A Study of a Dynamic Safety Net for Server Software

サーバソフトウェアのための動的なセーフティネットの研究

By

Kenichi Kourai

光來 健一

March 2002

A Dissertation Submitted to
Department of Information Science
Graduate School of Science
The University of Tokyo

In Partial Fulfillment of the Requirements
for the Degree of Doctor of Science.

Kenichi Kourai
A Study of a Dynamic Safety Net for Server Software
Ph.D Dissertation, Graduate School of Science, The University of Tokyo
March 2002

A Study of a Dynamic Safety Net for Server Software

Kenichi Kourai

Department of Information Science
The University of Tokyo
kourai@is.s.u-tokyo.ac.jp

Abstract

As server software is getting bigger and more complex, two major problems that server software involves in terms of safety come to the surface. One is an attack against server software and the other is its instability. Most of these problems are caused by software flaws. Since finding all software flaws is difficult, unknown flaws are left in a lot of server software. Therefore, there are serious demands on the facility that minimizes damages in case that server software becomes insane. We call this facility *safety net*. Since server software must handle requests from various users simultaneously and performance is important as well as safety, it needs to dynamically change the range of a safety net depending on the situations. However, it is not easy to achieve such a safety net in terms of security and performance.

This dissertation studies a dynamic safety net that enables server software to securely change the range of the safety net and achieves good performance. We have developed a system to provide such a dynamic safety net. The system consists of two mechanisms: an access control mechanism for user-level servers and a fail-safe mechanism for operating system modules. (1) Our access control mechanism allows a server process to impose appropriate access restrictions on it depending on the clients. To avoid risks involved in changing the access restrictions, this mechanism uses a new technique called *process cleaning*. Process cleaning recovers even a compromised server to be sane before changing access restrictions. (2) Our fail-safe mechanism, which we call *multi-level protection*, allows running each operating system module separate from the kernel so that misbehavior of particular modules due to software flaws does not affect the whole system. For performance improvement, the multi-level protection enables the users to lower the protection level of the modules without any modifications.

We have implemented these access control mechanism and fail-safe mechanism on Linux and NetBSD, respectively, and thereby showed that our ideas can be implemented with reasonable performance. For process cleaning, we experimented on the Apache web server and confirmed that the overhead is less than 35%. For the multi-level protection, we experimented on file system modules and network modules and confirmed that the overhead of this mechanism is less than 12% at the minimum protection level.

Acknowledgments

This thesis was supervised by Kentaro Shimizu. I would like to express my respect for him. He supported me while I studied at the University of Tsukuba and Tokyo Institute of Technology. I would also like to express deep gratitude to Takashi Masuda. He was my supervisor until the first grade in my doctoral course and has retired from the University of Tokyo. He continuously supported and encouraged me since I was assigned to the Masuda laboratory. I am grateful to Kozo Itano, who was my supervisor at the University of Tsukuba while I was the first and second grade in my doctoral course. He gladly accepted me into his laboratory as a special research student.

I greatly thank Shigeru Chiba, who was a research associate of the Masuda laboratory and belongs to Tokyo Institute of Technology now. He collaborated with me all the time since I was assigned to the Masuda laboratory. He spent a large amount of time for me and gave me a lot of suggestions while he belonged to the University of Tokyo and after he has moved to the University of Tsukuba and then Tokyo Institute of Technology. Kenji Kono, who was a research associate of the Masuda laboratory, also gave me useful comments after Shigeru Chiba has moved to the University of Tsukuba.

Moreover, I thank my colleagues of the Masuda laboratory in the University of Tokyo, the HLLA laboratory in the University of Tsukuba, and the Chiba laboratory in Tokyo Institute of Technology. In particular, discussions with Michiaki Tatsubori, who belongs to the University of Tsukuba, were very interesting.

Finally, I am grateful to my thesis committee, Akinori Yonezawa, Kei Hiraki, Yutaka Ishikawa, Shinichi Morishita, and Masami Hagiya.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The remarkable growth of software technology is making software bigger and more complex. In this trend, software involves two major problems in terms of safety. One problem is attacks against software. Software that interacts with the Internet is always exposed to danger of attacks from crackers. Even if software does not interact with the Internet, it can be attacked from local malicious users. A recent typical attack is the buffer overflow attack [103], which hijacks software, that is, takes the full control of it. Another type of attack is to exploit the careless design of software, for example, as seen in a `phf` script in a web server [7]. Once a cracker succeeds in attacks, he can abuse the privileges of the compromised software and thereby access private information.

Various techniques have been proposed to prevent such attacks. Most of these techniques detect known attack patterns and stop attack attempts before crackers succeed in attacks. For example, many techniques such as StackGuard [27] can prevent the stack smashing attack [28, 10, 77, 32]. Misuse-based or rule-based Intrusion Detection Systems (IDSs) [78, 60] can detect intruders with familiar intrusion patterns. However, these techniques have limitations that they cannot prevent new attacks or unknown attacks with different attack patterns. The reason is that the proposed techniques do not assume such unknown attacks at that time when it has been developed. In fact, an old version of StackGuard could not detect some type of stack smashing attack, which the current version can detect.

The other problem of software safety is that there exists unstable software such as software released as a beta-test version and some of irresponsible free software. Unstable software has triggers of internal errors within it and the triggers are pulled if some conditions are satisfied. Examples of the internal errors are memory leaks, lack of error handling, and so on. If unstable software triggers its internal errors, anyone does not know what happens within the privileges of the software. The software may corrupt important data or may freeze. Language or compiler supports such as using type-safe languages [71, 38] and Proof-Carrying Code (PCC) [69] mitigate the occurrence of such internal errors but it is difficult to eliminate all the errors.

A large part of the causes of attacks and most of the causes of instability are derived from software flaws. Attacks exploit hidden software flaws, which do not emerge if the legitimate users use. Conversely, instability emerges in certain conditions that trigger errors due to software flaws even if the legitimate users use. Software flaws include implementation errors and design errors. Since implementation errors are relatively easy to detect, many detection techniques have been proposed as described above. Nevertheless, yet unknown errors lie in a lot of software. Also, the programmers often make design errors and such errors are difficult to automatically detect.

Therefore, there are serious demands on the facility that minimizes damages in case that software becomes insane due to its flaws. We call this facility *safety net*. A safety net prevents such damages due to software flaws from spreading outside it. Even if crackers compromise software, they cannot access the protected resources beyond the range of the safety net. Likewise, unstable software does not make the other software outside the safety net unstable. In fact, various safety nets have been developed [33, 9, 36, 38, 1, 104, 13, 69]. Most of them are provided by the operating system or the equivalent because a safety net that software with some flaws constructs by itself is often not useful when the software becomes insane. For example, a sandbox provided by the Java virtual machine, which is equivalent to an operating system, is a safety net to protect the system from malicious and misbehavioral applets or mobile code.

1.1 Motivating Problem

This dissertation focuses on a safety net particularly for server software. The reason is that attacks against server software and instability of server software are more serious than those of the other software such as client software and standalone software. If user-level servers such as a web server are compromised by attacks, secret information is disclosed or system integrity is destroyed since servers have higher privileges generally. Furthermore, crackers may attempt to attack the other servers via the compromised server as if they were resident in the compromised server. For operating system modules such as a server-specific file system, unstable modules make an impact on the whole system because they are closely related to the operating system. If one module crashes, the whole operating system may crash.

To construct a safety net for server software, there are two obstacles. (1) Server software must always run and simultaneously handle requests from various users. Unlike the other software, server software does not generally serve only the user that executes it. Since a different safety net is needed for each user, one fixed safety net is not sufficient for server software . (2) Server software must achieve good performance together with safety. While it must handle enormous requests from the clients, it must protect itself from attacks by malicious clients and instability involved in it. However, it is difficult to satisfy both requirements.

To overcome these two obstacles, a dynamic safety net is needed for server software. A dynamic safety net enables server software to dynamically change the range of its safety net. For user-level servers, they should change the range of the safety net depending on the client that uses the server and the contents of requests from the client. For example, when a web server handles requests from Internet users, it should narrow the range of the safety net and allow accessing only public data. On the other hand, when it handles requests from Intranet users, it should broaden the range and also allow accessing private data of each user.

However, changing the range of a safety net involves security risks since attackers can abuse this ability. If a server is hijacked, the attackers can substantially invalidate the safety net by maximizing the applicable range of the safety net. If Trojan horse code is injected into a server and is activated after the server legally changes the range of the safety net, the malicious code can be executed within an inappropriate safety net. If the operating system could allow only a sane server to change the range of the safety net, these risks could be avoided. Unfortunately, it is not easy to determine whether a server has been compromised or not. Even if a server seems to be sane, a part of the internal state might have been already compromised.

For operating system modules, on the other hand, it is desirable to change the range of a safety net depending on the stability of the modules. Unlike user-level servers, operating system modules are not likely to be compromised by remote clients since such modules are usually not accessed directly from remote clients. Therefore the main issue is misbehavior of such modules. For unstable modules, the range of the safety net should be narrowed so that misbehavior due to its flaws does not cause serious damages to the whole system even if the performance is sacrificed. For stable modules, the range of the safety net should be broadened for good performance because stable modules are unlikely to behave abnormally. As such, a safety net with one fixed range for all modules is not suitable.

A challenging issue in a safety net for operating system modules is to dynamically change precedence between fail-safety and performance depending on the stability of modules. Most of previous systems are not designed so that the users can make such a trade-off. In a system that allows the user to make a suitable trade-off between the two, one requirement is that changing that precedence is transparent to the users. Any modifications to the modules should not be needed when the users adjust the trade-off. The other requirement is that the mechanism itself for transparently changing that precedence should not degrade the performance of modules. If the modules run within the broadest range of a safety net, the performance should be almost the same as the modules which do not use any safety nets from the beginning.

1.2 Approach

We have developed a system to provide such a dynamic safety net. The system consists of two mechanisms: an access control mechanism for user-level server and a fail-safe mechanism for operating system modules.

Access Control Mechanism for User-Level Servers

Our access control mechanism achieves a dynamic safety net by appropriate access restrictions depending on the clients and the contents of requests from the clients. This mechanism enables a server process to impose appropriate access restrictions on it while the process is handling requests from a client. The activities of attackers are limited by the access restrictions even if the server process is hijacked.

A unique feature of this mechanism is to allow a server process to dynamically change its access restrictions so that they fit each client. To avoid risks involved in changing access restrictions to weaker ones, that is, removing some of them from the process, our access control mechanism cleans a server process up before removing access restrictions and then eliminates malicious code injected for compromising the server. We call this cleaning-up procedure *process cleaning* [53, 54]. Process cleaning recovers even a compromised server to be sane. To use this process cleaning facility, our access control mechanism first saves the whole state of a server process when it is guaranteed that the server is still sane. Thereafter, when removing access restrictions, this mechanism restores ‘that saved state so that the state illegally modified by attackers is recovered. The restored state is all state of a process: a thread of control, a memory image, and so on.

Process cleaning can prevent attackers from gaining non-permitted access privileges. If a hijacked server process attempts to remove access restrictions from it to gain higher privileges, process cleaning restores the thread of control of the process and takes the full control back from it. Thus, while the attackers take the thread of control, they cannot remove access restrictions from the hijacked process. Also, process cleaning can prevent a server process injected Trojan horse code from activating that code after the process legally removes its access restrictions. When the process removes its access restrictions, the Trojan horse code resident in the memory is eliminated by restoring the whole memory image.

Since process cleaning is used for a server process, the performance is also important. Performance overheads due to process cleaning mainly come from saving and restoring a memory image. To reduce the overheads, process cleaning uses the *copy-on-write* technique so that only modified pages are restored. Also, process cleaning allows the users to choose a strategy for restoring the memory image. The *remap* strategy is used by default while the *copy* strategy is suitable for a server whose memory access pattern is almost the same for every request handling. In addition, to optimize the data layout of server programs for process cleaning, we have developed a tool that changes the layout based on profiling information.

We have implemented this access control mechanism on Linux and measured the overheads of our process cleaning. We experimented on the Apache web server and confirmed that the overheads of process cleaning are less than 35%.

Also, we compared our web server using process cleaning with one using the fork-join method, which is the alternative method that enables to securely remove access restrictions. The result was that our server using process cleaning is 45% faster than one using the fork-join method on average.

Fail-Safe Mechanism for Operating System Modules

By default, our fail-safe mechanism runs each operating system module within the narrowest safety net. This safety net strictly restricts the interface between the module and the operating system kernel. Each module is separate from the kernel, the other modules, and all user-level processes so that abnormal behavior of the module does not affect the rest of the system. For example, a module is run as a user-level process and only kernel functions indispensable to the module are allowed to call via a kernel trap.

To achieve a dynamic safety net, our fail-safe mechanism enables the users to change the protection level of modules depending on the stability. We call this facility *multi-level protection* [56, 55]. The multi-level protection facility provides multiple levels of protection for modules, each of which is different in the type of detectable and recoverable errors. For example, in some protection levels, the module runs within a separate address space to detect its memory access violations. In other protection levels, deadlocks between threads of modules are detected. Note that at the minimal protection level, the module is embedded into the kernel and runs without any protection.

The protection level is changed by exchanging protection managers provided per module. The protection manager plays a role of a gateway between a module and the kernel and protects the kernel from corruption by an erroneous module. As a gateway, the protection manager forwards upcalls from the kernel to the module and kernel function calls from the module to the kernel. Thus, all protection managers, each of which provides a different protection level, provide a common Application Programming Interface (API) for the modules. If a module conforms to this API, the users can exchange one protection manager for another one without any modifications to the module and then change the protection level of the module.

To change the abilities to detect and recover from errors for each protection manager, the multi-level protection uses various protection techniques. Switching address spaces enables to locate a module in a separate address space and prevent it from corrupting the other modules and the kernel. Replicating the kernel data can make a module access replicas of the kernel data and protect the kernel data from erroneous modification by the module. A wait-for-graph can be created when modules lock or wait for resources and be used to find a loop that means a deadlock.

We have implemented this fail-safe mechanism on NetBSD and measured the runtime overhead of the multi-level protection. We experimented on some file system modules and network modules. Then we confirmed that the overhead at the maximal protection level is 75% at maximum if these modules are used by practical applications. On the other hand, the extra overhead left at the minimal protection level due to multi-level protection was less than 12% even in micro benchmarks.

1.3 Contributions

This dissertation has two major contributions. One is to allow server processes to dynamically change the access restrictions. In particular, the ability of dynamically removing access restrictions is not supported in previous work since it can be a security hole. Our novel technique called process cleaning successfully makes the operation of removing access restrictions secure by eliminating the impact of attacks from server processes. Due to process cleaning, the servers can use a dynamic safety net and minimize damages by attacks.

