 PostScript files ranging from 280 Bytes to 4.55 MB. Figure 10 shows the result. The overhead was mostly kept below 2%. The average overhead was 1.05% and the maximum was 11.1%.

For the Acrobat Reader experiment, we measured the time taken to convert PDF files of various sizes to PostScript files. The reader processes the files non-interactively. We used 153 PDF files ranging from 758 Bytes to 8.13 MB for the experiment. Figure 11 shows the result. The overhead was kept below 8.3%. The average overhead was 1.88%.

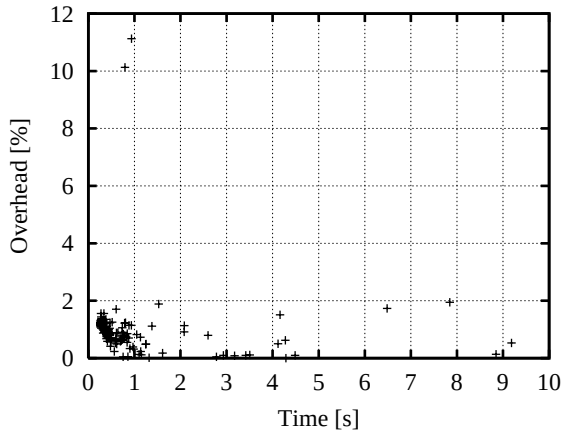

Figure 10. Overhead of L2 protection domain (Ghostscript)

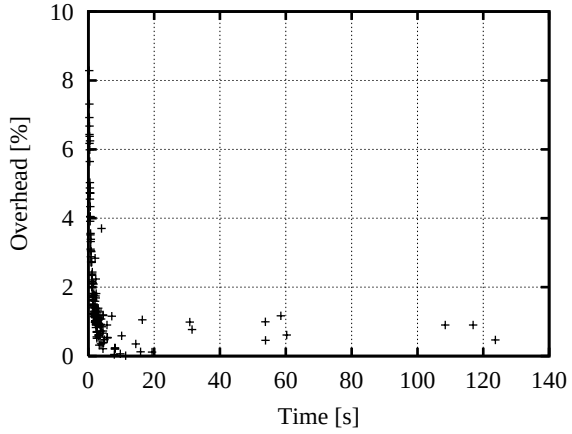

Figure 11. Overhead of L2 protection domain (Acrobat Reader)

These results show that the implementation achieved a low-overhead protection mechanism based on the hierarchical protection model.

6 Related Work

The Java applet is one of the most common executable content types. Recent Java has allowed us to control access from applets to resources with finer granularity. Java security [13] is ensured by the combination of various features such as type safety, byte code verification, and runtime checks by the Java virtual machine. Since all of these features are completely implemented in the virtual machine, a user's computer may be compromised if there is an implementation flaw in the virtual machine. Therefore, Java, in our terminology, provides only the L1 protection domain and lacks a fail-safe mechanism provided by the L2 protection domain.

Jaeger *et al.* [14] present a security architecture that enables system and application requirements to be enforced on applications composed of downloaded executable content. This architecture allows a more flexible and fine-grained access control than Java. However, it still corresponds to the L1 protection domain and does not address issues on robustness and granularity of protection.

Many sandboxing (or confining) systems have been developed to lessen the damage caused by a compromised process. In these systems, the process of the application is confined to restrict accesses to the resources. Janus [15] and MAPbox [16] use the system call tracing facility to check the system calls. SBox [17] uses a wrapper script to confine untrusted CGI scripts. Some of the others [18, 19] modify the kernel to confine untrusted applications or to provide virtual execution environments. Since the sandbox mechanism provides a fail-safe mechanism preventing unauthorized access by a compromised process, it corresponds to the L2 protection domain.

SubDomain [20] is an operating system extension designed to prevent vulnerabilities in Internet server platforms. Like other sandboxing systems, SubDomain provides a least privilege mechanism for programs and can be used to implement the L1 protection domain. The notable feature of SubDomain is that it enables arbitrary software components to be confined. Therefore it could be used to provide the L2 protection domain. However, SubDomain does not address the issues on robustness and granularity of protection.

