Flexible and Efficient Sandboxing Based on Fine-Grained Protection Domains

Takahiro Shinagawa[1], Kenji Kono[2,3], and Takashi Masuda[2]

[1] Department of Information Science, Graduate School of Science, University of Tokyo
shina@is.s.u-tokyo.ac.jp
[2] Department of Computer Science, University of Electro-Communications
{kono, masuda}@cs.uec.ac.jp
[3] PRESTO, Japan Science and Technology Corporation

Abstract. Sandboxing is one of the most promising technologies for safely executing potentially malicious applications, and it is becoming an indispensable functionality of modern computer systems. Nevertheless, traditional operating systems provide no special support for sandboxing; a sandbox system is either built in the user level, or directly encoded in the kernel level. In the user-level implementation, sandbox systems are implemented by using support for debuggers, and the resulting systems are unacceptably slow. In the kernel-level implementation, users are obliged to use a specific sandbox system. However, users should be able to choose an appropriate sandbox system depending on target applications, because sandbox systems are usually designed for specific classes of applications. This paper presents a generic framework on top of which various sandbox systems can be implemented easily and efficiently. The presented framework has three advantages. First, users can selectively use the appropriate sandbox systems depending on the target applications. Second, the resulting sandbox systems are efficient enough and the performance is comparable to that of kernel-implemented sandbox systems. Finally, a wide range of sandbox systems can be implemented in the user level, thereby facilitating the introduction of new sandboxing systems in the user level. The presented framework is based on the mechanism of fine-grained protection domains that have been previously proposed by the authors.

1 Introduction

Sandboxing is one of the most promising technologies for safely executing applications. A sandbox system enables users to execute untrusted applications without concern, because it allows the users to define a security policy to be enforced on the applications. Even if a malicious one attempts to violate the security policy and to make unauthorized access, the sandbox system detects the access and prevents it. The facilities of sandboxing is indispensable for executing Internet applications in the today's Internet environment. Internet applications, which are applications that process data obtained from the Internet — such as mails, web pages, and electronic documents —, often contain security vulnerabilities to malicious attacks. Such vulnerable applications may be hijacked by crackers if the crackers manipulate the data on the Internet and exploit their vulnerabilities. For example, widely-used Internet applications such as Ghostscript and Adobe Acrobat Reader have pointed out their vulnerabilities to malicious attacks [1, 2].

Many research efforts have been devoted to developing sandbox systems [3–12], and some companies are shipping products based on sandbox technology. Many of these sandbox systems use the technique called *system call interception* to monitor the behavior of sandboxed applications. A *reference monitor* is a piece of program code interposed between a sandboxed application and the OS kernel to intercept system calls issued by the sandboxed application. To enforce a security policy, the reference monitor filters out undesired system calls that may violate the given security policy. This technique allows reference monitors to totally control resource access attempts made by sandboxed applications.

A sandbox system is implemented either in the kernel level or in the user level. In kernel-level approaches [3–7], a specific sandbox mechanism is directly encoded in the kernel. These approaches lack *flexibility* because the users are forced to use a specific sandbox mechanism already implemented in the kernel. A sandbox system should allow the users to choose an appropriate sandbox mechanism depending on the target applications because sandbox mechanism are typically designed for a specific class of applications. For example, SubDomain is designed for Internet servers and SBox is for CGI programs. In addition, it is risky to implement a new reference monitor in the kernel, because a trivial flaw in the implementation might expose the entire kernel to the danger of unauthorized access.

In user-level approaches [8–12], a reference monitor is placed in an external process to protect the reference monitor from sandboxed applications. To intercept system calls issued by the sandboxed applications, the reference monitor uses the operating system's (os)'s support for debuggers like `ptrace()` and `/proc` file systems. The primary advantage of the user-level approaches is their *flexibility*. Since a reference monitor is implemented in the user level, it is easy to implement various types of sandbox systems and we can choose an appropriate sandbox mechanism that is specific to a certain class of applications. The disadvantage of this approach is that it significantly degrades the performance of sandboxed applications. Whenever a sandboxed application issues a system call, a context switch is made to the reference monitor and the switching incurs terrible runtime overhead.

