

A Low-Cost Runtime-Privilege Changing System for Shared Servers

Daisuke Hara*, Yasuichi Nakayama*

**Department of Computer Science, The University of Electro-Communications, Chofu, Tokyo 182-8585 Japan*

hara-d@igo.cs.uec.ac.jp, yasu@cs.uec.ac.jp

Abstract— We propose a low-cost runtime-privilege changing system that solves security problems in shared servers. The main problem with a shared server operating under conventional access control, i.e., an owner/group/other in combination with a Web server that runs under the privilege of the same user is that malicious users potentially can steal, delete, or tamper with other user's files. Existing approaches solve a portion of this problem, but they either lack performance, site-number scalability, or generality. POSIX ACL and a secure OS do not ensure security by themselves. Containers and virtual machines (VMs) have low scalability and low generality because they have the overhead of virtualization and because they typically require modifying the kernel. We implemented our system for an Apache on a Linux OS and evaluated its effectiveness. Our experimental results show that the throughput with it was, on average, 0.5% lower than that with Apache and was a maximum of 4.7% lower. Our system should be used for practical Web servers because its overhead is very low.

Keywords— Security in a Server, Shared Hosting Service, Web Server System, Runtime Privilege, Site-number Scalability

I. INTRODUCTION

More people are creating their own content and publishing it on the Web as the Internet grows in popularity. Although various types of Web services are available for creating Web content, many powerful Web publishers tend to use shared hosting services. Providers of these services typically lease server resources for a monthly/yearly fee for use in building Web sites. Customers login to an assigned server with a given account and install favorite weblogs, wikis [1], and content management systems (CMSs) [2], among other things. A customer can thereby publish its content more flexibly and powerfully than Web publishers that use Web services to create Web content.

However, malicious users potentially can steal, delete, or tamper with other user's files in a shared server operating under conventional access control, i.e., an owner/group/other in combination with a Web server that runs under the privilege of the same user.

We describe a low-cost runtime-privilege changing system that solves these security problems in a shared server. In our system, (1) the effective user ID/group ID of server

processes are changed to an ordinary user/group by using `seteuid()` and `setegid()` system calls when receiving a request. It has high performance because it does not need a process termination after each HTTP session by using `seteuid()` and `setegid()` system calls. (2) Our system can only invoke a series of `setuid()` and `setgid()` system calls to change the runtime privilege; i.e., user scripts cannot invoke these system calls.

We implemented our system for an Apache on a Linux OS and evaluated its effectiveness. Our experimental results show that the throughput with it was, on average, 0.5% lower than that with Apache and was a maximum of 4.7% lower. It should be used for practical Web servers because its overhead is very low.

The remainder of this paper is structured as follows. In section II, we describe existing approaches and their limitations. In section III, we describe the key aspects of our design. In section IV, we describe the implementation of our system on a Linux OS. In section V, we describe our evaluation of the system. Finally, in section VI, we summarize the key points of this work and discuss future work.

II. EXISTING APPROACHES AND THEIR LIMITATIONS

In this section, we describe runtime privileges in UNIX-like OSes, existing approaches to the security problems in a shared server, and their limitations.

A. Runtime Privileges in UNIX-like OSes

Existing UNIX-like OSes have system calls that change a process' runtime privilege. One series of `setuid()`¹ changes the real user ID/effective user ID, and one series of `setgid()`² changes the real group ID/effective group ID. Once a process that runs under the privilege of a root user changes its runtime privilege to an ordinary user by invoking `setuid()` and `setgid()` system calls, it cannot change its runtime privilege back to the root user. These system calls change the real and effective user ID/group ID; i.e., they are noninvertible. However, `seteuid()` and `setegid()` system calls are invertible because they change only the effective user ID/group ID.

¹A series of `setuid()` include `setuid()`, `seteuid()`, `setreuid()`, and `setresuid()` system calls.

²A series of `setgid()` include `setgid()`, `setegid()`, `setregid()`, and `setresgid()` system calls.