Many alternatives to the L1 protection domain have been developed by operating systems researchers. Multics [21] uses the ring protection mechanism of a GE-645 processor to provide hierarchical protection *between multiple processes*, not in a single process. The model and implementation described in this paper address the hierarchical protection *inside a single process* and are totally different from Multics. Palladium [22] and PSL (Protected Shared Libraries) [23] use the memory management unit (MMU) to provide intra-memory protection. Palladium uses the ring protection mechanism of the Intel x86 family, and allows each process to have two protection domains in a single process. PSL exploits the memory management features of the Power processor, and enables multiple processes to share data safely in shared libraries.

Other techniques, not relying on hardware support, have also been proposed by operating system researchers. Software fault isolation (SFI) [24] provides intra-process memory protection. To confine all memory access in a fixed range of memory, SFI inserts a couple of instructions before every memory access instruction. Proof carrying code (PCC) [25] also guarantees safety properties such as memory safety. PCC attaches a formal proof to native code and the correctness of the proof is verified before execution in order to guarantee the safety properties.

Compared to our hierarchical protection model, all of these alternatives provide only the L1 protection domain. In addition, they primarily focus on intra-process memory protection and do not pay much attention to protecting other resources such as files.

Instead of providing fine-grained protection domains, many attempts [26, 27, 28, 29, 30] have been made in the context of micro-kernel research to make inter-process communication more efficient. These attempts focus on the interprocess communication issues and do not address the issues of granularity and robustness of protection mechanisms.

7 Summary

We introduced a hierarchical protection model to protect against malicious executable content. This model provides a fail-safe mechanism that tolerates the inevitable implementation flaws in the protection mechanisms. The design of nesting multiple protection domains can improve robustness without degrading the granularity of protection mechanism. Inner domains provide fine-grained protection and the outer protection domains cover flaws in the implementation of the inner protection domains to act as a fail-safe mechanism. This paper described the two-level hierarchy applied to this model as protection against executable content: an inner L1 protection domain and an outer L2 protection domain. We demonstrated an efficient implementation of the hierarchical protection model that exploited the fine-grained protection domains proposed in our previous papers. We showed that it can successfully suppress damage from attacks by executable content by using a real example of attacks against Adobe Acrobat Reader. Our experimental results showed that the implementation of our model achieved a low-overhead protection mechanism. The overhead of the L1 protection domain is estimated at about 22.6%. The overhead of the L2 protection domain is estimated at 1.05 – 1.88 % on average and a maximum of 8 – 12%.

References

[1] CERT Advisory, “CA-1995-10 Ghostscript Vulnerability,” Aug. 1995.

[2] CERT Advisory, “CA-1997-20 JavaScript Vulnerability,” July 1999.

[3] CERT Advisory, “CA-2000-07 Microsoft Office 2000 UA ActiveX Control Incorrectly Marked ”Safe for Scripting”,” May 2000.

[4] SPS Advisory, “#39: Adobe Acrobat Series PDF File Buffer Overflow,” July 2000.

[5] M. Lipow, “Number of faults per line of code,” *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 437–439, 1982.

[6] R. V. Binder, “Six sigma: Hardware si, software no!.” http://www.rbsc.com/pages/sixsig.html.

[7] T. Shinagawa, K. Kono, and T. Masuda, "Exploiting segmentation mechanism for protecting against malicious mobile code," Tech. Rep. 00-02, Department of Information Science, Faculty of Science, University of Tokyo, May 2000. An extended version of Shinagawa et. al [8].

[8] T. Shinagawa, K. Kono, M. Takahashi, and T. Masuda, "Kernel support of fine-grained protection domains for extention components," *Journal of Information Processing Society of Japan*, vol. 40, pp. 2596–2606, June 1999. In Japanese.

[9] M. Takahashi, K. Kono, and T. Masuda, "Efficient kernel support of fine-grained protection domains for mobile code," in *Proc. 19th IEEE International Conference on Distributed Computing Systems* (*ICDCS '99*), pp. 64–73, May 1999.