This paper presents a generic framework for flexible and lightweight sandbox systems. This framework enables various types of sandbox mechanisms to be built on top of the framework. The presented framework exploits *fine-grained protection domains* that have been proposed by the authors [13–15]. The framework combines the benefits of both the user-level and kernel-level approaches. In this framework, a reference monitor is implemented in the user level and thus, the flexibility of user-level approaches is retained. To retain the high performance of kernel-level approaches, a reference monitor is placed in the *same* process as the sandboxed application. By doing this, our framework eliminates the overheads involved in context switching. To prevent a sandboxed application from compromising the reference monitor, we use the functionality provided by fine-grained protection domains: a fine-grained protection domain protects a reference monitor from the sandboxed application in the same process, as if the reference monitor were outside the process of the sandboxed application.

The rest of the paper is organized as follows. Section 2 describes the goals of our sandboxing framework. Section 3 discusses the mechanism of fine-grained protection domains and Section 4 describes the implementation of the sandboxing framework. Section 5 shows experimental results obtained measuring the overhead of the sandboxing framework. Section 6 describes related work and Section 7 concludes the paper.

2 Goals of Sandboxing Framework

This section presents the goals of our sandboxing framework and shows approaches to meeting these goals. We have three goals: *flexibility*, *performance*, and *transparency*.

2.1 Flexibility

There are various design choices available in developing sandbox systems, such as passive or active, global or local, mandatory or discretionary, static or dynamic, generic or specific, and transient or persistent [16]. These design choices determine the expressive power of security policies that can be enforced by the resulting sandbox system. Therefore, it is difficult for a *single* sandbox system to enforce various classes of security policies that differ in their expressive power. To make various security policies enforceable, a generic sandboxing framework should be flexible enough to allow sandbox systems of various designs to be implemented on top of it. If a sandboxing framework allows only a specific sandbox design to be implemented, the enforceable security policies are limited, reducing the range of applicability of the sandboxing framework.

The primary goal of our sandboxing framework is flexibility: many types of sandbox systems must be able to be implemented on top of it. To ease the implementation of sandbox systems, our framework allows sandbox systems to be implemented in the user level. User-level sandbox systems are easier to implement, debug, and test than kernel-level systems. Furthermore, our framework enables each application to be sandboxed using different sandbox systems. This allows the users to choose a sandbox system suitable for their needs. Since the expressive power of security policies required for sandboxing depends on the user's demands, this property enhances the flexibility of our sandboxing framework.

2.2 Performance

The second goal of our sandboxing framework is performance. To control resource access attempts made by sandboxed applications, any sandbox system incurs additional overhead to some extent. If the additional overhead is too large, the resulting system is unacceptable for use. Unfortunately, user-level sandbox systems incur large overheads because they require a large number of context switches to intercept system calls issued by a sandboxed application. At least two context switches are required per system call since an external process intercepts the system call. A context switch involves large overhead incurred by scheduling and TLB flushing, therefore, it is difficult to implement lightweight sandbox systems based on the user-level approaches.

To reduce overhead of context switches, our framework places a reference monitor *inside* the sandboxed application's process. By doing this, the overhead involved in the system call interception can be eliminated because no context switch is involved, and the resulting sandbox system is a high-performance system. This approach, of course, poses a new security problem in that a reference monitor itself may be compromised easily because a sandboxed application can tamper with the reference monitor's code or/and data to bypass the reference monitor. To protect a reference monitor from the sandboxed application, we exploit fine-grained protection domains [13–15]. Fine-grained protection domains are a kernel-level protection mechanism that establishes *intra*-process protection. Using fine-grained protection domains, we can protect a reference monitor from the sandboxed application, as if the monitor resided in another process. In the next section, we discuss the mechanism of fine-grained protection domains.

2.3 Transparency

The third goal of our sandboxing framework is *transparency*. Transparency here means that an existing application can be sandboxed without having to modify it. As is in other sandbox systems, our framework uses the system call interception technique to accomplish transparency. Since access control is done by intercepting system calls, existing applications do not have to be modified.

3 Fine-grained Protection Domains

Before describing the implementation of our framework, we briefly summarize the mechanism of fine-grained protection domains. The fine-grained protection domain is a kernel-level extension, currently implemented as an extension to a Linux kernel, and greatly versatile in implementing protection mechanisms [13–15]. Here we introduce the functionalities of the mechanism that will be needed to explain the implementation of the sandboxing framework. The details of fine-grained protection domains have been previously reported [13–15].