The second contribution is to allow the users to make a trade-off between performance and fail-safety for operating system modules. In most of previous systems, the relationship of the two is fixed. Some systems allow the users to take either performance or fail-safety, but the runtime overhead for achieving that flexibility is relatively large. Our new fail-safe mechanism called multi-level protection enables the users to take only necessary fail-safety and makes the runtime overhead negligible when they take no fail-safety.

1.4 Organization

The rest of this dissertation is organized as follows. Chapter 2 mentions software flaws and a safety net. Then we describe further details of the needs for a dynamic safety net and related work for safety nets. Chapter 3 proposes process cleaning, which allows servers to securely remove access restrictions, and explains the implementation of our access control mechanism including process cleaning. Chapter 4 proposes the multi-level protection, which allows the users to change the protection level of operating system modules, and explains the implementation. Chapter 5 measures the overheads of a dynamic safety net, that is, those of process cleaning and the multi-level protection and shows the results. Finally, Chapter 6 concludes this dissertation.

Chapter 2

Needs of a Dynamic Safety Net for Server Software

2.1 Software Flaws

Most of software flaws are exploited by attackers as well as making server software unstable. These two aspects are heads and tails of a coin. For example, a buffer overflow is a flaw that the size of an input is larger than the size of a buffer and the memory area following the buffer is illegally overwritten by the input. By a buffer overflow of an input buffer of server software, attackers can overwrite some pointers such as a return address from a function pushed in a stack frame and inject malicious code as a part of the input. If the thread of the server software jumps using the modified pointers, for example, by returning from a function, the injected malicious code can be executed and the server software can be hijacked. Until the modified pointers are exploited, the injected malicious code is regarded as Trojan horse code. Once server software is hijacked, it is used for performing malicious operations. For example, the attackers can disclose or alter all files that the server software can access. Also, they can impersonate a server's owner and then attack another server by exploiting trust relationship of each other.

If the buffer overflow attack fails, the server software may get unstable. The buffer overflow attack succeeds if and only if the overwritten pointers are used for properly jumping to the injected malicious code. If the pointers do not point to that code properly, the thread of control of the server software may go out of range and that jump may cause memory access violation. Even if an overflowable buffer is not exploited by the attackers, an overflow of the buffer causes misbehavior of the server software. If a buffer allocated in a stack memory overflows, the stack frame may be corrupted and the server software may not be able to continue the execution.

Also, a race condition is exploited for attacks. A race condition is an error caused by timing when events occur. For example, when server software creates

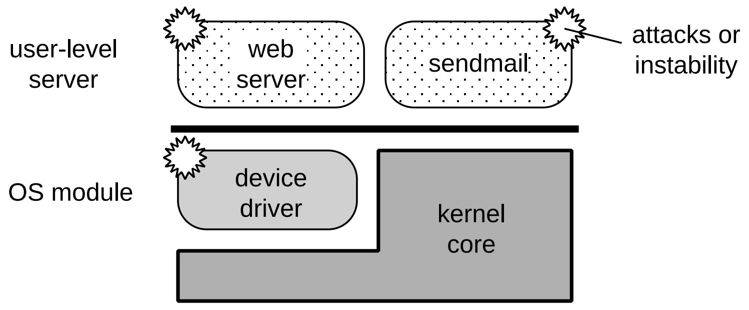


Figure 2.1: A model of the server system. Any components except the kernel core can be affected by attacks from crackers or instability of software.

a temporary file in `/tmp`, attackers can create a symbolic link to an arbitrary file that is writable to the server software and corrupt the file if the name of the temporary file is guessable. Even if a race condition is not exploited for attacks, some kind of race condition causes a deadlock. When two servers lock different files, a deadlock occurs if each server attempts to lock a file that the other server has locked unfortunately. The deadlock may freeze even the whole system.

In this dissertation, we focus on security flaws for user-level servers such as a web server although some types of instability can be also addressed by the solution to security flaws. Since user-level servers are frequently compromised by attacks from remote hosts, attacks are the most critical problem for user-level servers. In contrast, even if one server gets unstable, only the server or some related servers are affected. For operating system modules, on the other hand, we focus on software flaws that cause instability. Since operating system modules are tied up with the operating system, it is possible that their flaws affect the whole system. In contrast, such modules are unlikely to be compromised because most of them serve user-level servers but are not used from remote hosts directly.

In addition, we assume that the kernel core excluding operating system modules from the operating system kernel is the Trusted Computing Base (TCB). In other words, it does not have any software flaws and therefore all the mechanisms it provides are tamper proof and non-bypassable. Figure 2.1 shows a model of the server system we assume.

2.2 Safety Net

A safety net is defined as a defense line that can detect anomalous behavior of server software. The regular activities of sane server software are called normal behavior while irregular activities are called anomalous behavior. A safety net regards anomalous behavior as attacks from malicious users or misbehavior due to software flaws and then confines damages from insane server software within the safety net. In other words, a safety net detects violations of security policies or safety guidelines by insane server software and then prohibits accesses beyond the privileges of that server software. The examples of safety nets are access control mechanisms, Intrusion Detection Systems (IDSs), Software Fault Isolation (SFI) [104], and so on. A safety net is not dependent on patterns of attacks and can confine the impact by unknown flaws.

However, a safety net does not limit the activities within the safety net of insane server software. If a broad safety net is constructed, it cannot prevent some of anomalous behavior of server software. This is a problem called *false negative* in IDSs. In such a case, attackers can exercise higher privileges of the server and disclose or alter some resources such as privileged files. One solution is to use detection techniques specific for each software flaw together as well as both anomaly-based and misuse-based detections are used together in many IDSs. As such, a safety net detects the activities that the server software never performs while the detection techniques detect known attacks, which may include normal behavior.

The other promising solution to false negative is to specify the range of a safety net so that the range becomes as narrow as possible. In other words, normal behavior of server software should be specified so that the server has only least privilege. The principle of least privilege [92] gives a minimum set of privileges for performing necessary operations to server software. Based on this principle, server software is allowed to use only necessary resources such as system files and indispensable kernel interfaces such as system calls. However, the least privilege of server software tends to be broad because it is used under various situations. For example, the privileges that servers need change depending on the clients.

2.3 Needs of a Dynamic Safety Net

To achieve the principle of least privilege in any cases, the range of a safety net must be changed dynamically, depending on the situations. A dynamic safety net enables the operating system to keep the range of the safety net for server software as narrow as possible. However, it is not easy from the viewpoint of security and performance.

2.3.1 Client-Dependent Safety Net

The range of a safety net of a user-level server should be changed depending on the clients that use the server and the contents of requests from the clients. This is because least privilege necessary for a server is different for each request handling. Only one set of least privilege is not sufficient to construct a safety net that severely limits activities of a compromised server. Since a server is always running unlike a client, it is necessary that the applicable range of a safety net for the server be dynamically changed. For user-level servers, access restrictions to the server construct a safety net.

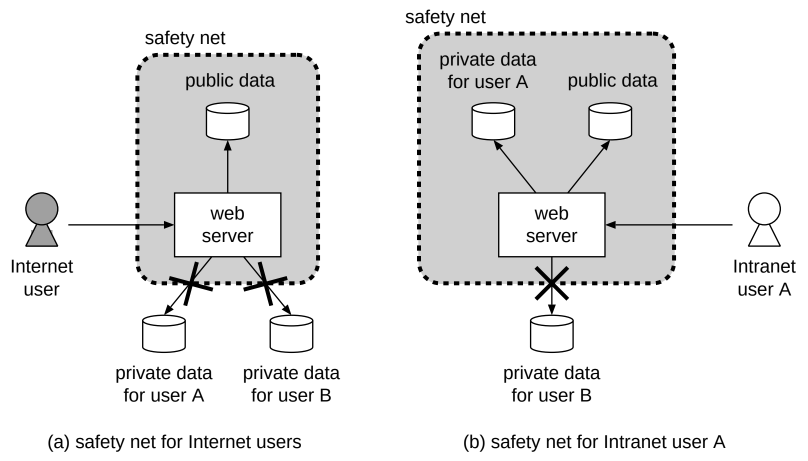


Figure 2.2: Various safety nets for a user-level web server.

Suppose that a web server is serving both the Internet and Intranet users. The administrator would like to impose as strong access restrictions as possible on the server while it is serving the untrusted Internet users so that the range of safety net gets minimum. For example, the server should be able to read only public files that anonymous users can read (Fig. 2.2 (a)). On the other hand, the administrator would like to impose weaker access restrictions on the server while the server is serving the trusted Intranet users. For example, while the server is handling a request from Intranet users, it should be able to read a private file that only the Intranet users can read so that they can see the contents of that file through the web browser (Fig. 2.2 (b)). If the server could not change its access restrictions dynamically and therefore the safety net were fixed, the administrators would need to impose the most weaker access restrictions of possible ones on the server so that the server can handle any requests from any users. In this case, remote attackers can read the private files if they compromise the server since the access control mechanism of the operating system does not prevent the compromised server from reading such private files.

Another example is a server that maintains personal schedules such as Lotus Notes [45] and Yahoo! Calendar [105]. While the server handles a request from user Alice, the server should be able to access only the data representing Alice's schedule. If the server next receives a request from another user Bob, then the system must change the access restrictions of the server before handling Bob's request so that the server can access the data representing Bob's schedule. If changing the access restrictions is not allowed, the server must be always able to access both Alice's and Bob's schedules. However, this involves a security problem since Alice can illegally access Bob's private schedule if Alice compromises the server.

In spite of these demands of a dynamic safety net, the facility of changing access restrictions involves security risks. If a server is hijacked, the attackers can remove the access restrictions from the server since the system allows even attackers to change access restrictions. As an example in UNIX, it is reported that the `seteuid` system call can be a security hole [73]. This system call is used to temporarily drop the privileges of a process, which means imposing stronger access restrictions, while the process is in danger of attacks. Since `seteuid` allows to recover the original privileges, which means removing access restrictions, whether the process is compromised or not, even a compromised server can gain higher privileges.

It is not easy to allow a server to change the access restrictions only if the server is not compromised. A server that seems to be normally running may contain malicious code injected by attackers for hijacking the server later. If this server is allowed to change its access restrictions, then that malicious code may be activated after the access restrictions are changed. Also, the execution environment of the server may be compromised. For example, if the variable `argv[0]` in a process is modified, attackers can send a `HUP` signal to the process and thereby execute an arbitrary command indicated by that variable for vulnerable software [19]. Detecting the existence of hidden malicious code and compromised execution environment is extremely difficult without serious performance penalties.

2.3.2 Stability-Dependent Safety Net

For operating system modules, it is desirable that the range of a safety net is changed depending on the stability of the module. For unstable modules, the range should be narrowed so that the modules are protected from crashes due to their software flaws (Fig. 2.3 (a)). On the other hand, for stable modules, the range of a safety net should be broadened so that the modules can execute as efficiently as possible (Fig. 2.3 (b)). For example, the operating system should allow the module to perform efficient but dangerous operations such as memory access without protection. Since the stability of modules changes over time [24] and both fail-safety and performance are significant for modules, many of which are developed for server efficiency, the fixed range of a safety net is not suitable. For operating system modules, error detection of modules constructs a safety net.

Suppose operating system modules provided by third-party vendors. Such modules such as device drivers are often unstable as seen in the Windows world and therefore often crash due to their software flaws. Even in such a case, the users may use the modules under sufficient protection if they want the functions of the modules. With sufficient protection such as memory protection, even if a module crashes, the impact is confined to only the module although the performance is sacrificed. On the other hand, many old modules are stable even if they are made by third-party vendors. In such a case, the users can take precedence of performance since fail-safety is no longer necessary.

As another example, suppose that vendors develop new operating system modules. If the developers implement a module directly in the kernel, the

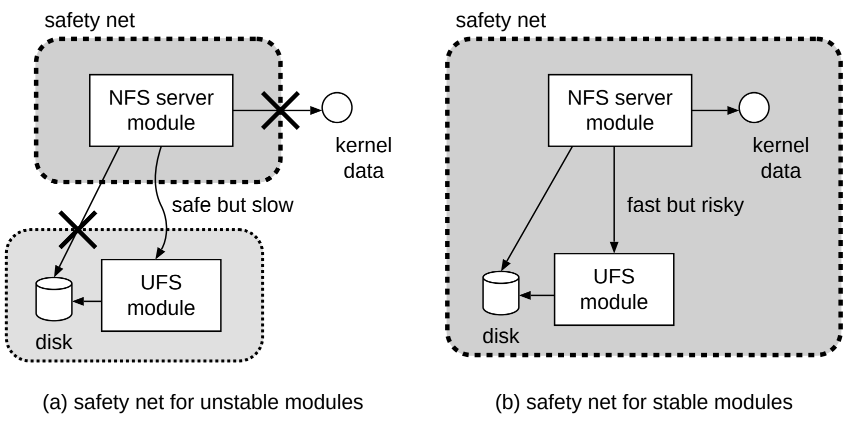


Figure 2.3: Various safety nets for a NFS server module. (a) The NFS server module can access a UFS module only through a safe but slow interface. (b) The NFS server module can access the UFS module, disk, and kernel data.

debugging is very hard. They must sacrifice fail-safety during debugging to gain good performance at product release. On the other hand, if they implement a module as a user-level daemon program to make debugging easy, they must re-implement it in the kernel after the debugging of the user-level daemon finishes since the programming interfaces between the user level and the kernel level are largely different. They need hard work to change the precedence between performance and fail-safety depending on the development phase.

Since performance and fail-safety make a trade-off as such, expanding the range of the safety net of a module for performance improvement degrades the security level, and vice versa. It is a challenging issue to dynamically make a suitable trade-off between performance and security, depending on the stability of a module. First, a system that allows dynamically making a suitable trade-off should be designed to do that in a transparent manner to the users. For example, since an interface at the kernel level is different from one at the user level, the operating system must hide the differences from the modules. Also, any modifications to a module should not be needed when the users change the precedence between performance and fail-safety. Second, the mechanism for transparently making the trade-off should not degrade the performance of the modules. If a module runs without any protection, the performance should be almost the same as an embedded version of the module, which does not use this mechanism.

2.4 Prior Work in Safety Nets for User-Level Servers

2.4.1 Access Control Mechanisms

Mandatory Access Control

Janus [36] restricts for untrusted programs to issue system calls, using a system call tracing facility of the operating system such as Solaris. The tracing facility enables a process to monitor the child processes and mediate all system calls issued by them. If untrusted programs violate security policy set by the administrators, the monitoring process rejects that access to the operating system. Janus requires neither the modification of the operating system nor the modification of user programs. The weak point is that this mechanism is for helper applications of a web browser, not for servers. This mechanism assumes that monitored processes are running under fixed access restrictions and does not allow dynamically changing the access restrictions.