Processes that run under the privilege of different users in UNIX-like OSes, such as shell processes, are typically created as a root privilege. They change their real and effective user ID/group ID to an ordinary user ID/group ID before operation by invoking `setuid()/setgid()` system calls. In contrast, Web server processes such as Apache [3] conventionally run under the privilege of a dedicated user ID³. If users that share the same server machine have Web sites, each user uploads his or her content files⁴ under a home directory. Read, write, and execution permission on these content files must be granted to an other, which is defined by the UNIX permission model owner/group/other so that server processes that run under the privilege of the dedicated user ID can read, write, and execute them⁵ (Figure 1. (0)). These permission settings enable content files to be published (Figure 1. (1-3)).

The problem in this situation is that the files can be illegally stolen, deleted, or tampered with by malicious users that share the server by using command-line tools, such as `cp` and `rm` (Figure 1. (i-1)). They can also attack through the Web server (Figure 1. (i-2)). For example, a malicious CGI script or server-embedded script [4][5][6][7] that deletes an other user's writable files can run because the script runs under the privilege of the dedicated user, which can write the file.

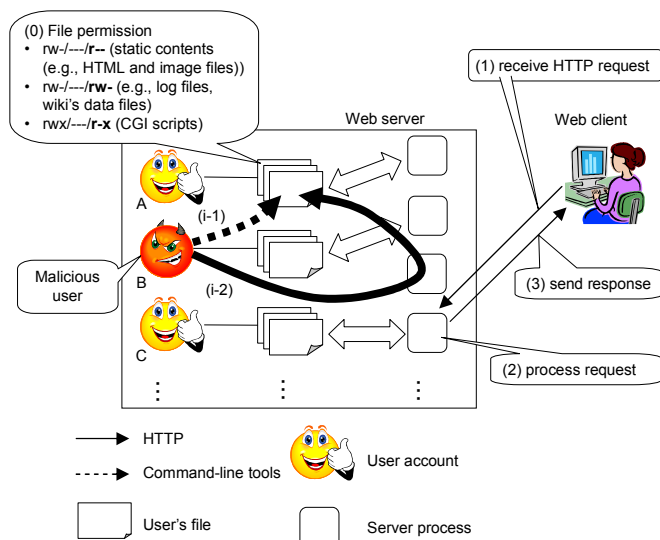


Figure 1. Problem with server processes running under the same privilege

B. POSIX ACL

POSIX ACL [8] provides access control for each user, unlike conventional access control, i.e., owner/group/other. If read and execution permission for content files is granted only to a dedicated user by using POSIX ACL, the files can

be published without granting permission to an other. It can therefore prevent stealing, deletion, or tampering with the files by executing `cp` and `rm` commands (Figure 1. (i-1)).

In contrast, it cannot prevent attacks through the Web server (Figure 1. (i-2)) because they usually run under the privilege of the dedicated user. However, it can prevent attacks with only CGI scripts because they run under the privilege of the site owner by using `suEXEC` [3][9][10].

`suEXEC` cannot achieve the speed of server-embedded interpreters because it needs two process terminations after each request (Figure 2. (1)). In addition, because server-embedded scripts that are executed by using server-embedded interpreters [4][5][6][7] run under the privilege of the dedicated user, they cannot ensure security in a shared server. Therefore, `suEXEC` is applied only to a CGI.

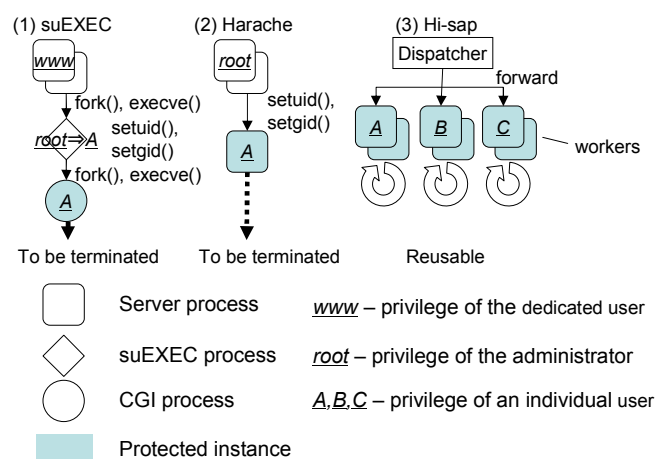


Figure 2. Processes of Web server systems

C. Secure OS

Secure OSes [11][12][13] enhance security features, e.g., mandatory access control (MAC) [14] and least privilege [15] security. The MAC mechanism enforces access control for all users and processes without exception. In the least privilege security model, a higher-than-needed privilege level is not granted to users and processes.