3.1 Intra-process memory protection

Traditionally, the notion of the process coincides with that of the protection domain, therefore, *intra*-process memory protection is almost impossible. A fine-grained protection domain is finer than process-level protection domains, and multiple fine-grained protection domains can co-exist inside a single process. Figure 1 illustrates that two fine-grained protection domains, A and B, co-exist inside a process. Each fine-grained protection domain is associated with a code fragment, and determines to which memory pages the associated code can access. In Figure 1, domain A can access the entire memory of the process while domain B can access only memory pages $p1$, $p2$, and $p3$.

Memory protection is set up in the grain of memory pages. For each memory page, different fine-grained protection domains can have different access rights. In Figure 1 memory page $p0$ is accessible from domain A but not accessible from domain B, while memory page $p1$ is accessible from both domain A and domain B.

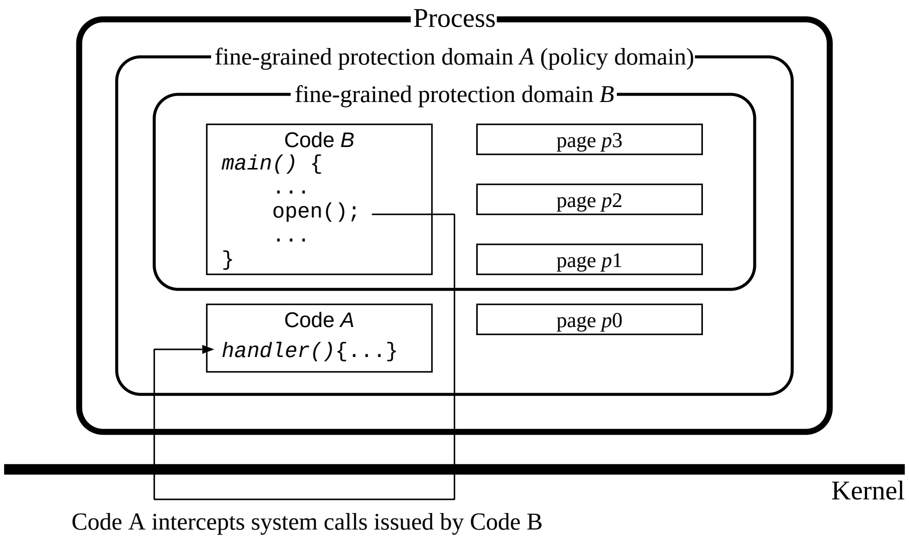
Process
fine-grained protection domain A (policy domain)
fine-grained protection domain B
Code B
main() {
...
open();
...
}
page p3
page p2
page p1
Code A
handler(){...}
page p0
Kernel
Code A intercepts system calls issued by Code B

Fig. 1. Fine-grained protection domains: two fine-grained protection domains A and B co-exist inside a process. The domain A and B are associated with Codes A and B respectively. Code B can not access Code A and a memory page $p0$. System calls issued by Code B are intercepted by a handler of Code A.

3.2 System call interception

For each process, there is exactly one fine-grained protection domain that controls the access rights of all fine-grained protection domains in the same process. This fine-grained protection domain is called a *policy domain* since it determines the policies of memory protection. In Figure 1, domain A is the policy domain. That is, domain A can change the access rights of domains A and B while domain B cannot.

In our implementation of fine-grained protection domains, a policy domain can intercept system calls issued by other fine-grained protection domains. To intercept system calls, a policy domain registers a system call handler *a priori* with the OS kernel. In Figure 1, domain A intercepts system calls issued by domain B. For example, if the code associated with domain B issues an `open` system call, the OS kernel upcalls the system call handler of domain A, passing the system call number and the system call arguments. Then, the handler examines the arguments. If the policy domain determines that executing the system call does not violate a security policy, the kernel executes it. Otherwise, the kernel returns an error code to domain B.

A policy domain usually has access rights to the entire memory of the process. Therefore, the policy domain can inspect the system call arguments if the arguments contain pointers to the memory of another domain. The policy domain can traverse the pointer structure without any pointer conversions and data copies.

3.3 Implementation overview

The fine-grained protection domain is implemented as an extension to the Linux kernel. The implementation centers around the *multi-protection* page table, a mechanism that accomplishes the intra-process memory protection. A multi-protection page table is an extension of the traditional page table that enables each memory page to have multiple protection modes at the same time. At any instance in time, one of the multiple protection modes is effective. Since each memory page can have multiple protection modes, each fine-grained protection domain can have its own protection mode for each memory page, thus, memory protection is provided between the fine-grained protection domains. To switch fine-grained protection domains, a special software trap is prepared in which the effective modes of memory protection are switched. By using the memory management unit (MMU) in sophisticated ways, we avoid flushing TLBs and caches that would degrade the performance of switching protection domains.