Domain and Type Enforcement (DTE) [9, 8] extends type enforcement [16], which is a table oriented access control mechanism. In DTE, a subject such as a process is associated with a domain while an object such as a file is associated with a type. The Domain Definition Table (DDT) determines which domains have access to which types. The Domain Interaction Table (DIT) determines which domains have access to which domains. Using DTE, for example, the administrators can confine each server to a different domain and restrict the interaction between the domains. Since all subjects and objects are associated with domains and types, respectively, the attackers cannot bypass a DTE mechanism, which controls access rights of servers by looking up DDT and DIT. For security, the association of domains and types and the representation of DDT and DIT are configured only by the administrators as well as in Janus. Therefore DDT and DIT cannot be dynamically changed.

As another way to confine each server to a different domain, the `chroot` system call can be used. This system call changes the root directory of a process and then makes the process and the child processes run in the confined name space of the file system. For example, the root directory of an anonymous ftp server is changed to `/var/ftp` in the regular name space. The server cannot access `/etc/passwd` in the regular name space or name spaces for other confined processes. Only processes with root privileges can remove this access restrictions or the confinement to the name space. However, since attackers can also perform the same operation for servers running as root, it is not secure.

The SubDomain system [26] allows changing access restrictions in a safe manner based on probability. It provides least privilege confinement to program components such as a web server module for interpreting Perl scripts. The unique feature of SubDomain is to be able to confine not only processes but components that are a portion of a process. SubDomain allows a process to change the set of files that the process can access, using the `change_hat` system call. This system call is issued by a main program before it calls its component so that the process can impose access restrictions while it executes the component. The system call is also issued to remove access restrictions from the process when the component returns the control to the main program. To prevent the malicious component from removing access restrictions by issuing `change_hat`, `change_hat` takes as a parameter a cookie value, that is, a random number that the component cannot easily guess. As far as the component cannot read the secret cookie value, this mechanism is secure.

However, when the process is hijacked by the buffer overflow attack, the attacker can read the value from memory. To protect the memory area containing the secret cookie, the authors propose to use language-based protection such as proof-carrying code [69], strong type checking [38], and Software Fault Isolation [104], but language-based protection is not useful for some types of attacks such as the buffer overflow attack. For example, since the buffer overflow attack takes the full control of a process, language-based protection that the user-level program achieves does not work. Also, the authors consider this mechanism safe if a component is written in a scripting language and is executed by an interpreter, but user-supplied data to a script can exploit the vulnerabilities of the script or the interpreter itself.

Capability Systems

Capability-based systems such as the Amoeba operating system [101] and the TRON system [11] allows a client to temporarily grant a subset of its capabilities to a server and revoke it from the server later. If capabilities are granted to a server by a client, they extend the server's protection domain so that the server can handle a request from the client. After the client receives replies from the server, the client revokes the capabilities that it granted to the server. In such systems, the server uses the privileges of the client only while it handles a request from the client.

Unfortunately, this mechanism is based on trustworthiness of a server. Once a server is compromised or hijacked by malicious clients, capabilities that clients grant to the server after that may be abused by the compromised server. Although the clients can revoke the granted capabilities at any time, the compromised server can have enough time to abuse the client's privileges during request handling. Using our system with capability systems, a compromised server cannot abuse the granted capabilities since our system cleans a server up by process cleaning before a client grants its capabilities to the server. Thus, clients do not grant the capabilities to a compromised server. In other words, clients can always trust servers.

Role-Based Access Control

Role-Based Access Control (RBAC) [33] is similar to capability systems in that RBAC assigns the users to *roles*, to which access permissions are assigned, like capabilities. However, capability system is discretionary access control (DAC), where the users can grant access permissions to other objects, while RBAC is non-discretionary access control. In RBAC, permissions are assigned to roles by the administrators and therefore the users cannot pass access permissions to other users. One advantage of RBAC is that the cost of changing access permissions of users is low. To change access permissions, RBAC only changes the association between users and roles while DAC like capability systems must update access permissions of each user. Instead, RBAC does not have flexibility that a server can change its role by itself.

Temporally Raising Privileges

The Java virtual machine allows a method of a class to raise its privileges using the `doPrivileged` static method. In Java, each method belongs to either the system domain, which has all permissions, or an application domain, which has some permissions. The permissions of a method are not limited only by the belonging domain but also by its caller's domain. For example, when a method in an application domain calls a method in the system domain, the method in the system domain has only the permissions of the method in the application domain. Exceptionally, the `doPrivileged` method can make only the specified method ignore its caller's permissions and raise its permissions to the belonging domain's.

The only problem is that vulnerabilities of such privileged methods may be security holes. Suppose that the system library provides the `openPasswordFile` method, which uses the `doPrivileged` method so that applications can change user's passwords. Malicious applications can get a file handler for a password file using `openPasswordFile` and change the password entries as they want. In this case, the system library should provide the `changePassword` method. The `doPrivileged` method makes the designers of the system library take responsibility for avoiding such vulnerabilities.

The setuid mechanism in UNIX is similar to this `doPrivileged` mechanism. The setuid mechanism allows the users to execute a program with the setuid bit on under the privileges of the owner of the program. As a typical example, the `passwd` program is setuid-ed to `root` and even regular users can access `/etc/passwd`, which only `root` can modify, using the program. The setuid mechanism is riskier than the `doPrivileged` mechanism in Java. The `doPrivileged` method allows only a few methods to raise their privileges while the setuid mechanism allows a large program to do that. More vulnerabilities tend to be included in larger code fragments.

Intrusion Detection Systems

Strictly speaking, an Intrusion Detection System (IDS) [60, 29, 41, 3, 79] is not an access control mechanism, but it, particularly, a real-time IDS [60], can be considered as a variety of access control mechanism in that IDS prevents intruder's activities. There are two types of IDSs: a misuse-based IDS and an anomaly-based IDS. A misuse-based IDS is not a safety net since it needs signatures representing attack patterns to detect intrusions. On the other hand, an anomaly-based IDS is a safety net and can detect unknown attacks using user's statistical profile. The user profile is a collection of metrics such as a CPU load and the number of network connections. Our access control mechanism enables the users to describe only simple security policies, but various efforts of this type of IDS can be also applied to our system. The disadvantage of IDS is that the detection overheads get larger as the user model becomes more complex. Moreover, when the system switches target users for which it detects anomaly, the attackers can take a chance of compromising the system.

2.4.2 Cleanup of Processes

Fork-Join Method

The security effects by process cleaning can be also achieved with the *fork-join* method, which is used in `inetd` in UNIX. With this technique, a server issues the `fork` system call to create a new child process when it receives a request from a client. Then it imposes appropriate access restrictions on that child process, which then handles the received request. If the child process finishes handling the request, then it simply issues the `exit` system call to terminate. The child process dose not need to remove its access restrictions. While the child process is handling the request, the server can receive a new request from another client. In this case, the operating system does not have to allow to remove access restrictions and can support changing access restrictions safely.

This fork-join method and our process cleaning are equally secure because a request from a client is handled under appropriate access restrictions and the access restrictions are not removed insecurely even if the process handling the request is compromised. Namely, damages by cracking are never propagated to the next request handling. A difference between the two techniques is that process cleaning recycles an existing process for handling a next request whereas the fork-join method discards an old process and creates a new one. To be recycled, a process must be carefully cleaned up; all stains put by an attacker must be washed out before the process restarts handling a new request under a new set of access restrictions.

A significant advantage of process cleaning is better performance than the fork-join method. Process cleaning is more suitable for high-performance servers. Although process cleaning involves performance overheads, the fork-join method involves larger overheads due to process creation and destruction. Moreover, process cleaning can be used with the process pool technique since it enables to recycle the process. The process pool technique creates several processes in advance and lets them handle requests in parallel. A server with process pool is more efficient than one that spawns a new process for each request.

Checkpointing

Process cleaning can be regarded as a variation of the technique known as checkpointing and recovery. Saving the state of a process is checkpointing while restoring the saved state is recovery. Several researchers have proposed to use the `copy-on-write` technique [15, 86] for efficiently implementing checkpointing and recovery [57, 21] like our implementation explained in Section 3.3. Our contribution is to apply that technique to the security domain instead of traditional domains such as database transactions, process migration, and fault tolerance. In fact, the design of process cleaning is highly customized for access control mechanisms. For example, only updated memory pages are saved since preserving all the memory pages is not necessary. The saved memory pages are written in the kernel space, not on a disk drive, since process cleaning is not for fault tolerance.

Well-formed and partially-formed transactions [85], which are based on the Clark-Wilson integrity model [25], are also applications of checkpointing and recovery to the security domain. This mechanism optimistically allows the users to perform questionable actions, for example, that a non-root user changes her office's printer configuration. To prevent malicious users from corrupting the system integrity using this optimistic facility, this system logs system calls issued by all users so that any actions that the users perform can be rolled back. For example, even if a printer configuration is changed mistakenly, this mechanism recovers it from a copy made before it is changed. Since this mechanism can recover modifications only to the file system, it can complement our process cleaning, which recovers only the state of a process, each other. However, logging system calls and recovering the file system are heavy tasks from aspects of both disk space and time and therefore this mechanism is not suitable for high-performance servers.

2.4.3 Process Trace

Process trace is a facility that our system uses for better access control. Process trace is a similar concept to information flow. Information flow represents which data moves from where to where. For example, Multi-Level Security (MLS) systems restrict the flow of information based on the sensitivity level of the information and the clearance level of the users. The labeling of the information and the assignment of the clearance level are fixed and only the administrators can control information flow. On the other hand, process trace does not control the flow of information but the flow of execution. Therefore the access control mechanism can restrict the activities of a process based on its previous activities.

Trace of Method's Callers in Java

As described above, the security mechanism of Java grants access permissions to a method only when the permissions are granted to all the callers. To enforce this principle of least privilege, when a method is called, the security mechanism tracks all the callers one by one using the stack introspection. Our process trace mechanism is similar to this mechanism in that it traces dependencies between processes in operating systems instead of methods in Java for access control. However, tracing communication dependencies between processes is more difficult than tracking the callers of a method. The callers of a method are recorded in a local stack frame and therefore tracking them in a reverse order of calling is easy. On the other hand, to trace communication dependencies between processes, the trace mechanism has to check processes through network. When the mechanism checks that dependencies, network connections may be lost and some processes may be terminated. In this point, it is not easy to achieve the security mechanism of Java in real distributed systems. This security mechanism is a start point of our process trace.

Trace with Mobile Agents

The Intrusion Detection Agent (IDA) system [6] traces the path of an intrusion and identifies the origin of the intrusion using mobile agents. If IDA detects a mark left by a suspected intruder from audit logs, a tracing agent traces the intrusion until it reaches the origin of the intrusion. Based on the gathering information, IDA decides whether an intrusion occurs or not. Since IDA traces the path of an intrusion on demand and therefore takes much time to identify the origin of the intrusion, it is not suitable for real-time intrusion detection. In addition, every host must maintain large audit logs on all network communication. On the other hand, our taint mechanism for achieving process trace, described in Section 3.5.3, gathers information on suspected intrusions eagerly and compresses logging information for reasonable access control. For example, our taint mechanism records only a host with the lowest confidence of hosts that a process interacts with, instead of recording all of these hosts. Thereby, out taint mechanism takes less time to detect intrusions and maintains only a little amount of data.

Tainting Systems

In the tainting file system [61], files are assigned a level of trustworthiness, based on the conditions under which they are created or modified. For example, the level of trustworthiness of a file downloaded from an untrustworthy site is low. In addition to files, this system also assigns a level of trustworthiness to executing processes. The trustworthiness of a process is affected by the resources from which the process reads, such as files, network connections, and inter-process communications. The level of trustworthiness is never raised. Thus the execution of a process is restricted based on the trustworthiness of the process and files. The tainting file system is similar to our taint mechanism. Our taint mechanism differs from the tainting file system in that it can make taint information propagate via network in the distributed system. In the tainting file system, propagation of information on trustworthiness is limited within the local operating system. Since it simply assigns trustworthiness to remote hosts, it cannot trace trustworthiness of a further remote host of a remote host. This means that the tainting file system cannot detect attacks from an external host via vulnerable internal hosts.

LOMAC [35], which is based on the Low Water-Mark model [14], assigns a level of integrity to objects such as a file and subjects such as a process, instead of a level of trustworthiness. Like the tainting file system, the integrity level of files downloaded from Internet is low while that of system files is high. If a subject reads from an object that has a lower integrity level, the integrity level of the subject is reduced to the integrity level of the object. The integrity level is never raised once it is reduced. The difference between LOMAC and the tainting file system is that the integrity of objects is never changed in LOMAC. It is not allowed that a subject attempts to access an object or a subject that has higher integrity level.

2.4.4 Network-Level Authentication

Network-level authentication is also used for better access control by the operating systems. IPsec [52] is the most familiar network-level authentication protocol. To prepare IPsec communication, two hosts first authenticate each other using the Internet Key Exchange (IKE) protocol. After authentication, the two hosts communicate using the Encapsulated Security Payload (ESP) [51] protocol or a Authentication Header (AH) [50] protocol. In either protocol, authentication data is carried with each IP packet. Since IPsec authenticates the users by the unit of host, it is not suitable for authentication between a client and a server. Once the authentication between a client and a server succeeds, all client software in the client host can communicate with all server software in the server host as if they were authenticated each other. Our network-level authentication supports end-to-end, namely, process-to-process authentication and then one client software in the client host is authenticated only by one server software in the server host.

2.5 Prior Work in Safety Nets for Operating System Modules

Some operating systems take approaches to download the operating system modules into the kernel. Several UNIX systems like NetBSD and Linux allow the users to link a loadable kernel module (LKM) with the kernel dynamically. A LKM is implemented as a part of the kernel and runs very efficiently after linked with the kernel. However, vulnerabilities of the module can make the whole operating system crash because fail-safety is not considered at all.

2.5.1 User-Level Protection

Microkernel

In the microkernel operating systems like Mach [1], the users can implement the operating system modules as user-level servers. The user-level servers are separate from the kernel and execute the functions by communicating with the kernel and the other modules. Errors in one module are confined to only the module and the rest of the system is not affected. However, at the early stage of the research, all subsystems such as file systems and network subsystems are included in the single UNIX server [37]. Therefore, if one subsystem crashes due to the errors, the whole UNIX server crashes although the microkernel itself is not affected. In this system, the operating system can provide sufficient fail-safety only among multiple operating system personalities such as the UNIX server and the MacOS server and the microkernel. This type of fail-safety is too coarse-grained.

To improve this insufficient fail-safety between subsystems, a Mach that supports a multi-server model is proposed [39]. For example, user-level protocol servers [87] allow a new network protocol to be implemented and thereby errors of each protocol server do not affect the rest of the operating system. An erroneous server is simply terminated by the kernel. The multi-server system achieves sufficient fail-safety in a fine-grained manner but the performance is sacrificed because the overheads of inter-process communication (IPC) and context switches between subsystems are large [12, 20].