In conventional UNIX-like OSes, a root privilege can operate an entire OS. However, secure OSes can restrict the operations by a root user. If the root privilege is appropriated due to a security hole or misconfiguration, the access control of the secure OS suffers little effect.

Although it can prevent stealing, deletion, or tampering with content files by executing `cp` and `rm` commands (Figure 1. (i-1)), it cannot prevent attacks through the Web server by itself (Figure 1. (i-2)).

D. Container and Virtual Machine

Containers [16][17][18][19][20][21][22] are OS-level virtualization methods. Multiple containers with server software programs can run concurrently in an OS (Figure 3. (1)). Each container has different namespaces. Assigning a container to every site creates high security in the server. However, using containers at shared hosting services is

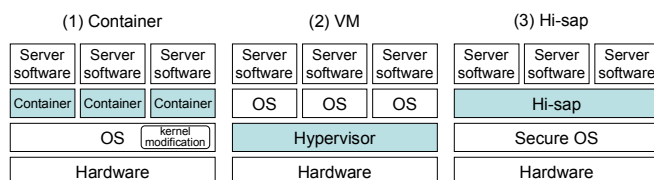
³e.g., *apache*, *www-data*, *www*

⁴Their owner is each user.

⁵To publish static files, such as HTML and image files, read permission must be granted to an other. To publish wikis or weblogs, write permission must also be granted to write their data files. To publish CGI scripts, execution permission must also be granted.

difficult because of their scalability for the number of sites in a server. Although this mechanism can scale up to a few hundred sites, service providers require scaling of up to about 1000. In addition, some containers, for example Linux-VServer [19], need to modify the kernel. Kernel modifications are dependent on the kernel version, so keeping them up to date generally requires significant porting [22]. If the porting is not done, the kernel's latest features and devices cannot be used.

In VMs [23][24][25], a hypervisor can run multiple OSes concurrently on the same server machine (Figure 3. (2)). Assigning an OS to every site also creates high security in the server. However, using VMs at shared hosting services is difficult because of the overhead involved. The utilization of computation resources for each site dramatically increases when this mechanism is used. This strongly affects the scalability of the number of sites in a server. For example, an OS that runs server programs on VMware ESX Server reportedly uses about 200 MB of memory [24]. That means about 200 GB of memory is required to provide 1000 sites. In addition, paravirtualization [25] needs to modify the kernel, and it has low generality.



- (1) Some containers need to modify the kernel.
 (2) A paravirtualization needs to modify the kernel.
 (3) Hi-sap does not need to modify the kernel, and it can use any secure OS.

Figure 3. Comparison of container, VM, and Hi-sap

E. Harache and Hi-sap

We proposed two Web server systems, Harache [26][27] and Hi-sap [28], which solve the security problems in a shared server (Figure 1. (i-1, i-2)).

Harache enables safe and convenient use of server-embedded programs such as server-embedded interpreters and WebDAV [29]. Each process of a Web server runs under the privilege of an individual user for every site. Therefore, permission is granted to only an owner for any content that includes server-embedded scripts. Although Harache has up to 1.7 times the performance of suEXEC, it cannot achieve the speed of server-embedded interpreters because it needs a process termination after each HTTP session (Figure 2. (2)).

Hi-sap speeds up server-embedded interpreters. The privilege of processes is changed in advance to avoid performance degradation (Figure 2. (3)), unlike suEXEC and Harache. A “dispatcher” distributes requests to “workers” that run under the privilege of an individual user. In Hi-sap, server software programs share a single namespace in an OS (Figure 3. (3)). Because it dynamically controls the number of Web servers in proportion to the volume of access traffic, it achieves high scalability of 1000 sites per server. It has up to 14.3 times the throughput of suEXEC. Because the throughput with it is, on average, 2.0% lower than that with

Apache and is a maximum of 6.9% lower, Hi-sap's overhead is very low. However, its latencies are, on average, 24% higher than those with Apache and is a maximum of 55% higher.