Our extended kernel can be ported to a wide range of processor architectures. The prototype system is running on Intel IA-32, SPARC, and Alpha processors. To implement the multi-protection page tables efficiently, we used the processor-specific features fully. For the Intel IA-32 family [17], we utilized the segmentation and the ring protection mechanisms. For the SPARC processors (versions 8 and 9) [18], the tagged TLBs and the register windows are used. For the Alpha processors [19], tagged TLBs and PAL code are utilized. Our extensions to the kernel are not large. Only 1,398 lines of code are added to the Linux 2.2.18 in the IA-32 implementation. The details of the implementation can be found in [13–15].

4 Implementation of Sandboxing Framework

Our sandboxing framework uses system call interception provided by the facilities of fine-grained protection domains. We prepare two fine-grained protection domains to sandbox an application. One protection domain is assigned to the application itself and the other domain is assigned to a reference monitor. The domain assigned to the reference monitor plays the role of the policy domain. Therefore, the reference monitor can intercept system calls issued by the sandboxed application and set up the memory protection policies.

The memory protection policies are as follows. To protect a reference monitor from the sandboxed application, the memory pages of the reference monitor are made *not* accessible from the fine-grained protection domain associated with the sandboxed application. On the other hand, the memory pages of the sandboxed application are made accessible from the fine-grained protection domain associated with the reference monitor. By doing this, the reference monitor can directly access a system call argument even if the argument contains a pointer to the data in the sandboxed application's domain.

In our sandboxing framework, fine-grained protection domains must be created and assigned to a sandboxed application before the application starts executing. We modified a program loader (an ELF loader in the case of Linux) to create and assign fine-grained protection domains. The program loader creates two fine-grained protection domains, loads a reference monitor and a sandboxed application to each domain, and sets up the memory protection policies. Then, the loader jumps to the sandboxed application to start execution.

Our approach makes our sandboxing framework flexible because it allows reference monitors to be implemented in the user level. To implement a reference monitor, the programmer simply prepares system call handlers that determine which system calls are allowed to be executed, based on a given security policy. Therefore, any sandbox systems that use the system call interception technique can be implemented on top of our sandboxing framework. Furthermore, since a reference monitor is associated with each application, the user can choose an appropriate sandbox system based on their needs.

5 Performance Experiments

This section presents the results of performance experiments. We measured the overheads of a sandbox system implemented on top of our sandboxing framework. The experimental results demonstrate that that our sandboxing framework is practical enough to implement lightweight sandbox systems on top of it.

The overheads of sandbox systems are important because sandbox systems would be impractical if their overheads are unacceptably high. To validate our approach, we have implemented a sample sandbox system on top of our framework. We call this sandbox system a *lightweight* sandbox. As a comparison, we also implemented a user-level sandbox system that uses debugging facilities to intercept system calls. We call this sandbox system a *process-based* sandbox since the reference monitor is implemented as an external process. In the experiments, we used a simple security policy designed to protect file systems. The reference monitors intercepted an `open()` system call, and the pathname passed as an argument is checked to prevent undesired file access. The reference monitor compares the pathname with 10 directory names that contain sensitive files such as `/etc/passwd`, and denies access to the files under those directory trees.

We conducted micro-benchmark and macro-benchmark tests. In the micro-benchmark tests, the overheads of system call interception are examined. In the macro-benchmark tests, the performances of real applications confined in sandbox systems are measured. We conducted the experiments on a PC having a 1GHz Pentium III processor, 512 MB of RAM, and a 75 GB hard-disk drive (IBM DTLA-307030). The kernel used was the Linux 2.2.18 that had been extended to support fine-grained protection domains.

5.1 Micro-benchmark

We measured the performance of system calls issued by a sandboxed application. Using the cycle counter of a Pentium III processor, the time required to complete one system call is measured. The system calls measured were `getpid()` and `open()`.