The performance of this cross-domain communication has been improved in the second generation microkernels [34, 58, 17]. For example, the L4 microkernel [58] uses recursive address spaces, which is a mechanism that a user-level server can grant a portion of its address space to other servers to make IPC fast. The performance overhead of the Linux server on top of L4 is from 5% to 10% in typical applications such as compiling the Linux server [40]. However, the overhead of Remote Procedure Call (RPC) on UDP is 50% and applications that frequently use user-level modules are not optimized sufficiently. This mechanism can be used together with our multi-level protection to make a module that takes precedence of fail-safety more efficient.

The Chorus operating system [90, 89] allows the users to download the operating system modules created as user-level servers into the kernel. At this time, recompiling them is not needed. Chorus achieves sufficient fail-safety when the modules are running at the user level. On the other hand, Chorus achieves good performance when the modules are downloaded into the kernel level. This approach is similar to our multi-level protection. The difference is a basic concept. Our multi-level protection starts from the monolithic kernel design and runs the modules at the user level to make them safe. The communication between the modules and the kernel is optimized so that the performance is maximum when the modules are running at the kernel level. In contrast, Chorus starts from the microkernel design and downloads the modules running at the user level to make them efficient. The communication between the modules and the kernel is done by IPC even if the modules are running at the kernel level. Therefore the downloaded modules are not enough efficient. For instance, in the read operation, the downloaded file system is 80% slower than the same file system hand-crafted in the kernel [5].

Operating Systems as User-Level Library

The Exokernel operating system [31, 49] and some operating systems using a protocol library [62, 102] link functions of the operating system implemented as a library with application programs. In Exokernel, in particular, almost all functions of the operating system are exported to the library to reduce the number of cross-domain switches between the applications and the kernel. The kernel provides only the facilities to bind resources securely and multiplex physical devices. The operating system modules are implemented as a portion of the library. In such systems, the fail-safety of the modules is sufficient since only the application that is linked with a library including an unstable module is affected. However, developing new modules in Exokernel is difficult due to the low abstraction Exokernel provides. Since Exokernel provides a big library including most functions of the operating system in the same abstraction with the monolithic kernel, they need the same efforts with ones for modifying the monolithic kernel. The difficulties of the module development cause more software flaws.

2.5.2 Kernel-Level Protection

Some operating systems take an approach that they download the operating system modules into the kernel and protect the modules from the rest of the kernel.

Fault Isolation

The VINO operating system [96, 97] uses software fault isolation (SFI) [104, 98] to protect the kernel from illegal memory accesses due to the downloaded extension modules. SFI is a technique that a compiler inserts check code of address for read, write, and jump into a module. If the module attempts to access out of allowable memory range, the inserted check code detects its violation. VINO also limits the maximum amount of resources that the modules can use at any given time and automatically releases the resources if a certain timeout expires. To recover from errors, VINO provides a kernel transaction system. Thus VINO provides a relatively lightweight and sufficient fail-safety. However, VINO entails certain fixed overheads even if the modules are enough stable. For instance, the overhead of SFI is always from 5% to 200%, depending on applications.

Language/Compiler Support

As another approach to download the modules into the kernel safely, several operating systems use language supports. The packet filter [66] is downloaded into the kernel and multiplexes network packets to appropriate applications. The language for the packet filter is specific to the domain of packet multiplexing and does not allow to describe malicious filters. However, such domain-specific mechanisms are not suit for general modules since the power of expression is low. Moreover, the safe execution of packet filter gets rather overheads. In the early stage, packet filters were interpreted in the kernel. To reduce the overheads, more efficient packet filters have been proposed. The BSD packet filter [63] redesigns the original stack-based packet filter and is up to 20 times faster. The dynamic packet filter (DPF) [30] uses dynamic code generation and is 10% to 50% faster than the other fastest packet filters.

The SPIN operating system [13] allows the users to download the operating system modules written in the Modula-3 language [71] into the kernel. Modula-3 is a type-safe language and the modules written in Modula-3 do not cause memory access violations at runtime. To detect memory violations that the compiler cannot statically check, SPIN needs runtime checks such as the range check of array boundaries. The overhead is small for practical modules. One disadvantage of Modula-3 is that it does not allow to cast data structures to and from the array of bytes in a safe manner. The cast is needed to handle protocol headers in the array of bytes from a network device driver without extra copies. To allow safe cast, SPIN added the `VIEW` operator to Modula-3 [44]. `VIEW` translates the given array of bytes to the specified data structure, and vice versa. The translation is safe since the translated data structure must be within the range of the array. This safe cast does not cause any memory violation but may cause data corruption due to casting the array to an unexpected data structure. In other words, the `VIEW` operator breaks strict type-safety.

The other disadvantage is that Modula-3 has not been used for system programming for a historical reason. So far, the system programmers mainly use the C language, which has unsafe pointers and enables to write as efficient programs as the assembly language. The developers of SPIN say that it is easy that the programmers shift their programming language from C to Modula-3. However, many system programmers have a lot of know-how for developing system programs with C and hence the shift is not very easy. This is also the reason why system programmers continue to use C (or C++) for a long time.

Proof-Carrying Code (PCC) [69] can be used to guarantee safe behavior of operating system modules. First, the PCC compiler generates a proof that a source program adheres to a given safety rules and forms the PCC binary from the compiled program and the proof. When the binary is downloaded into the kernel, the operating system validates the proof, which is a part of the PCC binary. If the validation is passed, the program binary is executed without additional runtime checks. PCC is applied to the network packet filter, the IP checksum, the type-safe assembly language [68], and so on. However, pure PCC cannot perfectly prove all programs written in C although the CCured type system [70] can prove most pointers of many C programs to be type safe. Moreover, the overhead of runtime checks for type-unsafe pointers in CCured reaches 150% at maximum.

2.6 Summary

This chapter first described software flaws that cause attacks and make server software unstable. A buffer overflow is exploited for hijacking server software while it often crashes server software after the buffer overflow attack fails or even if the attack does not happen. A race condition is also exploited for modifying files at the server's privileges while it may freeze the whole system by deadlocks. These two aspects of software flaws are heads and tails of a coin. In this dissertation, we deal with attacks for user-level servers and instability for operating system modules since attacks are always threats for user-level servers and unstable operating system modules are threats to the whole system.

Next, this chapter mentioned a safety net for server software. A safety net is a defense line for detecting anomalous behavior of server software. Anomalous behavior is regarded as attacks from malicious users and misbehavior due to software flaws and is detected by the operating system. A safety net has a disadvantage that the false negative rate can be large depending on the security policy. To address this disadvantage, server software should change the range of its safety net depending on the clients so that server software always follows the principle of least privilege. This type of safety net is called dynamic safety net.

Enabling a dynamic safety net is not easy from the viewpoint of security and performance. For user-level servers, the range of the safety net should be changed depending on the clients or the requests. However, the facility of changing access restrictions can be a security hole since attackers may hijack the servers or inject Trojan horse code into the server for executing it later. For operating system modules, on the other hand, it is desirable that the range of the safety net is changed depending on the stability of the modules. However, it is not easy to make a trade-off between performance and fail-safety so that changing precedence between them is transparent to the users and the performance of the modules that take precedence of performance is enough good.

Finally, this chapter described prior work related to safety nets. For access control mechanisms of user-level servers, the advantages and disadvantages of previous work are shown in Table 2.1. Mandatory access control (MAC) and Role-Based Access Control (RBAC) do not allow a process to change the access restrictions. SubDomain, which is also MAC, allows a process to change the access restrictions but that mechanism is not secure for some type of attacks. Capability systems are also insecure when the server is compromised. The fork-join method, which is the most comparable to our process cleaning, enables a process to change access restrictions in a secure manner but the performance is not good.

For fail-safe mechanisms of operating system modules, the advantages and disadvantages of previous work are shown in Table 2.2. A loadable kernel module (LKM) is insecure. Mach, which runs a module as a user process, is secure but achieves poor performance. L4, which IPC has been improved, achieves relatively good performance for some applications but the overheads are still large for some applications. Chorus achieves both sufficient fail-safety and good performance by running the modules either at the user level or at the kernel level. However, the overhead of modules running in the kernel is slightly large and the trade-off is the alternative of the user level or the kernel level. Exokernel provides only too low abstraction for the module programmers to develop the

Table 2.1: Comparison of access control mechanisms. $\surd$ means to satisfy the condition sufficiently.

	Dynamicity	Security	Performance
MAC		$\surd$	$\surd$
SubDomain	$\surd$		$\surd$
Capability systems	$\surd$		
RBAC		$\surd$	$\surd$
Fork-join with MAC	$\surd$	$\surd$	

Table 2.2: Comparison of fail-safe mechanisms. $\surd$ means to satisfy the condition sufficiently while $\triangle$ means to satisfy it in some degree.

	Safety	Perfor-mance	Abstrac-tion	Expression power	Feasi-bility	Trade-off
LKM		$\surd$		$\surd$	$\surd$	
Mach	$\surd$		$\surd$	$\surd$	$\surd$	
L4	$\surd$	$\triangle$	$\surd$	$\surd$	$\surd$	
Chorus	$\surd$	$\triangle$	$\surd$	$\surd$	$\surd$	$\triangle$
Exokernel	$\surd$	$\surd$		$\surd$	$\surd$	
VINO	$\surd$	$\triangle$	$\surd$	$\surd$	$\surd$	
DPF	$\surd$	$\surd$	$\surd$		$\surd$	
SPIN	$\surd$	$\surd$	$\surd$	$\surd$		
CCured	$\surd$	$\triangle$	–	$\surd$	$\surd$	

modules. VINO, which uses a compiler support, achieves sufficient fail-safety and relatively good performance but the overheads is dependent on the modules. Dynamic Packet Filter (DPF) is secure and fast but the expression power is low because of a domain-specific language. The approach of SPIN is the best in the listed approaches but SPIN has a problem of the feasibility because of using Modula-3, not C. CCured achieves sufficient fail-safety but the performance is dependent of the applications. Since CCured is not a framework for operating system modules, the abstraction cannot be estimated.

If there is an approach that satisfies safety, performance, abstraction, power, and feasibility, the ability of making a trade-off between performance and fail-safety is not necessary, but any approaches do not satisfy all abilities. Our approach is also considered as the improvement of Chorus in that it reduces the overheads of the modules running at the kernel level and enables to make a more flexible trade-off between performance and fail-safety.

Chapter 3

Access Control Mechanism for User-level Servers

We have developed the Compacto operating system, which provides a novel access control mechanism for user-level servers. In this chapter, we present our access control mechanism, in particular, a unique feature called *process cleaning* and the implementation.

3.1 Safety Net by Access Restrictions

Compacto constructs a safety net by imposing access restrictions to a server process. To flexibly change the range of the safety net, Compacto uses various information on the server process and the clients. For example, the owner's user ID of the server process and the network address of each client are used as well as in UNIX. Furthermore, the information on the previous activities of each client and the authentication information by the operating system are also used.

3.1.1 Imposing Access Restrictions

The Compacto operating system allows to dynamically impose access restrictions on a process at the system call level to limit accesses to the operating system. Compacto can prohibit issuing some types of system calls with some kinds of parameters. This access control complements the existing UNIX access control. Even for the super user, who is almighty in UNIX, Compacto can limit the privileges so that the super user can perform only necessary operations. For example, if a non-privileged server program needs a privileged network port, Compacto gives only the privilege for using a certain privileged port, not the full privileges, to the server. Such access restrictions are enforced for a process by the operating system kernel to ensure security and are inherited from a parent process to the child processes.

Table 3.1: System calls that Compacto can limit.

Category	System calls
File management	`open`, `unlink`, etc.
Network management	`connect`, `accept`, etc.
Process management	`exec`, `setuid`, etc.
Memory management	`mmap`, `mprotect`, etc.
System administration	`reboot`, `mount`, etc.

The system calls that Compacto can limit are categorized as Table 3.1. The list of limitable system calls is shown in Appendix A. For each process, Compacto can either allow or deny each of these system calls. Since the access control of Compacto is complement to that in UNIX, only access restrictions in UNIX are applied to a process by default. In other words, system calls that are not denied explicitly are allowed. Suppose that a ssh server has an IP address of 131.112.40.1 and waits for a ssh client in TCP port number 22. When the ssh server receives a request from a client whose IP address is 131.112.40.65 and TCP port number is 1023, an example of policy rules applied to the server are as follows:

```
allow recv 131.112.40.1:22 from 131.112.40.65:1023
deny recv * from *
allow send 131.112.40.1:22 to 131.112.40.65:1023
deny send * to *
```

These rules enforce the server to communicate only with the client that sent a request.

Imposing such strict access restrictions prevents the rest of the system from being damaged in case that one server is compromised. Suppose that an attacker compromises a server, for example, by the buffer overflow attack. If the server is running under the access restrictions specified by the policy rules above, the attacker cannot use that server for compromising other servers because that server cannot communicate with other servers.

Compacto can limit issuing system calls not only by the type and the parameters passed such as a file name but by attributes of a process. The attributes are a program name, an owner's user ID, and so on. For further limitations, Compacto uses the attributes of not only the current process but the ancestor processes. The program name for an ancestor process is helpful when Compacto imposes access restrictions depending on the login style of the user. If there exists `sshd` in the ancestor processes, Compacto can determine that the user logs in from a remote host and impose stronger restrictions. The sample policy rules for this access control are as follows:

```
allow write "/etc/passwd" via-prog "login"
deny write "/etc/passwd" via-prog "sshd"
```

Also, the owner's user ID of an ancestor process is helpful for limiting the privileges of programs setuid-ed to `root`. Compacto can limit the process of the root-setuid program and the descendant processes by the original user, who executed the root-setuid program. The use of the attributes of ancestor processes is deeply related to process trace described in Section 3.1.2. As other conditions, a taint level described in Section 3.5.3 and user authentication information using certificates described in Section 3.1.3 are also used. These details are described in each section. All conditions are listed in Appendix A.

Policy rules that the users can describe in our access control mechanism are quite simple. Describing a security policy with such a fine-grained specification is cumbersome. To mitigate this problem, we have to provide higher-level policy rules and tools to help describing rules. However, our aim of this research is to enable to securely change access restrictions. For ease of describing policy rules, several access control mechanisms have been proposed [2, 9].

3.1.2 Process Trace

Since Compacto controls the access restrictions of a server process based on a direct client process, the servers cannot prevent attacks from trusted hosts that attackers have compromised. If an attacker first compromises a vulnerable server and then accesses a victim server from the compromised server, the victim server may regard its access as one from a trusted client. The delegation of restrictors is useful for propagating client's access restrictions. In the above case, the victim server can get the restrictors of attacker's original process and can prevent the attack. However, that delegation is limited within one administrative domain.