III. DESIGN

We designed our low-cost runtime-privilege changing system that can be used with UNIX-like OSes. Our goal is that the system ensures the security in a shared server with little performance degradation. We aim at high throughput and low latency.

A. Change in Runtime Privilege

In our system, server processes are invoked under the privilege of a root user. When a request is received (Figure 4. (1)), the server process changes its runtime privilege (effective user ID/group ID) to an ordinary user/group by using `setuid()/setgid()` system calls (Figure 4. (2)). Then, it processes the request (Figure 4. (3)) and sends the response (Figure 4. (4)). After that, it changes its runtime privilege back to 0 (root) (Figure 4. (5)). File permissions are granted to only an owner for any content (Figure 4. (0)).

If our system is applied to a Web server, the server process changes its runtime privilege to a site owner. Since `setuid()/setgid()` system calls are used, a process termination after each HTTP session like Harache is not required; i.e., a server process can process many requests. Therefore, our system should have high throughput and low latency.

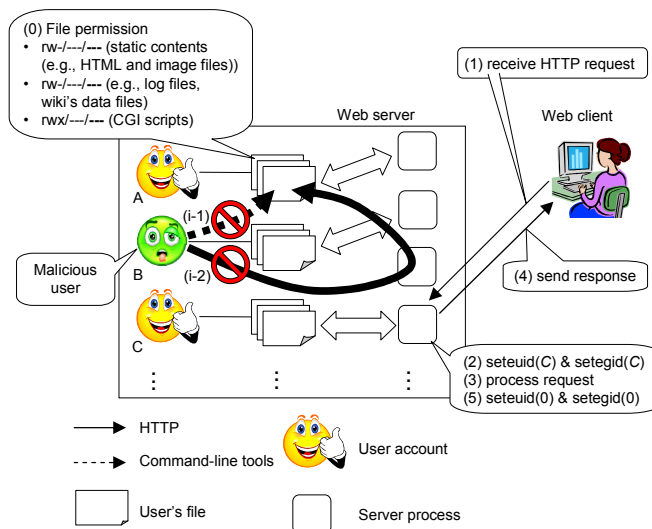
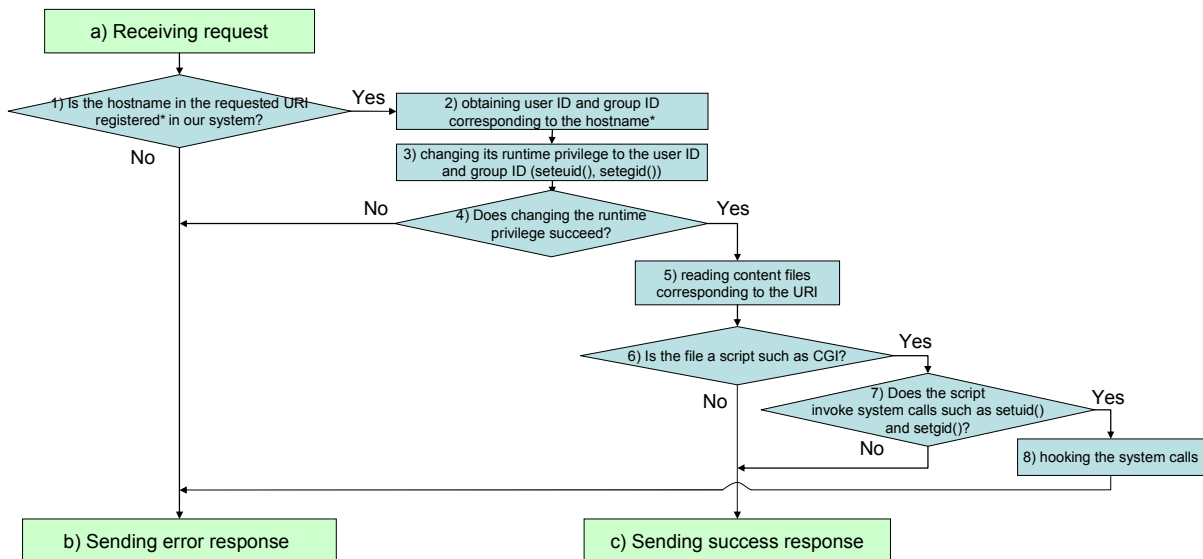