The measured performance of `getpid()` is the minimum overhead of system call interception. In the security policy used in the experiments, the reference monitors do nothing on this system call: they intercept `getpid()` but no checks are done. The

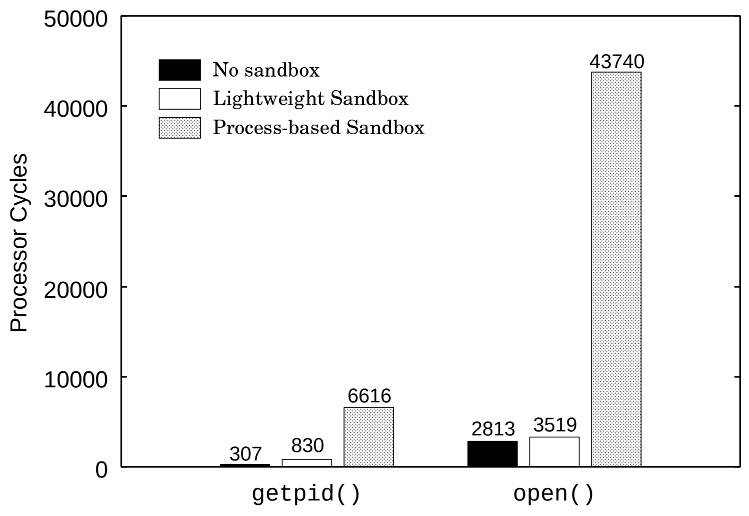

Fig. 2. Performance of system calls in sandbox systems. The lightweight sandbox, our approach, significantly outperforms the traditional process-based sandbox.

experimental results of `getpid()` are shown on the left-hand side of Figure 2. In Figure 2, *Lightweight Sandbox* is the sample sandbox system built on our sandboxing framework and *Process-based Sandbox* is the sandbox system that uses an external process for a reference monitor. The lightweight sandbox requires an additional cost of only 523 processor cycles, while the process-based sandbox adds 6,309 cycles. The lightweight sandbox costs one-twelfth less than the process-based sandbox.

Experimental results of `open()` are shown on the right-hand side of Figure 2. The results show that the lead is increased. The lightweight sandbox adds 706 cycles, while the process-based sandbox adds 40,927 cycles. The cost of the process-based sandbox is more than 57 times higher than the lightweight sandbox. This is because many interprocess communications are required to check the arguments of system calls. Since the arguments of `open()` are located in a process different from that of the reference monitor, the reference monitor needs inter-process communications to obtain memory data from the sandboxed application. Due to the limitation of the `ptrace()` system call, one interprocess communication is required to read four-byte data of the sandboxed process. Therefore, `open()` system call needs inter-process communications in proportion to the length of the pathname, and this results in large overheads. On the other hand, the lightweight sandbox does not incur any extra costs of inter-process communications, since the reference monitor is located in the same process with that of the sandboxed process. As a result, the advantage of the lightweight sandbox becomes larger.

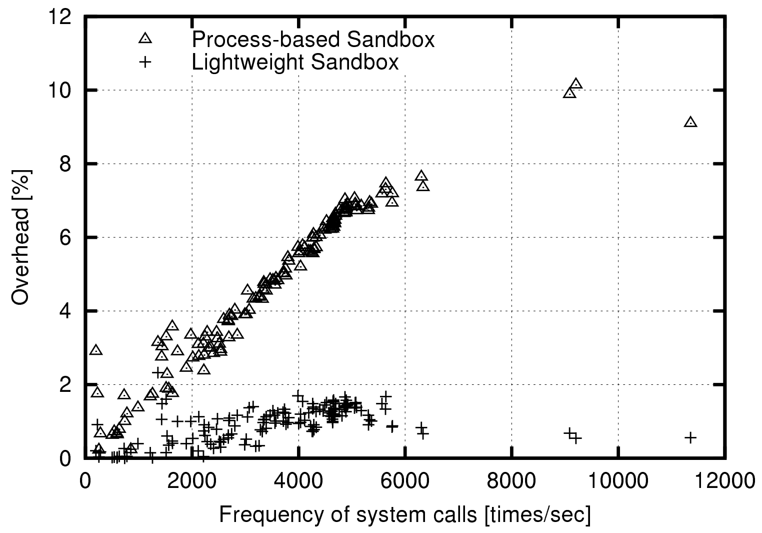

Fig. 3. Overhead of Ghostscript with sandbox mechanisms. The overhead is sorted by the frequency of system calls. The lightweight sandbox (our approach) outperforms the conventional process-based sandbox.