To obtain detailed information on attackers, Compacto traces the activities of the attackers. Suppose that an attacker that resides in client Z attempts to compromise server B. Server B can prevent direct attacks from client Z using the safety net. However, if client Z can compromise server A, which is trusted from server B, the attacker can also compromise server B via server A. (Figure 3.1). With the process trace facility, when the attacker in server A attempts to compromise server B, server B can recognize that the access is from client Z via client A. Using this additional information, server B can restrict the activities of server A based on the information on client Z. Namely, server B can prevent the attempt of the indirect attack by client Z.

For this process trace, Compacto uses dependencies between processes. Compacto traces an impact that a process makes on the other processes based on those dependencies. For example, information of which process is executed in which order from a server process shows whether the server process is used in a proper manner or not. If a web server directly executes a shell program, Compacto can regard that activity as an intrusion exploiting security flaws and can make that execution fail. If a server is connected from a trusted host that has a trail of intrusion, Compacto can deny logging in from that host.

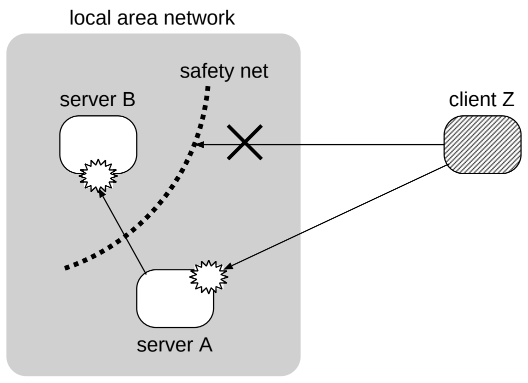


Figure 3.1: An indirect attack to server A via server B.

3.1.3 Network-Level Authentication

Compacto decides access permission of a remote user based on the IP address of a host where she resides. As a typical example, Compacto allows remote login for the internal users resident in the hosts within the same local network while Compacto denies that for the external users resident in the other hosts. Although Compacto can allow remote login even for users resident in specific external hosts such as hosts of a partner corporation, it cannot be guaranteed that remote login is really from hosts with permissions. The IP address of external hosts cannot be trusted because of potential IP spoofing.

However, it is inconvenient that a trusted remote user resident in an external host is not allowed to access services provided by internal servers. If even an internal user goes to the other site, she may not be able to log in her own host since the host decides that she is one of untrusted users. In traditional systems, user-level servers authenticate the users that attempt to use their services. Authentication at the user level is bypassable and is therefore vulnerable to attacks before the authentication process completes. Also, some servers such as `in.rlogind` do not authenticate remote users in a secure manner.

Compacto authenticates the remote users at the network level. In other words, the operating system automatically authenticates a remote user when the user connects to an internal server. This authentication is not bypassable for the remote user. Also, since this authentication uses public key infrastructure, it is difficult that an untrusted remote user impersonates a trusted user. The other reason why the operating system supports server authentication is to secure all servers even if some of them are not concern about network security. In fact, this network-level authentication is transparent to user-level servers.

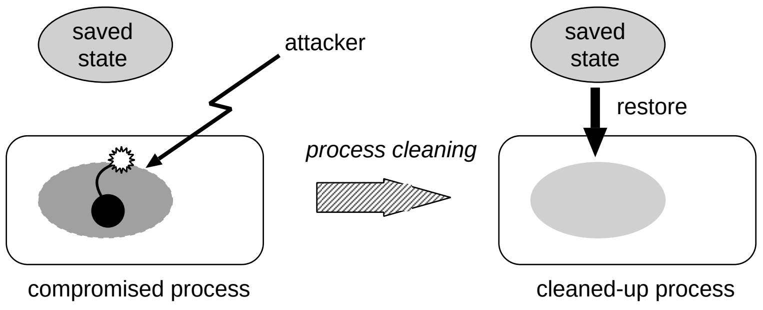


Figure 3.2: The basic concept of process cleaning. The system restores the saved state and cleans up the process compromised by the attacker.

3.2 Dynamic Change of Access Restrictions

To dynamically change access restrictions of a server process, the process first has to remove some of them and then impose new access restrictions. From the viewpoint of security, it is not a threat that a compromised process imposes more strict access restrictions on itself although the attacker can attempt the Denial of Service (DoS) attack by that. However, removing access restrictions is a threat.

3.2.1 Process Cleaning

Providing the ability to remove access restrictions is not a naive task. At least, a compromised server must not be able to remove access restrictions for obtaining full access privilege. Only sane servers must be able to do that. However, it is difficult to detect whether a server is compromised or not.

To solve this problem, Compacto cleans a process up before removing access restrictions and eliminates injected malicious code for compromising the server as illustrated in Figure 3.2. This means that even a compromised server is recovered to be sane. We call this cleaning-up procedure *process cleaning* [53, 54]. The idea of process cleaning is simple. First, programmers save the whole state of a server process when they can guarantee that the server is still sane. Then, Compacto restores that saved state when access restrictions are removed so that the state illegally modified by an attacker is recovered and thereby the server becomes sane. The restored state includes the followings:

- the thread of control;
- the memory image, including stack, heap, and environment variables;
- signal handlers;
- status of open files and sockets;

```
save_state();                                  (1)
fd = accept();                                 (2)

   authenticate a client                       (3)

rid = create_restrictor(policy rules);         (4)
bind_subject(rid, getpid());
bind_object(rid, fd);

   handle a request                            (5)

restore_state();                               (6)
```

Figure 3.3: An example code of a server using process cleaning.

- the user and group ID; and
- all other state.

Note that the thread of control (that is, the instruction pointer) is restored. Hence, even if malicious code injected by an attacker is running, the execution of that code is terminated.

Compacto provides two system calls for removing access restrictions with process cleaning: `save_state` and `restore_state`. The `save_state` system call saves the state of a process listed above. It also records all the access restrictions imposed at that time. The `restore_state` system call first restores the process state saved by `save_state` and then removes all the access restrictions imposed on the process since `save_state` was last issued.

3.2.2 Example

Fig. 3.3 is a simplified example code of how the `save_state` and `restore_state` system calls are used by a server. (1) After finishing initialization, the server issues `save_state` system call. (2) Then it waits until a client connects to it. (3) If a client connects, the server first authenticates the client user or obtains the IP address of the client host. (4) It creates a restrictor from policy rules depending on the client and binds the restrictor to the server process and the socket for request handling. (5) Then the server handles the request from that client under applied access restrictions. (6) After that, the server issues the `restore_state` system call. This system call restores the state of the server process; it removes the access restrictions imposed at (4). Since it also recovers the instruction pointer, the thread of control is moved back to the next statement of (1). The server can repeatedly handle another request from a client.

Compacto disables an attacker from obtaining unlimited access privileges if the attacker compromises a server on top of Compacto. Since the compromised server is under appropriate access restrictions, it cannot illegally access protected resources. If it attempts to remove the access restrictions, the control is recovered back from the attacker. In contrast, the compromised server can continue to run without removing the access restrictions. In this case, the server is not allowed to access protected resources and thus is secure for the rest of the operating system although it is compromised. As a last resort, the attacker can execute an infinite loop for the purpose of disabling the service (the denial-of-service attack). If Compacto can detect that the service is disabled, for example, using statistical information, Compacto can interrupt the execution of the server and force to issue `restore_state`.

In the example above, the server program was sequential. To make it parallel, we must use the *process pool* technique, with which several processes are created in advance and they handle a request in parallel. In the case of the above example, all processes execute the program in Fig. 3.3 in parallel.

3.2.3 Effectiveness

Process cleaning can prevent the following kind of attacks derived from removing access restrictions:

- Hijacking a server process, removing the access restrictions, and then gaining non-permitted access rights, such as the privileges of administrator or other users. Restoring the thread of control of the process can take the full control back from the hijacked process.
- Injecting Trojan horse code to a server process, for example, by the buffer overflow attack, and activating it for hijacking the process after the process removes the access restrictions and regains higher privileges. The Trojan horse code can be activated by modified pointer variables or signal handlers. Restoring a memory image of the process can eliminate the Trojan horse code from the memory. It also clear pointer variables modified so as to jump to such code. Moreover, restoring signal handlers can avoid jumping to such code by sending signals.
- Modifying the execution environment of a server process so that the server process performs malicious operations after access restrictions are removed legally. For example, if the PATH environment variable is modified, unexpected external programs may be executed. Restoring a memory image can undo such modifications.

Additionally, process cleaning can prevent a kind of denial-of-service attacks using malicious code injected by an attacker:

- Randomly modifying part of the memory of the process.
- Closing files or sockets in use. Or, raising the limits of maximum resource consumption.

These attacks makes a server process unstable or, worse, terminate. If Compacto can detect the unexpected termination, it can execute process cleaning on that server so that the server can recover.

Compacto is responsible for protecting a server on top of that from attacks described above. The server developers are released from troublesome tasks for protecting the server from those attacks. They do not need to be frightened of unknown attacks since Compacto detects them.

3.2.4 Limitations

Process cleaning can protect a server from attacks described in the previous section. However, process cleaning unfortunately does not protect a server from all types of attacks. The server developers still have responsibility for carefully implementing their servers so that the security of the servers cannot be broken by other attacks, for example,

- The server may be compromised before access restrictions are imposed. If so, the compromised server can obtain full access privileges. A typical server is not affected by this type of attacks since it imposes access restrictions just after connected by a client.
- The compromised server may exploit another running process or an external resource such as file and network, which are not included in the state restored by process cleaning, for obtaining unauthorized access privileges. Strict restrictions to inter-process communication and accessing external resources can mitigate damages by this type of attacks.
- The compromised server may execute a local program that has vulnerabilities and compromise it. In this case, the attacker can obtain the full control of the local program even after the `restore_state` system call. Restricting local programs that the server can execute is useful for this type of attacks.

Also, Process cleaning involves limitations on the applicability to servers. It cannot be used for stateful servers because all side effects caused since the last process cleaning are cleared whenever the `restore_state` system call is issued. All requests must be processed independently of each other; even recording log data on memory or caching the contents of frequently accessed files are not allowed. To avoid this limitation, a server state must be stored in shared memory or files, which are not restored by the `restore_state` system call. However, this fact may be used for injecting Trojan horse code in a server. Since process cleaning does not protect servers from security attacks using this fact, server developers must be responsible for that the method of maintaining a server state does not involve any vulnerability.

Another limitation is that a server must be single-threaded. Compacto cannot impose or remove access restrictions on a per thread basis since a thread is not securely protected from the other threads in the same process and thus access control per thread is useless. It is relatively easy that a thread hijacks the execution of another thread. Besides, process cleaning cannot selectively restore part of the process state used by a particular thread. One remedy for this limitation is to introduce securely protected threads [47, 100, 23] and extend the functionality of Compacto to be a per thread basis.

3.2.5 Comparison with the Fork-Join Method

Process cleaning can achieve the same level of security with the fork-join method, which is the alternative technique for securely changing access restrictions and is described in Section 2.4.2. The fork-join method can prevent the attacks that process cleaning can prevent, which we describe in Section 3.2.3. In the fork-join method, a server process can continue to maintain the thread of control even if the child process spawned for handling a request is hijacked. Thus, the server process can make the hijacked child process terminate and then recover the thread of control of the child process from the attacker. Also, Trojan horse code injected into the memory of the child process is eliminated by terminating the child process. In addition, the modified execution environment of the child process does not affect the server process and the other child processes since the execution environment of the child process is isolated from the others.

In terms of security, limitations of the fork-join method are also the same with those of process cleaning, which is described in Section 3.2.4. If a server process itself is compromised before spawning a child process, an attacker can abuse the server process. Even after a child process is spawned successfully, it can be compromised before imposing appropriate access restrictions. Moreover, the fork-join method can eliminate only an impact made on a child process. An impact that a hijacked child process makes on the other processes or the external resources such as files is left in the system. The fork-join method cannot also prevent local programs from being compromised by a hijacked child process.

In terms of applicability, limitations of the fork-join method are also almost the same with process cleaning. In the fork-join method, a routine for request handling in a child process must be stateless. If the child process changes the state of the parent process or the other child processes, the impact is not eliminated when the child process is terminated. Also, the server process cannot let a thread handle a request without using a child process since it spawns a child process for each request. However, process cleaning limits the functionalities of a server process, comparing with the fork-join method. The fork-join method allows a process to use threads only within the server process and the child process while process cleaning does not allow using threads. Also, the fork-join method allows a child process to issue the `exec` system call whereas process cleaning does now allow it. To execute a new program in a server with process cleaning, the server process must first spawn a new process by the `fork` system call and then issue the `exec` system call. Fortunately, these limitations of functionalities do not degrade server security and make a server more efficient.

For the comparison of the performance between the fork-join method and process cleaning, we show the details in Section 5 based on our experiments.

3.3 Implementation of Process Cleaning

3.3.1 Restored Process State

For process cleaning, the `restore_state` system call restores the state of a process as follows.

Thread of Control

The `restore_state` system call restores the values of all registers. It is analogous to `longjmp` included in the standard C library. Those registers include the instruction pointer and the stack pointer. If the `restore_state` system call is issued, the program counter is reset so that it points to the instruction next to issuing the `save_state` system call. Also, the stack pointer is reset so that it points to the stack frame when this system call was issued. If `save_state` is recursively issued, then the previously saved process state is overwritten. Successive issues of `restore_state` restore the state saved by the last issue of `save_state`.

Memory Image

The `restore_state` system call restores the contents and protection mode of all memory pages. What it restores is the stack frame, the heap, the area of environment variables and arguments of a process, static data, and so on. It also restores the memory mapping between physical and virtual address and the data segment size changed by the `brk` system call.

Since it is difficult to distinguish between the pages modified by an attacker and the pages properly modified by the process, the whole memory image is restored. Pages that did not exist when issuing the `save_state` system call are unmapped and discarded. For a page that was unmapped after `save_state`, Compacto allocates a new page and restores the contents. To do that, Compacto maintains pages unmapped after `save_state`. Exceptionally, pages for the stack frame expanded after `save_state` and pages for the BSS segment paged in after `save_state` are zero-cleared because the contents are indefinite.

The implementation details for optimization are described in the next section.

Signal Handler

The `restore_state` system call restores signal handlers, which may be overwritten by an attacker. Before that, the `restore_state` system call delivers pending signals to a process so that all the signals are handled by old handlers. This is needed to prevent a signal from being processed by a handler modified by an attacker. Pending signals are not saved by `save_state` and therefore are not restored by `restore_state`.

Open File and Socket

The `restore_state` system call closes all the files and sockets that have been opened after the `save_state` system call so that illegal network connections can be shut down. It re-opens all the files and sockets that have been closed after `save_state`. To re-open them correctly, Compacto keeps all objects for files and sockets even if they are closed by attackers. The `restore_state` system call also restores the file descriptors for them.

Other State

The `restore_state` system call also restores the state of the following resources: the user ID and the group ID set to a process by `setuid` and `setgid`, the current working directory changed by `chdir`, the root directory confined by `chroot`, resource usage limits set to a process by `setrlimit`, the file creation mask by `umask`, the Linux capability set, and the priority set by `nice`. In addition, the taint of a process is restored. The details of the taint are described in the Section 3.5.3.