Figure 4. Overview of our system

B. Limitation with Changing Runtime Privilege by User Scripts

A Web server has user scripts available such as a CGI. Because these scripts usually can invoke `setuid()/setgid()` system calls as well as our system can, malicious users potentially can appropriate a root privilege.



(*) Our system has a relationship between the hostname and user ID/group ID.

Figure 5. Flow chart of our design

To prevent it, our system hooks calls for a series of `setuid()` and `setgid()` system calls⁶ and disables them. Therefore, our system can only change the runtime privilege. A flow chart of our design is shown in Figure 5. It is an example for a Web server. The concept of our design can be applied to other server programs that provide the service to many users.

IV. IMPLEMENTATION

We implemented our system for an Apache HTTP server 2.2.10 [3] on a Linux OS. The function for changing the runtime privilege was implemented as a module, *mod_setuid.so*, on an Apache. The function that limits user scripts when their runtime privilege is changed was implemented as a shared object, *setuid_hooks.so*, outside of an Apache.

Our system has a simple and user-level implementation and does not need to modify the kernel. Therefore, our system can be easily ported to any UNIX-like OSes.

The details of an Apache installation, *mod_setuid.so*, and *setuid_hooks.so* are as follows.

A. Apache installation

An Apache that is the base of our system is compiled with the following options.

- Server processes (*httpd*) run under the privilege of the root user.
- CGIs are enabled⁷.
- Dynamic Shared Objects (DSOs) are enabled⁷.

The *mod_setuid.so* is dynamically installed on the Apache.

⁶ Our system also hooks calls for a `setuid()`, `setgid()`, `setreuid()`, `setregid()`, `setresuid()`, `setresgid()`, `setfsuid()`, and `setfsgid()` system calls.

⁷ The default installation configuration of an Apache enables CGIs and DSOs.

B. *mod_setuid.so*

If our system receives a request from a Web client (Figure 5. (a)), it checks whether or not a hostname in requested URI is registered in our configuration file (*httpd.conf*) (Figure 5. (1)). If our system has registered the URI, it obtains the user ID and group ID corresponding to the hostname from the configuration file (Figure 5. (2)). It then changes its effective user ID/effective group ID to the user ID/group ID by invoking `setuid()` and `setgid()` system calls (Figure 5. (3)). If the effective user ID of the server process is not 0 (root) when changing the runtime privilege, it changes its runtime privilege back to 0 in advance. After changing the runtime privilege to the user ID and group ID, it reads content files corresponding to the URI (Figure 5. (5)) and sends success response to the Web client (Figure 5. (c)).

If our system has not registered a hostname in the requested URI or has an error when changing the runtime privilege (Figure 5. (4)), it sends an error response to the Web client (Figure 5. (b)).

C. *setuid_hooks.so*

As described in section III-B, the shared object named *setuid_hooks.so* hooks calls for a series of `setuid()` and `setgid()` system calls (Figure 5. (6–8)) to prevent the scripts in the content files from appropriating a root privilege. To activate this hooking, the path to the *setuid_hooks.so* was added to */etc/ld.so.preload* before starting the Apache.

If our system hooks calls for a series of `setuid()` and `setgid()` system calls from user scripts, it returns an error to the user script and sends an error response to the Web client (Figure 5. (b)).

V. EVALUATION

We evaluated our system using the hardware configuration listed in Table 1.