5.2 Macro-benchmark

The overhead of the sandbox systems was measured using real applications. Two applications, Ghostscript and Adobe Acrobat Reader, were used in these benchmark tests. Ghostscript is a Postscript document viewer and Acrobat Reader is a PDF document viewer. Each application was sandboxed and processed a lot of documents. For each document, the total execution time is measured. Experiments were performed in three cases for each application: no sandboxes, lightweight sandbox, and process-based sandbox. The overheads were calculated by the following equation.

$$overhead = \frac{T_{sandbox} - T_{nosandbox}}{T_{nosandbox}} \qquad (1)$$

$T_{sandbox}$ is the execution time with a sandbox system (the lightweight sandbox or process-based sandbox). $T_{nosandbox}$ is the time without any sandboxes. The calculated overhead is expected to be in proportion to the frequency of system calls, since the overhead is incurred by the cost of system call interception. Therefore, the experimental results were sorted by the frequency of system calls.

In the experiments using Ghostscript, Ghostscript processed 158 Postscript documents, the size of which ranged from 112 bytes to 175 Mbytes. Ghostscript was executed non-interactively and the outputs of Ghostscript were written to `/dev/null`. The version of Ghostscript used was 5.50.

Figure 3 shows the experimental results obtained for Ghostscript. The lightweight sandbox always outperformed the process-based sandbox. When the frequency of sys-

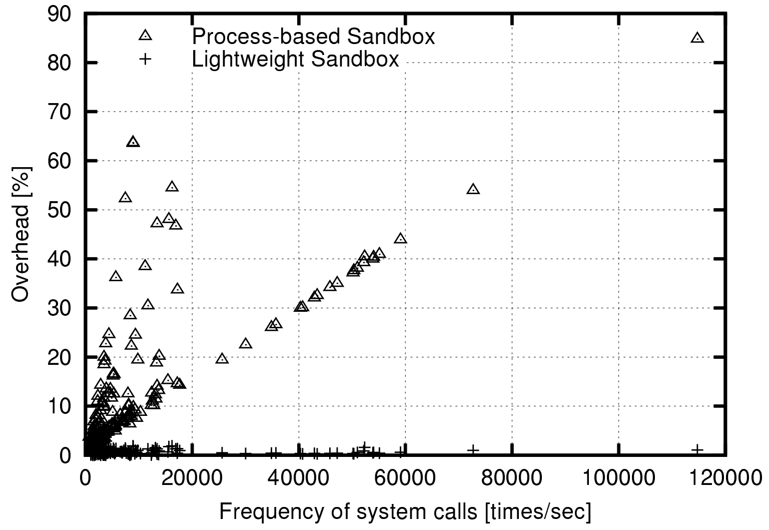

Fig. 4. Overhead of Acrobat Reader with sandbox mechanisms. The overhead is sorted by the frequency of system calls. The lightweight sandbox (our approach) outperforms the conventional process-based sandbox. Note that the scale of x-axis is different from that of the graph of Ghostscript.

tem calls was about 5,000 times per second, the overhead was about 1.5% with the lightweight sandbox, while about 7% with the process-based sandbox. Therefore, the lightweight sandbox was found to be approximately five times as fast as the process-based sandbox.

We performed the same experiments using Acrobat Reader. It processed 172 PDF documents of various sizes ranging from 1.6 Kbytes to 8.1 Mbytes. It converts each PDF document into a Postscript document. Acrobat Reader was executed non-interactively and the outputs were written to `/dev/null`. The version of Acrobat Reader used was 4.05.

Figure 4 shows the experimental results obtained for Acrobat Reader. With Acrobat Reader, the frequency of system calls tended to be higher than that with Ghostscript: the maximum frequency was about 11,300 times per second, which is ten times higher than that of Ghostscript. From the trace of system calls, we found that Acrobat Reader issued many `_llseek()` system calls. As a consequence, the overhead tends to be higher in Acrobat Reader than in Ghostscript. When the frequency of system calls is about 50,000 times per second, the overhead is about 1.6% with the lightweight sandbox, while about 37% with the process-based sandbox. Therefore, the lightweight sandbox is approximately 24 times as fast as the process-based sandbox. In addition, some PDF files incurred higher overheads than expected in comparison to other PDF files. This result was obtained probably because the performance of the process-based sandbox was affected by mysterious behavior of the scheduling, cache misses, and TLB flashes.