Note that profiling information including the CPU usage is not restored.

3.3.2 Efficient Memory Save/Restoration

Performance penalties of process cleaning are mainly due to copying memory for saving and restoring the state of a process. A naive implementation, which saves and restores the whole memory image, is slower than the fork-join method. To reduce the amount of saved memory, Compacto uses a technique known as *copy-on-write* [15, 86]. Furthermore, Compacto provides two implementation strategies for restoring a memory image. As shown in Fig. 3.3, a typical server running on Compacto saves its state only once and repeatedly restores the saved state whenever it finishes handling a request. The user can select an implementation strategy so that process cleaning works efficiently in that case.

Delayed Save

The `save_state` system call does not immediately duplicate the whole memory image when it is issued. It first changes the state of every writable memory page into the write-protected mode. The memory page is duplicated only if the process attempts to write in the page and hence a page fault occurs. The page table is modified so that the original page frame is moved into the kernel address space (Fig. 3.4 (1)) and a new page frame allocated for the duplication is mapped at the original virtual address (Fig. 3.4 (2)). It is the original page frame that must be kept in the kernel address space because it may be shared with parent and/or child processes for copying on write. The original page frame in the kernel space is untouchable for the server process. We call the newly allocated page frame a *shadow* page. Since the shadow page is writable, no page fault occurs after the first one.

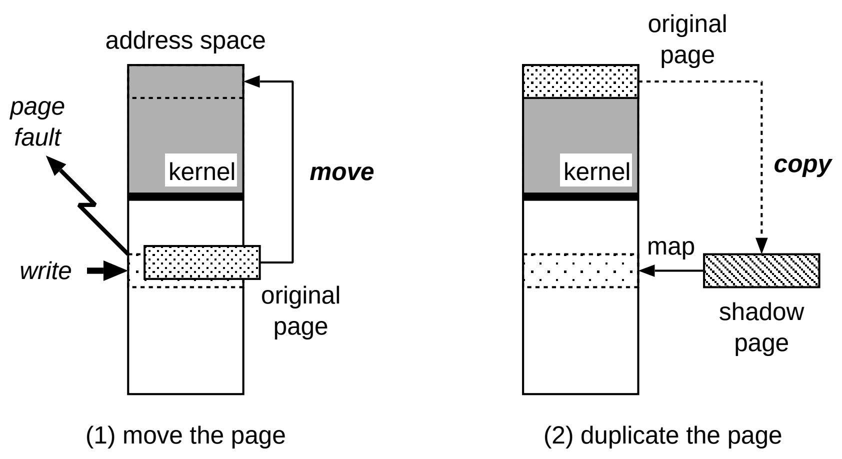


Figure 3.4: Delayed memory save with copy-on-write.

Also, Compacto clears the dirty bit of the saved page so that Compacto can detect modified pages in the `restore_state` system call. At this time, Compacto saves the dirty bit of the saved page. The dirty bit is important for paging algorithm since it is used to determine frequently used pages. Moreover, a page with a dirty bit is written in the swap area when it is swapped out.

Restoration Strategies

The `restore_state` system call restores only the memory pages that have been duplicated since the last `save_state` system call was issued. For restoring them, Compacto can choose one of two strategies. The first strategy is to unmap a shadow page, discard it, and move the original page frame back from the kernel address space (Fig. 3.5 (a)). We call this the *remap* strategy. The second strategy is to copy the contents of the original page frame into the shadow page. The original page frame remains in the kernel address space (Fig. 3.5 (b)). We call this the *copy* strategy.

Since the remap strategy does not need to copy memory for the restoration, Compacto normally selects this strategy. It is suitable for a request performed only once like login authentication. However, a typical server running on Compacto repeatedly restores the same state saved by the `save_state` system call at the beginning. In this case, the remap strategy may be less efficient than the copy strategy. If Compacto uses the remap strategy, the `restore_state` system call must change the state of the restored memory page into the write-protected mode so that it is duplicated again if the process attempts to write in that page until the next issue of the `restore_state` system call. If a page fault occurs, the page fault handler must again allocate a shadow page and copy the contents of the page into it.

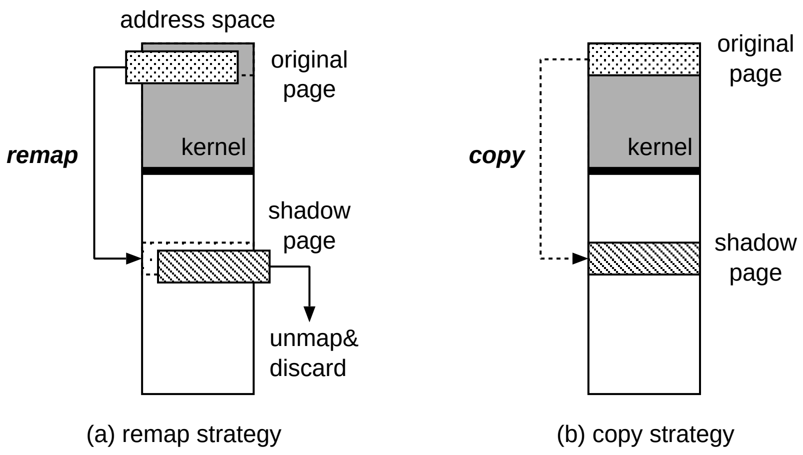


Figure 3.5: Two strategies for restoring memory.

On the other hand, with the copy strategy, the restored memory page is still a shadow page; the original page frame is left in the kernel address space. Compacto does not have to change the state of the restored page into the write-protected mode or to catch a page fault. Since the copy strategy reuses a shadow page, it causes a smaller number of page faults than the remap strategy. The amount of memory copying depends on the pattern of memory write access until the next issue of the `restore_state` system call. If the process writes in the same set of memory pages, that is, only the shadow pages, the copy strategy needs only the same amount of memory copying as the remap strategy (Fig. 3.6 (a)(b1)). Since no page fault occurs in this case, the copy strategy is faster than the remap strategy. On the other hand, if the process writes in a totally different set of memory pages, maintaining shadow pages is meaningless and hence the copy strategy needs a larger amount of memory copying. Compacto must copy the saved images to all the shadow pages in the `restore_state` system call (Fig. 3.6 (a)) and duplicate other pages when page faults occur (Fig. 3.6 (b2)).

If the copy strategy is selected, the `restore_state` system call restores only the memory pages whose dirty bit is set. The dirty bit is set by the hardware if a process writes in the memory page associated with that dirty bit. After the restoration, all the dirty bits are reset so that Compacto can determine whether the page is written in between this and the next restoration. The `restore_state` system call does not copy memory pages whose dirty bit is clear.

Since the copy strategy always has to maintain both the original page frames and the shadow pages, it always needs more memory pages than the remap strategy. To reduce memory consumption, Compacto can discard the shadow page that has not been written in for a long time. Compacto examines how long a page has not been used by a change history of the dirty bit of each page. The

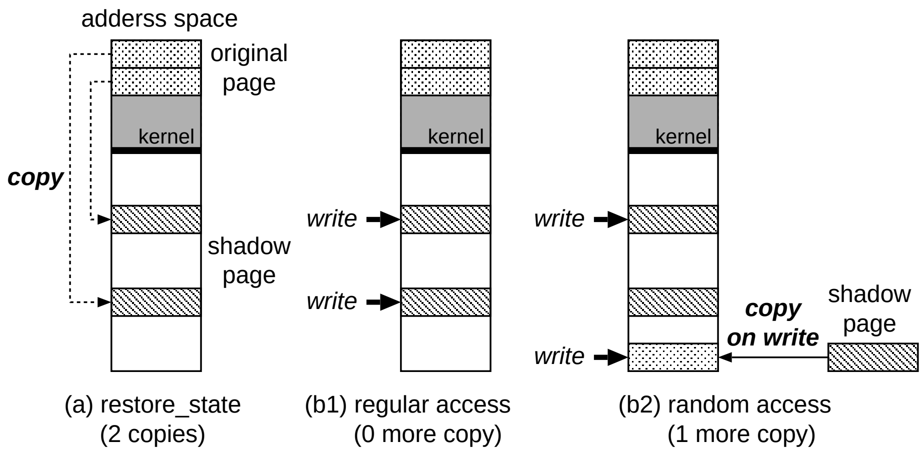


Figure 3.6: The performance of the copy strategy depending on a pattern of memory write access. After the `restore_state` system call is issued (a), if the access pattern shows good locality (b1), the copy strategy is faster. Otherwise (b2), it is slower.

`restore_state` system call unmaps and discards an unused shadow page and moves the original page frame back from the kernel address space. The original page frame is made write-protected to detect the next write.

3.3.3 Optimizing Server Program Layout

The overheads due to process cleaning can be reduced if a server program is statically linked with all necessary libraries such as the standard C library. If the libraries are dynamically linked, the static data used by them are allocated in distinct segments. As a result, the number of memory pages modified between `save_state` and `restore_state` increases and then the cost of restoring a memory image increases. On the other hand, if the libraries are statically linked, all the static data are allocated in the same segment as the static data segment of the server program. This reduces the number of memory pages allocated for the static data and hence it reduces the number of memory pages that must be restored when the `restore_state` system call is issued. Statically-linked binaries are simply created by the `-static` option of `gcc`.

Furthermore, the developer of a server program can use a tool that we developed to further reduce the number of memory pages restored by the `restore_state` system call. This tool relocates the layout of a static data segment by using a runtime profile so that the locality of the memory write access is increased (Fig. 3.7). Using this layout optimization, the number of memory pages restored by the `restore_state` system call is minimized. This relocation might also improve the cache hit ratio to the static data.

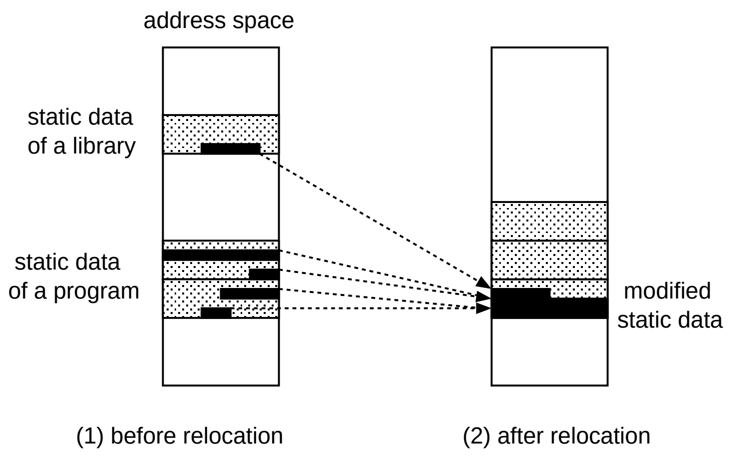


Figure 3.7: Memory layout before/after relocation of a static data segment.

This tool first embeds the code for recording memory writes in a server program. This tool can accept i386 assembly code emitted by `gcc` as an input and inserts a logging function as follows before instructions performing memory writes such as `mov` and `push`.

```
pusha                 # save all registers
pushf                 # save CPU flags
pushl $6              # written memory size (2nd argument)
leal 42(%esp),%eax    # written memory address (1st argument)
pushl %eax
call wprof_rec_write  # logging function
addl $8,%esp
popf                  # restore CPU flags
popa                  # restore all registers
```

In addition, this tool inserts such a logging function after system calls that write data in buffer passed from a user process, such as `accept` and `read`. Then, this tool links a module for a logging function with the server program and runs the modified server program to record a profile.

Based on the profile, this tool relocates the layout of the static data segment of the server program. Regularly, initialized static data is located in the `.data` segment and uninitialized static data is located in the `.bss` segment. This tool creates a new segment `.data0` and relocates both initialized and uninitialized static data modified during profiling in the new segment. Since each segment is located in a contiguous memory area, modified static data is collected in less memory pages.

3.4 Implementation of Access Restriction

3.4.1 Security Policy Rules

To enable to describe robust policy rules on top of the operating system based on UNIX, there are several problems. One problem for describing policy rules is that a path name is not unique in UNIX. For one file, there exists multiple names using symbolic links, hard links, and relative path names with "." and "..". To prevent an attacker from bypassing security policies by changing path names using such aliases, Compacto uses canonical path names to examine security policies. To get a canonical path name, Compacto normalizes all path names emerging both in the policy rules and the parameters of system calls. Compacto tracks symbolic links and transforms a relative path name to an absolute path name by complementing the current directory. For hard links, Compacto treats them as it is since tracking them is difficult and time consuming.

Expanded Process Tree

Compacto allows the users to write policy rules using the `via-` conditions such as `via-prog`. The `via-` condition makes Compacto apply a policy rule only if either the process or one of the ancestor processes satisfies the condition. To make the `via-` conditions work correctly, a process tree that reflects a parent-child relationship between all processes is needed. However, the process tree in UNIX does not maintain sufficient information. In UNIX, when a parent process terminates, the child processes become orphan processes. In other words, that child processes become direct children of the `init` process, which is the root of the process tree. In this process tree, information on ancestor processes is not available if a user executes some processes in background and then logs out. For example, as illustrated in Figure 3.8, when an attacker logs in using ssh, executes `httpd` in background, and then logs out, the fact that `httpd` is executed by the attacker is not left in the process tree in UNIX. In addition, when a process issues the `exec` system call, a program running in the process is replaced by another program and then information on the process before `exec` is lost. Except the case that a process duplicates itself using the `fork` system call before `exec`, information on the original process is not available.

To correctly maintain a relationship between a parent process and the child processes, Compacto expands the process tree that traditional UNIX maintains. The expanded process tree leaves information on a process as far as its child processes are running even if the parent process terminates. This makes information on all ancestor processes always available. Moreover, information on any ancestor processes is not lost even if some processes are replaced using the `exec` system call. To suppress the memory consumption for the expanded process tree, only the necessary information on a terminated process, such as the program name and the taint information, is left.

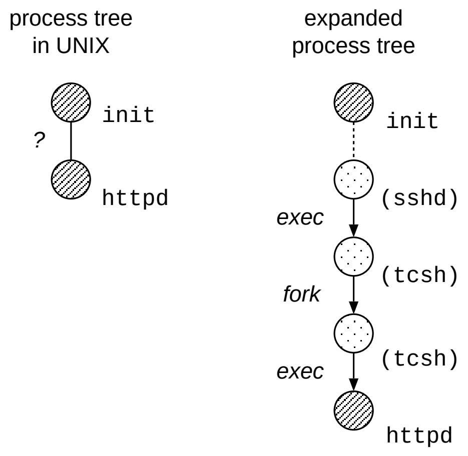


Figure 3.8: The process tree in UNIX and the expanded process tree. In the process tree in UNIX, the parent of `httpd` is `init` and the other information is not left. On the other hand, the expanded process tree shows that `httpd` was executed by a user that logged in using ssh from a remote host.