[10] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," in *DARPA Information Survivability Conference and Exposition*, pp. 1119–1129, Jan. 2000.

[11] L. Gong, "Java2 platform security architecture." http://java.sun.com/j2se/1.4/docs/guide/security/spec/security-spec.doc%.html.

[12] Aleph One, "Smashing the stack for fun and profit," 1996. http://www.shmoo.com/phrack/Phrack49/p49-14.

[13] L. Gong, *Inside Java2 Platform Security: Architecure, API Design, and Implementation*. Addison Wesley, 1999.

[14] T. Jaeger, A. Prakash, J. Liedtke, and N. Islam, "Flexible control of downloaded executable content," *ACM Transactions on Information and System Security (TISSEC)*, vol. 2, pp. 177–228, May 1999.

[15] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A secure environment for untrusted helper applications," in *Proc. 6th USENIX Security Symposium*, July 1996.

[16] A. Acharya and M. Raje, "MAPbox: Using parameterized behavior classes to confine untrusted applications," in *Proc. 9th USENIX Security Symposium*, Aug. 2000.

[17] L. D. Stein, "SBOX: Put CGI scripts in a box," in *Proc. 1999 USENIX Annual Technical Conference*, June 1999.

[18] M. Bernaschi, E. Gabrielli, and L. V. Mancini, "Operating system enhancements to prevent the misuse of system calls," in *Proc. 7th ACM Conference on Computer and Communications Security* (*CCS '00*) , pp. 174–183, Nov. 2000.

[19] P.-H. Kamp and R. N. M. Watson, "Jails: Confining the omnipotent root," in *Proc. 2nd International System Administration and Networking Conference (SANE)*, May 2000.

[20] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor, "SubDomain: Parsimonious server security," in *Proc. 14th Systems Administration Conference*, pp. 355–367, Dec. 2000.

[21] R. C. Daley and J. B. Dennis, "Virtual memory, processes, and sharing in Multics," *Communications of the ACM*, vol. 11, pp. 306–312, May 1968.

[22] T. Chiueh, G. Venkitachalam, and P. Pradhan, "Integrating segmentation and paging protection for safe, efficient and transparent software extensions," in *Proc. 17th ACM Symposium on Operating Systems Principles* (*SOSP '99*), pp. 140–153, Dec. 1999.

[23] A. Banerji, J. M. Tracey, and D. L. Cohn, "Protected Shared Libraries – A New Approach to Modularity and Sharing," in *Proc. 1997 USENIX Annual Technical Conference*, pp. 59–75, Oct. 1997.

[24] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proc. 14th ACM Symposium on Operating Systems Principles* (*SOSP '93*), pp. 203–216, Dec. 1993.

[25] G. C. Necula and P. Lee, "Safe kernel extensions without runtime checking," in *Proc. 2nd Symposium on Operating Systems Design and Implementation* (*OSDI '96*), pp. 229–243, Oct. 1996.

[26] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy, "User-level interprocess communication for shared memory multiprocessors," *ACM Transactions on Comuputer Systems (TOCS)*, vol. 9, pp. 175–198, May 1991.

[27] G. Hamilton, M. L. Powell, and J. G. Mitchell, "Subcontract: A flexible base for distributed programming," in *Proc. 14th ACM Symposium on Operating Systems Principles* (*SOSP '93*), pp. 69–79, Dec. 1993.

[28] B. Ford and J. Lepreau, "Evolving Mach 3.0 to a Migrating Thread Model," in *Proc. USENIX Winter 1994 Technical Conference*, Jan. 1994.

[29] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter, "The performance of μ-kernel-based systems," in *Proc. 16th ACM Symposium on Operating Systems Principles* (*SOSP '97*), pp. 66–77, Oct. 1997.

[30] J. Liedtke, K. Elphinstone, S. Schönberg, H. Härtig, G. Heiser, N. Islam, and T. Jaeger, "Achieved IPC performance," in *Proc. 6th Workshop on Hot Topics in Operating Systems* (*HOTOS '97*), pp. 28–31, May 1997.