In summary, the lightweight sandbox, which is based on our sandboxing framework, outperforms conventional process-based sandboxes. In experiments using real applications, our approach was approximately 5–24 times as fast as traditional process-based approaches.

6 Related Work

A variety of sandboxing techniques have been previously developed. According to the location where reference monitors are interposed, these sandboxes are classified into three approaches: user-level approaches, kernel-level approaches, and application-embedded approaches.

The first approach is to implement reference monitors in the user-level. Several sandbox mechanisms, such as Janus [8], MAPbox [9], SBOX [10], Consh [11], and the system presented in [12] exploit this approach. By using the debugger support of system call tracing, sandboxed applications are monitored by external processes that implement reference monitors, thus performing access control of the sandboxed applications. This approach is advantageous because it is easy to adopt various types of access control mechanisms. On the other hand, this approach incurs substantial overheads because system call interception needs many interprocess communications.

The second approach is to implement reference monitors in the kernel level. Several sandbox systems have a reference monitor directly encoded in the kernel. SubDomain [4] provides a server-specific sandbox mechanism by way of system call interception. TRON [7] allows users to specify capabilities to files and directories, enforced by system call interception. An implementation of Domain and Type Enforcement [6, 20] provides a kind of mandatory access control. The `jail()` facility of FreeBSD [5] provides simple partitioning of the operating system environment. These kernel-level approaches, in which specific sandbox mechanisms are encoded in the kernel, limit the flexibility to support various security policies to meet the demands of the users.

To overcome the limitations of directly encoding specific sandbox mechanisms in the kernel, some systems employ loadable kernel modules [21, 22] to implement a specific sandbox system. LSM (Linux Security Module) [23] provides a general-purpose access control framework for Linux kernels that enables many different access control models to be implemented as loadable kernel modules. These approaches increase the flexibility of supporting various types of security policies. However, embedding reference monitors in the kernel creates significant risks. Implementations of reference monitors must be fully trusted, or the entire system becomes vulnerable. On the contrary, in our sandboxing framework, only the basic mechanism for system call interception is implemented in the kernel and the reference monitors are implemented at the user space. Therefore, implementation flaws in reference monitors do not cause serious system-wide vulnerabilities.

Peterson *et al.* [16] proposed a generic kernel-level framework for constructing sandboxes. It provides a minimal set of primitives that are designed to implement a wide variety of sandbox mechanisms. Although it provides a flexible sandboxing framework, the expressive power of enforceable security policies are still limited. In addition, it relies on external processes to handle the application-dependent aspects of sandboxing and thus, involves the overheads of interprocess communications.

The third approach is to embed sandboxes in sandboxed applications themselves. Java [24] implements sandboxes at the programming language level. Java ensures type safety at the language level and the Java virtual machine enforces security policies. These language-based approaches restrict the selection of programming languages for programmers. SFI [25] inserts processor instructions to perform memory confinement of sandboxed applications by directly modifying their binary executables. PCC [26] examines the binary code of a sandboxed application and ensures that it obeys predefined security policies. These instruction-level approaches cannot be applied to non-binary programs such as scripts.

Several research efforts are devoted to provide kernel-level protection domains within processes [27, 28], which are similar to our fine-grained protection domains. These kernel-level protection domains provide *memory* protection and are designed to support protection among software components or shared libraries. However, they do not address the issues of a generic sandboxing framework.

7 Conclusion

In this paper, we presented a generic and flexible sandboxing framework, on top of which various types of sandbox mechanisms can be implemented. This framework provides a lightweight method of system call interception, allowing user-level reference monitors to efficiently perform access control of sandboxed applications. Lightweight system call interception is accomplished by exploiting the kernel support of intra-process protection domains called fine-grained protection domains, which we have proposed in previous papers [13–15]. To apply sandbox systems without modifying existing applications, we developed a program loader that initializes and applies the sandbox mechanisms transparently to sandboxed applications. To demonstrate the feasibility of our approach, we implemented a sample sandbox mechanism on our framework. Our experimental results demonstrate that the overhead of our sandbox mechanism is reasonable. Our sample sandbox incurs only 1.5-1.6% of overheads when sandboxing Ghostscript and Adobe Acrobat Reader.