Program Identification

Using a program name to identify a program executable may allow malicious users to bypass security policies. If a malicious user copies a program executable and renames it, he can execute the program without any access restrictions. If an attacker replaces an executable with a Trojan horse version, he can make legitimate users execute the malicious program. To uniquely identify a program, Compacto allows using the SHA-1 [74] value of a program executable, which is a message digest value like MD5 [88], instead of a variable program name. This value is almost unique to a program executable.

At system boot time, the administrator registers a tuple of a program name and the calculated SHA-1 value to database in Compacto. When a program is executed, Compacto retrieves the corresponding SHA-1 value from the database and compare it with a SHA-1 value of a program executable specified by a program name. If these two values do not match, Compacto can make a decision that the program name is renamed to bypass security policies or the program executable is replaced to a Trojan horse version. A SHA-1 value is powerful for identifying a program, but its calculating cost is high. Therefore, the administrators can use this powerful program identification mechanism to only important programs.

3.4.2 Security Policy Checker

When a process issues a system call, Compacto calls a routine for checking security policies, which is referred to as security policy checker, before calling a routine for the specified system call. Our security policy checker makes the system call terminate with an error code `EPERM` if its execution is denied. The security policy checker examines both the security policy registered for the process and ones inherited from ancestor processes. If any security policies applied do not deny the execution of the specified system call, the execution is allowed; even if only one security policy denies it, its execution is denied. The security policy checker performs minimum validity checks of the parameters of a system call that is needed to check security policies. Extra validity checks of the parameters are performed in the body of the system call. For example, the security policy checker examines whether a file descriptor specified in the `write` system call is for a file or a socket but does not examine whether the specified file or socket is locked or not.

The security policy checker also examines security policies in the middle of the execution of some system calls. In the `accept` system call, it examines the source IP address and port number after a new connection is established and a new socket is created. In the `recvfrom` and `recvmsg` system calls, it examines the source IP address and port number when data is read. Also, the security policy checker records a canonical path name for files and directories in the `open` system call. It is used to examine security policies for system calls that take a file descriptor as a parameter.

For efficient policy checks, when a new security policy is registered to a process, Compacto compiles the security policy with already registered policies including the inherited policies. Using compiled policies minimize the overheads due to performing the security policy checker for every system call. In policy compilation, Compacto merges all security policies using the relation of inclusion among policy rules so that the number of policy rules to be checked is reduced. Compacto calculates the intersection of all *allow* rules and the union of all *deny* rules. For example, suppose that there are four rules as follows:

```
allow read "/usr/local/apache/*"
deny read "/usr/local/*"
allow read "/usr/local/apache/htdocs/*"
deny read "/usr/local/apache/*"
```

These rules are merged to two rules as the below:

```
allow read "/usr/local/apache/htdocs/*"
deny read "/usr/local/*"
```

3.4.3 Restrictor

Compacto provides system calls with which a process can impose access restrictions on itself at runtime. To make it easier to dynamically impose access restrictions, we introduce a *restrictor* as a policy unit. A restrictor is a kernel object and holds a set of access restrictions. A process can create a restrictor from policy rules and bind it to the process itself. Compacto does not provide a method of unbinding or destroying a restrictor for a security reason. Hence, the access restrictions by a restrictor can be removed only when process cleaning, proposed in Section 3.2.1, is performed. Since process cleaning restores the state of the process to that before the restrictor is bound, as a result, the restrictor is unbound and destroyed. A restrictor is somewhat similar to a capability but is different in that a restrictor consists of a set of access restrictions while a capability consists of a set of access permissions. A capability must be a subset of capabilities that the process has. On the other hand, a restrictor is freely created as far as it does not conflict the security policy that has been already applied to the process.

The `create_restrictor` system call creates a restrictor from the specified policy rules. This system call returns a restrictor ID, which is a unique random number, and the user has to pass this ID to other system calls related to restrictors. The `bind_subject` system call binds a restrictor to a subject. The subject is specified by a process or a process group in the current implementation. The access restrictions that the restrictor holds are applied to the subject. The `bind_object` system call binds a restrictor to an object. The object is specified by a resource or a resource group such as a descriptor for a file or a socket. The access restrictions that the restrictor holds are applied only to the object.

Delegation of Restrictors

Compacto supports delegation of restrictors in distributed systems. When a server communicates a client, the server process can request the restrictors of the client process so that the server can limit its activities depending on the clients. With the delegated client's restrictors, the server can handle requests from the client without any preliminary knowledge, for example, security policies for the client. This is somewhat similar to delegation of capabilities, but the clients do not need to worry about granting their capabilities like capability systems. The restrictors that the clients pass to the server do not contain any access permissions and are used only for client security. Even if the server misbehaves due to previous attacks, the client's restrictors bound to the server can minimize the damages of such attacks and can prevent the compromised server from accessing all of the client's data.

To delegate restrictors, the operating system of a client sends a restrictor group ID, which is an ID for a set of restrictors, to a server using an IP option field of a packet for a request. If the server process needs client's restrictors, the operating system of the server process requests the instance of client's restrictors of the client using Remote Procedure Call (RPC). That instance is indicated by the restrictor group ID sent from the client. The operating system of the client replies all the policy rules included in the specified restrictor group. The operating system of the server creates a restrictor from that policy rules replied and binds it to the server process. Figure 3.9 shows this algorithm.

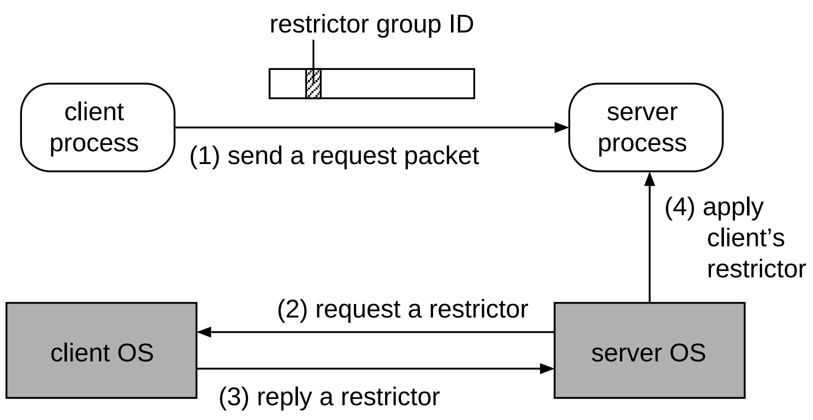


Figure 3.9: The delegation of client's restrictors.

After that, requests from the client are handled under both the server's original access restrictions and newly bound client's access restrictions. When the server process finishes to handle requests from the client, it removes the client's access restrictions by process cleaning.

However, this delegation mechanism has a limitation. Client's restrictors are not effective in the other administrative domains since they hold security policies in the client host. For example, if a client does not have knowledge about a server system such as the operating system and the directory construction, it cannot define policy rules for the server. To propagate the access restrictions of a client beyond the administrative domain, Compacto needs the process trace facility as described in Section 3.1.2.

3.5 Implementation of Process Trace

3.5.1 Dependencies between Processes

The dependency between processes is a relationship that arises when a process affects or is affected by another process. The dependencies between processes consist of communication dependency between processes and parent-child dependency between a process and the child processes. Communication dependency arises when two processes communicates with each other. Communication between processes is data exchange via operating system resources such as socket, pipe, and file. Since a sender process affects a receiver process by sending data, dependency from the receiver to the sender arises. On the other hand, parent-child dependency arises when a parent process creates a child process. We define creating a child process as not only duplicating a process by the `fork` system call but executing a new program by the `exec` system call. Since a parent process dominates creating a child process, dependency from the child

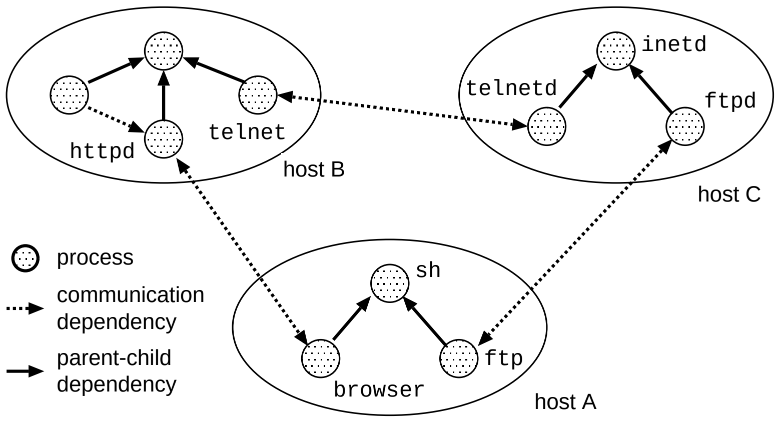


Figure 3.10: An example of dependencies between processes.

to the parent arises. Figure 3.10 shows an example of dependencies between processes over some hosts.

To trace an impact that a process makes on other processes, both communication dependency and parent-child dependency are indispensable. While network communication and inter-process communication are traced, communication via a regular file should be also considered. Suppose that an attacker modifies a configuration file of a server. If the server reads the file, the server may misbehave. This means that the impact made on a file by the attacker can reach to the server. On the other hand, parent-child dependency is needed to trace the execution flow of processes while it is also needed to trace an impact together with communication dependency. For example, even if a web server executes a CGI program as a child process and the CGI program communicates with another server, the impact from the web server to another server is traced correctly.

3.5.2 Problems in Dependency Graph

Dependencies between processes form a directed graph, whose node stands for a process and whose edge stands for dependency between two processes. An access control mechanism can trace the directed edges from a node corresponding to a target process and thereby find all the processes that affect the target process.

However, dealing with a dependency graph for access control involves some problems. First, an access control mechanism must maintain and manage a large amount of dependency data. The dependency graph includes even nodes related to terminated processes and edges related to communication in the past so that the mechanism can correctly trace the impact. Therefore, in large distributed systems, the dependency graph may grow largely. In addition, whenever new dependency arises from communication between two hosts, one host must send to the other host information on all dependencies that are traceable from the new dependency. If the dependency graph gets more complex, the amount of data exchanged gets more.

Second, examining the dependency graph at runtime can involve great performance penalty. When an access control mechanism attempts to restrict the access permissions of a process, it must trace the dependency graph and then examine all the processes that affect the target process. After that, it can allow or deny the access.

Third, all of the information included in the dependency graph do not need for access control. The dependency graph shows a detailed route of how an attacker has intruded to the system. However, from the viewpoint of defense from attacks, it is sufficient for access control to find a host with the lowest confidence from hosts that the attacker has intruded. That host is often an attacker's base host. For example, when an external client attacks an internal server via some internal hosts, the server can prevent the attack if it can recognize that the client resides in an external host.

3.5.3 Taint Mechanism

To compress a dependency graph as it is suitable for access control, we introduce a new concept called *taint* to a process and each resource. Taint consists of information on accumulated impacts made on a process by the other processes. An impact is accumulated when a process receives data from the other processes and when a process is created by the parent process. Taint is an index for risks of how extent a process is attacked from the other processes to. The extent of taint is determined depending on the confidence of hosts in which processes that affect the target process are running. If a process running in a host with lower confidence directly or indirectly affects a target process, the extent of taint of the target process gets severer. We say "a process is tainted" when a process is affected by another process with severer taint.

A taint level indicates the extent of taint and a higher value means be tainted more severely. The minimal taint level means that a process or a resource is not tainted at all. On the other hand, the maximal taint level means that a process or a resource is tainted most severely. In other words, this means that the process or the resource has been affected by a process with the maximal taint level, for example, from an untrusted host outside the local network. The middle taint levels are assigned corresponding to the confidence of hosts that affect a process or a resource. Since the confidence of hosts is often hierarchical in a distributed system, my host is considered as the most trustable host and hosts that belong to larger groups are considered as more untrustable hosts. For example, you can assign higher confidence to hosts within a department that your host belongs to and lower confidence to hosts outside the department as shown in Figure 3.11. As such, the user can assign different confidence to even hosts within the same local network and then prevent attacks by insiders.

Taint information is recorded in the corresponding kernel entities when a process, a pipe, a file, and a socket are tainted. Exceptionally, the taint infor-

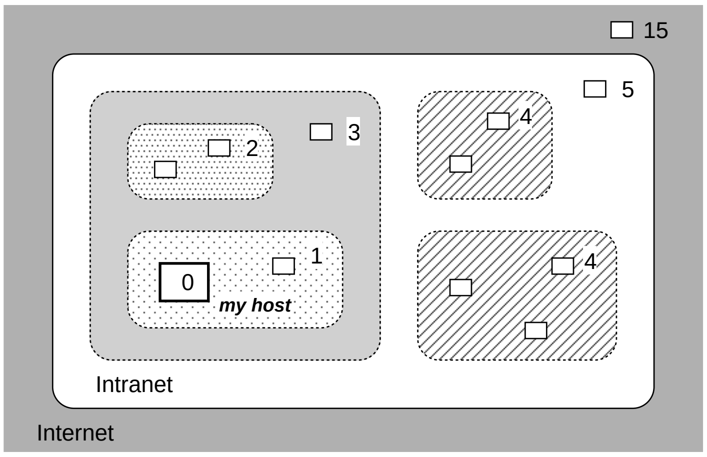


Figure 3.11: A mapping of a taint level to hosts from my host's point of view, according to the hierarchy of the organization.

mation for a file and a directory is also written in disk storage. Compacto stores the taint information in an unused area of a disk inode in the Linux ext2 file system when an inode is written back to disk. Additionally, the owner of a file can set to the file the allowable maximal taint level where a process is allowed to access the file. For example, if an allowable maximal taint level is set to 0, only processes with a taint level 0 can access it. This allowable maximal taint level is also stored in disk storage. Using this enforcement by the file system, Compacto can restrict file accesses by processes without any explicit policy rules.

3.5.4 Taint Propagation

To trace an impact that a process makes on surrounding processes, Compacto makes the taint information propagate to an affecting process. Taint information is propagated based on dependencies between processes, which are communication dependency and parent-child dependency. In communication between two processes, taint information is propagated via operating system resources such as a socket, a pipe, and a file. If a process reads data from a resource, the process is tainted to the taint level that the resource has. On the other hand, a process writes data to a resource, the resource is tainted to the taint level that the process has. Note that a higher one of either the current taint level or the propagated taint level is dominant and the taint level of a process and a resource never drops.