Table 1. Hardware configuration of experimental environment

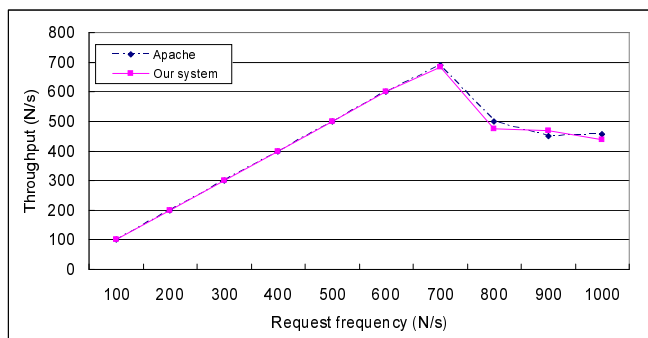
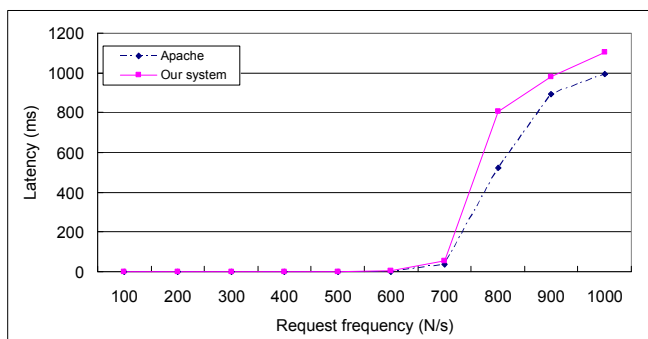
Client & Server	
CPU	AMD Opteron 240EE 1.4 GHz x 2
Memory	4 GB
OS	CentOS 5.3 (Linux 2.6.18)
NIC	Broadcom BCM5704C 1 Gbps

A. Basic Performance

We evaluated the basic performance of our system when processing dynamic content to determine its effectiveness. An Apache HTTP server 2.2.10 (Apache) was used for comparison. In our system and Apache, a PHP script was executed using the server-embedded interpreter. Our system and Apache used the default configuration files. We used httpperf benchmark 0.9.0 [30] to measure the performance.

We sent requests to the PHP script and measured the response throughput. The script calls *phpinfo()*, which displays the system information of the PHP language processor. The traffic generated by the script is 40 KB per request. As shown in Figure 6, the throughput with our system was, on average, 0.5% lower than that with Apache and was a maximum of 4.7% lower. As shown in Figure 7, the latency with our system was, on average, 31.6% higher than that with Apache and was a maximum of 59.9% higher. These were due to the overhead of *mod_setuid.so* and the hook operation for a series of *setuid()* and *setgid()* system calls. However, the throughput degradation is low. Because the maximum latency with our system was 1.1 seconds, it should be used for practical Web servers.

Therefore, this implementation is effective.

**Figure 6. Basic performance evaluation: throughput****Figure 7. Basic performance evaluation: latency**

B. Security

Server processes, in particular Web server processes, running under the privilege of the administrator (root user) account used to be considered unsafe. Although server programs, such as Samba [31] and mail servers, that are invoked under the privilege of a root user and that change the effective user ID/group ID to an ordinary user/group are available, a root user can do everything in a conventional OS. In particular, security cannot be ensured while scripts of an ordinary user run if the real user ID/group ID are 0 (root). These scripts can invoke a *setuid(0)* or *seteuid(0)* system call to appropriate a root privilege.

However, our system hooks calls for a series of *setuid()* and *setgid()* system calls to prevent the scripts from appropriating a root privilege. In addition, a root privilege can be restricted by using the secure OSes described in section II-C. If a server process is appropriated due to a security hole or misconfiguration, it can limit the scope of the effect. Therefore, our system can provide a safer server environment in combination with a secure OS.

VI. CONCLUSIONS

This paper has three contributions. First, we have clarified the security problems in a shared server. Second, we have clarified runtime privileges in UNIX-like OSes, existing approaches to the security problems, and their limitations. Finally, we have described our design of a low-cost runtime-privilege changing system and our implementation of it for a Web server on a Linux OS. Our evaluation results demonstrate that our system solves the security problems in a shared server with little performance degradation.

We plan to apply a SELinux feature to our system and evaluate it with real applications. In addition, we will apply the concept of our design to other server programs that provide service to many users.