References

1. SPS Advisory #39: Adobe Acrobat Series PDF File Buffer Overflow, July 2000.
2. CERT Advisory CA-1995-10: Ghostscript Vulnerability, August 1995.
3. Massimo Bernaschi, Emanuele Gabrielli, and Luigi V. Mancini. Remus: A security-enhanced operating system. *ACM Transactions on Information and System Security (TISSEC)*, 5(1):36–61, 2002.
4. Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. SubDomain: Parsimonious server security. In *Proc. of the 14th Systems Administration Conference*, pages 355–367, December 2000.
5. Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proc. of the2nd International System Administration and Networking Conference (SANE)*, 2000.

6. Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. A domain and type enforcement UNIX prototype. In *Proc. of the 5th USENIX UNIX Security Symposium*, June 1995.
7. Andrew Berman, Virgil Bourassa, and Erik Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proc. of the USENIX Winter 1995 Technical Conference*, pages 165–175, January 1995.
8. Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure enviroment for untrusted helper applications. In *Proc. of the 6th USENIX Security Symposium*, July 1996.
9. Anurag Acharya and Mandar Raje. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proc. of the 9th USENIX Security Symposium*, August 2000.
10. Lincoln D. Stein. SBOX: Put CGI scripts in a box. In *Proc. of the 1999 USENIX Annual Technical Conference*, June 1999.
11. Albert Alexandrov, Paul Kmiec, and Klaus Schauser. Consh: Confined execution environment for internet computations. Available at http://www.cs.ucsb.edu/ berto/papers/99-usenix-consh.ps, 1998.
12. K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Proc. of the ISOC Network and Distributed Security Symposium* (*NSDD '00*), pages 19–34, 2000.
13. Masahiko Takahashi, Kenji Kono, and Takashi Masuda. Efficient kernel support of fine-grained protection domains for mobile code. In *Proc. of the 19th IEEE International Conference on Distributed Computing Systems* (*ICDCS '99*), pages 64–73, May 1999.
14. Takahiro Shinagawa, Kenji Kono, and Takashi Masuda. Exploiting segmentation mechanism for protecting against malicious mobile code. Technical Report 00-02, Department of Information Science, Faculty of Science, University of Tokyo, May 2000. An extended version of [15].
15. Takahiro Shinagawa, Kenji Kono, Masahiko Takahashi, and Takashi Masuda. Kernel support of fine-grained protection domains for extention components. *Journal of Information Processing Society of Japan*, 40(6):2596–2606, June 1999. in japanese.
16. David S. Peterson, Matt Bishop, and Raju Pandey. A flexible containment mechanism for executing untrusted code. In *Proc. of the 11th USENIX Security Symposium*, pages 207–225, August 2002.
17. *The Intel Architecture Software Developer's Manual, Volume 3: System Programing Guide*. Intel, 1999. Order Number 243192.
18. Menlo Park and SPARC International. *The SPARC Architecture Manual Version 8*. Prentice Hall, 1992. ISBN 0-13-825001-4.
19. Richard L. Sites and Richard T. Witek. *Alpha AXP Architecture Reference Manual*. Digital Press, 1995. ISBN 1-55558-145-5.
20. Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Shermann, and Karen A. Oostendorp. Confining root programs with domain and type enforcement (DTE). In *Proc. of the 6th USENIX Security Symposium*, July 1996.
21. Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS software with generic software wrappers. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 2–16, 1999.
22. Terrence Mitchem, Raymond Lu, and Richard O'Brien. Using kernel hypervisors to secure applications. In *Proc. of the 13th Annual Computer Security Applications Conference* (*AC-SAC '97*), pages 175–182, December 1997.
23. Chris Wright, Crispin Cowan, James Morris, Stephen Smalley, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proc. of the 11th USENIX Security Symposium*, August 2002.
24. Java Team, James Gosling, Bill Joy, and Guy Steele. *The Java[tm] Language Specification*. Addison Wesley Longman, 1996. ISBN 0-201-6345-1.

25. 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 Systems Principles* (*SOSP '93*), pages 203–216, December 1993.
26. George C. Necula and Peter Lee. Safe kernel extensions without runtime checking. In *Proc. of the 2nd Symposium on Operating Systems Design and Implementation* (*OSDI '96*), pages 229–243, October 1996.
27. 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 Systems Principles* (*SOSP '99*), pages 140–153, December 1999.
28. Arindam Banerji, John Michael Tracey, and David L. Cohn. Protected Shared Libraries – A New Approach to Modularity and Sharing. In *Proc. of the 1997 USENIX Annual Technical Conference*, pages 59–75, October 1997.