REFERENCES

- [1] WikiWikiWeb. <http://c2.com/cgi/wiki?WikiWikiWeb>
- [2] S. Goodwin and R. Vidgen. Content, content, everywhere...time to stop and think? The process of Web content management, *IEEE Computing & Control Engineering Journal*, Vol. 13, No. 2, pp. 66–70, 2002.
- [3] Apache HTTP Server. <http://httpd.apache.org/>
- [4] PHP: Hypertext Preprocessor. <http://www.php.net/>
- [5] *mod_ruby*. <http://modruby.net/>
- [6] *mod_perl*. <http://perl.apache.org/>
- [7] *mod_python*. <http://www.modpython.org/>
- [8] A. Grunbacher. POSIX Access Control Lists on Linux, *Proc. FREENIX Track: 2003 USENIX Annual Technical Conference*, pp. 259–272, 2003.
- [9] N. Neulinger. CGIWrap: User CGI Access. <http://cgiwrap.sourceforge.net/>
- [10] S. Marsching. suPHP. <http://www.suphp.org/>
- [11] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System, *Proc. FREENIX Track: 2001 USENIX Annual Technical Conference*, pp. 29–40, 2001.
- [12] LIDS. <http://www.lids.org/>
- [13] T. Harada, T. Horie, and K. Tanaka. Task Oriented Management Obviates Your Onus on Linux, *Proc. Linux Conference 2004*, 2004.
- [14] J. McLean. The algebra of security, *Proc. 1988 IEEE Symposium on Security and Privacy*, pp. 2–7, 1988.
- [15] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. the IEEE*, Vol. 63, No. 9, pp. 1278–1308, 1975.

- [16] P.-H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root, *Proc. the 2nd International System Administration and Networking Conference*, 2000.
- [17] Parallels Virtuozzo Containers.
<http://www.parallels.com/products/virtuozzo/>
- [18] J. Dike. A user-mode port of the linux kernel, *Proc. the USENIX Annual Linux Showcase and Conference*, 2000.
- [19] Linux-VServer. <http://linux-vserver.org/>
- [20] Linux containers. <http://lxc.sourceforge.net/>
- [21] S. Soltesz, H. Potzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors, *ACM SIGOPS Operating Systems Review*, Vol. 41, No. 3, pp. 275–287, 2007.
- [22] P. Suranyi, H. Abe, T. Hirotsu, Y. Shinjo, and K. Kato. General Virtual Hosting via Lightweight User-level Virtualization, *Proc. the 2005 International Symposium on Applications and the Internet (SAINT 2005)*, pp. 229–236, 2005.
- [23] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization, *Proc. the ACM Symposium on Operating Systems Principles (SOSP 2003)*, pp. 164–177, 2003.
- [24] C. A. Waldspurger. Memory Resource Management in VMware ESX Server, *Proc. the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pp. 181–194, 2002.
- [25] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications, Technical Report 02-02-01, University of Washington, 2002.
- [26] D. Hara, R. Ozaki, K. Hyoudou, and Y. Nakayama. Design and Implementation of A Web Server for A Hosting Service, *Proc. the 9th IASTED International Conference on Internet and Multimedia Systems and Applications*, pp. 69–74, 2005.
- [27] D. Hara, R. Ozaki, K. Hyoudou, and Y. Nakayama. Harache: A WWW Server Running with the Authority of the File Owner, *Journal of Information Processing Society of Japan*, Vol. 46, No. 12, pp. 3127–3137, 2005 (in Japanese).
- [28] D. Hara and Y. Nakayama. Secure and High-performance Web Server System for Shared Hosting Service, *Proc. the 12th International Conference on Parallel and Distributed Systems (ICPADS 2006)*, pp. 161–168, 2006.
- [29] Y. Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. HTTP Extensions for Distributed Authoring – WEBDAV, *RFC 2518*, 1999.
- [30] D. Mosberger and T. Jin. httpperf—A Tool for Measuring Web Server Performance, *Proc. 1st Workshop on Internet Server Performance*, pp. 59–67, 1998.
- [31] Samba. <http://www.samba.org/>