To propagate taint between different hosts, one host must notify taint information of the other host via network. To minimize this extra communication overhead, Compacto piggybacks taint information on a packet used for data transfer using its IP option field. The increase of a packet size is 4 bytes, which

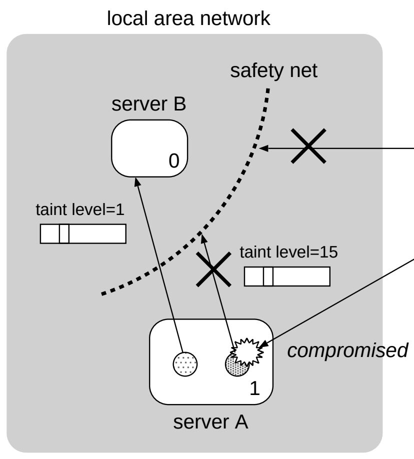


Figure 3.12: Prevention of an indirect attack using taint information. The lower right number of each host stands for the taint level corresponding to the confidence of a host from the viewpoint of server B.

consist of an option type of 1 byte, an option size of 1 byte, and taint information of 2 bytes. The taint level of a packet is relative to a process that handles the packet since it depends on the confidence of a sender host from the viewpoint of a receiver host. The taint level of a packet is a higher one of either the taint level piggybacked by the packet or the taint level determined by the confidence of a sender host. Taint information is propagated only within a local network since Compacto does not trust an IP option field of packets from external hosts. For packets from external hosts, Compacto always regards the taint level as maximum. In addition, Compacto prohibits using raw socket for processes with a higher taint level and thereby prevents tainted processes from forging a taint level in an IP option field.

Using taint information, Compacto can prevent attacks via insecure hosts within the local network. Consider the example of Figure 3.1 described in Section 3.1.2 again. We show it in Figure 3.12 again. Server A and server B belong to the same network and trusts each other. Client Z is outside the network and is not trusted by server A and server B. If a process in server A is compromised by an attacker in client Z, the taint level of the A's process changes the maximal value, for example, 15. Then, if the A's process attacks a process of server B, the taint level of the A's process is propagated to server B. Based on the propagated taint level, the server B can restrict accesses from the attacker in client Z via server A.

To use a taint level for access control, Compacto provides the `at-tlevel` and `of-tlevel` conditions. Policy rules are applied only if the taint level of a process or a resource satisfies these conditions. The `at-tlevel` condition is satisfied only when the taint level of a process is within the specified range. This is useful for preventing a compromised process from accessing sensitive data. The `of-tlevel` condition is satisfied only when the taint level of a resource is within the specified range. This is useful for denying network connections from a process with a higher taint level.

3.5.5 Cleaning Taint Up

In practical servers, using taint information for access control may make the access restrictions of a server too strict. Since Compacto does not allow a process to lower the taint level, the server may not be able to continue to provide their service due to the too strict access restrictions. For example, a web server that has been accessed from an external user is tainted to the maximal taint level and thereafter may deny access requests to private information from Intranet users. To prevent such situation, Compacto cleans up the taint of a process using process cleaning, described in Section 3.2.1. Process cleaning also recovers the taint of a process as well as the other state of the process. Even if a server process is tainted, its taint level is recovered to the value saved before when the server finishes handling requests from a client and then issues the `restore_state` system call.

3.6 Implementation of Network-Level Authentication

To authenticate remote users at the network level, Compacto supports Secure Socket Layer (SSL) [42] at the kernel level. With SSL, Compacto can authenticate a remote user using a user's X.509 certificate [43], which includes user's public key and is signed by a certificate authority (CA). This kernel-level SSL is implemented using the OpenSSL library [76], which is the most popular user-level SSL library. We modified OpenSSL so that it can treat the socket interface in the kernel. The kernel-level SSL can prevent attackers from bypassing of authentication. Figure 3.13 shows such safe communications. To prevent attackers from impersonating a legitimate user in a client host and succeeding in authentication with a server, the operating system of the client prohibits using this authentication if the taint level is high. In the current implementation, Compacto supports TCP/IP communication due to the limitation of the OpenSSL implementation.

The kernel-level support of SSL also enables the operating system to reuse SSL sessions between two hosts for each user. Applications using a user-level SSL library can reuse SSL sessions between two hosts only for each application. For example, even when a user in one host uses a web server, a telnet server, a mail server, and so on, in the other host, she must perform complete negotiation including the verification of her certificate for each server. With the kernel-level SSL, Compacto needs the complete negotiation only when a user establishes the first connection between two hosts. After that, she can perform simplified negotiation using secrets exchanged in the first negotiation. The cost

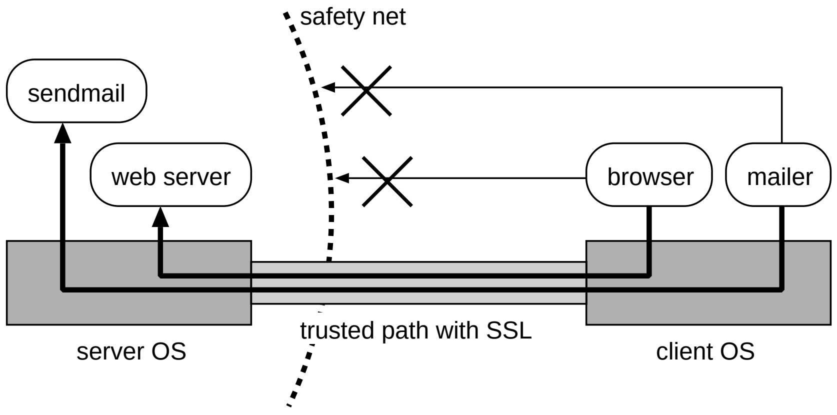


Figure 3.13: Safe communications with an untrusted client using SSL. The web browser and the mailer can communicate with the web server and sendmail through the trusted path established by SSL, respectively.

of the simplified negotiation is much lower than that of the complete negotiation. To avoid degradation of system security, Compacto performs the complete negotiation again if a certain time is expired.

The algorithm with which Compacto authenticates client users is as follows. When a client establishes a TCP/IP connection with a server, the operating system of the client starts a SSL negotiation in the `connect` system call. If the client receives the server certificate, it checks the certificate to prevent server impersonation and then sends the client's certificate to the server. The operating system of the server checks if the received user certificate is properly signed by a CA for the server and if the user is registered to the system. If this authentication succeeds, the client is allowed to access the server; otherwise, the connection between the server and the client is shut down.

After the authentication, the server can use the authentication information for its access restrictions. The user certificate is mapped to the corresponding user name when it is registered to the system. By the `by-auth-user` condition, the server can describe policy rules that are applied to only specific remote users. Also, this authentication is used to suppress that the server is tainted by the untrusted remote host. Even if the remote client resides a host with low confidence, the server is not tainted if it succeeds in the user authentication. The client user is treated as if she were within the local network as far as the client and the server are connected with the trusted path with SSL.

User Interfaces

Compacto provides two methods so that the applications can specify to use our kernel-level SSL for their communication. One method is to specify communications that uses the kernel-level SSL indirectly by the proc file system in Linux. The proc file system has a directory for each process and the owner of the process can read and modify some state of a process through the file system interface. We added a virtual file `ssllist` to the directory. This file consists of the IP addresses of hosts with which the corresponding process communicates using the kernel-level SSL. When the process attempts to communicate with one of these hosts, Compacto automatically uses the kernel-level SSL. Programs do not need to be modified to use the kernel-level SSL. In addition, the contents of this virtual file are inherited to the child processes. The other method is to set a special socket option in programs although the programs need to be modified a bit. Compacto provides the `SO_USE_SSL` socket option and allows sockets to which this option is set to use the kernel-level SSL.

Compacto also provides a new system call `certctl` for dealing with user certificates. This system call registers a user certificate, a user's private key, the host certificate, the host's private key, a server certificate, a server's private key, and the CA's certificate to the operating system. A user certificate and a user's private key registered in advance are used when the corresponding user starts SSL. Once a user registers this user-specific information, they are available for all processes that the user owns. The host certificate and the host's private key are used as a certificate and a private key for all servers in the host by default. If some servers would like to use other certificates and private keys, they can register their own certificates and private keys to sockets they use. In this case, server programs needs to be modified so that they register server-specific certificates and private keys by using `certctl`. Also, the CA's certificate is used for verifying user certificates.

3.7 Summary

This chapter presented process cleaning, which allows a server process to securely remove the access restrictions in case that the process has been compromised. To use process cleaning, Compacto saves the state of the process in advance and restores the saved state before removing the access restrictions from the process. With this cleaning-up procedure, the thread of control is recovered and the malicious code is eliminated from the memory even if attackers have hijacked the process or injected Trojan horse code into the process. To prevent attacks using the other resources a process has, Compacto saves and restores the signal handlers, the status of open files and sockets, and so on, as well as the thread of control and the memory image.

Since the performance bottleneck of process cleaning is to save and restore the memory image of a process, Compacto uses various techniques, some of which have been already proposed in other research fields. To reduce the amount of memory to be saved, Compacto uses copy-on-write, which copies only modified memory pages. Also, Compacto allows the users to select a memory restoration strategy depending on the memory access pattern of a server process. In addition to optimization of process cleaning itself, Compacto provides a tool for optimizing the data layout of server programs in order to reduce the number of modified memory pages.

This chapter also presented novel features of access control. Compacto can use information on the activities of a client process. The activities are specified by communication dependency between processes and parent-child dependency between a parent process and child processes. Since information on these dependencies can enlarge over time, Compacto compresses the dependencies to a taint level using our taint mechanism so that information on the most dangerous host with which the client process interacts directly or indirectly is left.

Also, Compacto authenticates remote users at the network level so that trusted external users can access internal servers under suitable access restrictions. This authentication is performed in a manner where applications on top of Compacto are not aware. For this authentication, Compacto uses the SSL implemented in the kernel. A remote user is authenticated with a user certificate, which is digitally signed by a certificate authority for a server that the remote user attempts to access. Compacto can impose access restrictions on the server depending on the authentication information instead of the owner's user ID of the server.

Chapter 4

Fail-Safe Mechanism for Operating System Modules

We have developed the CAPELA operating system, which provides a new fail-safe mechanism for operating system modules. In this chapter, we present our fail-safe mechanism called *multi-level protection* and the implementation. We refer to a module to extend the operating system for performance and functionality as an *extension module.*

4.1 Multi-Level Protection

We propose a new fail-safe mechanism called *multi-level protection* [56, 55], which enables the users to change the protection level of extension modules without modifying the binary code. Using the multi-level protection, the users can consider the trade-off between fail-safety and performance. The maximum protection level achieves sufficient fail-safety, but the minimal protection level does good performance. Figure 4.1 shows this concept roughly. For example, if an extension module is unstable, the users can use a complete fail-safe mechanism, sacrificing the performance. On the other hand, if an extension module is stable, the users can use a simplified fail-safe mechanism and then improve the performance.

To change the protection level of extension modules, the multi-level protection changes the ability to detect errors and the ability to recover from errors. Some errors may not be detected to decrease the ability of detection. Also, some errors may neither be prepared for recovery nor be recovered from to decrease the ability of recovery. For instance, the multi-level protection can allow illegal memory reads to reduce the overheads of the detection. It can also record no logs for recovery in order to reduce the overheads of the records. Needless to say, the protection level is not always linear. The protection level of user-level modules is obviously higher than that of kernel-level modules. But we cannot say which is higher of the protection level where illegal memory accesses are

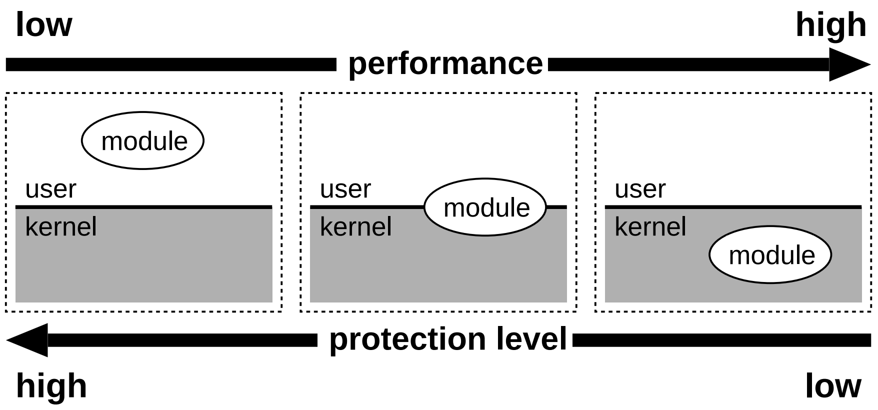


Figure 4.1: Making a trade-off between fail-safety and performance of extension modules by the multi-level protection.

detectable and the protection level where deadlocks are detectable.

To change the protection level without modifying the binary code of the extension modules, the multi-level protection provides an API to which the extension modules should conform. This API hides the differences between the implementation of fail-safe mechanisms, which are derived from the differences between the abilities of error detection and recovery. If the programmers do not conform to this API, for example, directly using privileged instructions or system calls for normal user processes, this good facility of modifying no binary code at changing the protection level is lost. It is responsible for the programmers whether they conform to this API. Also, the API exports the kernel data structure of high abstraction to the extension modules, and therefore the programmers are easy to extend the operating system although the extensibility is not very high.

Extension Modules by Third-Party Vendors

The multi-level protection can make the users easy to use extension modules provided by third-party vendors. Such modules may be unstable and crash every few days since the third-party vendors may misunderstand the specification of the modules and may not test the modules sufficiently. They include device drivers of minor devices like a CD-ROM changer and some third-party file systems like NT file system (NTFS) for PC UNIX, which the operating system vendors officially do not support. Although these unstable modules may frequently crash, the crash is acceptable if users seriously want to use them at any cost. However, the rest of the operating system should be kept stable even at that time.

The multi-level protection makes it possible to safely run these unstable modules. The fail-safe mechanism protects the modules sufficiently, and safely detaches the modules from the operating system if the modules crash. Therefore the rest of the operating system is not affected by erroneous extension modules. On the other hand, many modules supplied by the third-party vendors are stable and do not need the fail-safe mechanism. The system should run stable modules without any protection and eliminate the performance penalties. If there is still a possibility that the modules involve errors, the system should run the modules at a lower protection level and more efficiently.

Easy Development

The multi-level protection also makes the users easy to develop extension modules. So far, in many operating systems, the extension modules like file systems have been implemented directly in the kernel, or have been re-implemented in the kernel after they were developed as user-level libraries that emulates system calls, and so on, for debugging. In the former method, the developers are hard to debug the extension modules while the finished modules are very efficient. On the other hand, in the latter method, the developers must write the extension modules twice both for the emulation libraries and for the kernel modules while debugging at the user level is easy.

This means that only a single protection level is not enough to make the extension modules easier to develop. The protection level of the extension modules should be changed during the development since the kinds and frequencies of errors depend on the stability of the extension modules. For instance, the extension modules include many errors at the beginning of the debug phase, but they are getting stable.

The multi-level protection provides appropriate fail-safety in each development phase below. In the debug phase, the fail-safe mechanism keeps the full capability of the protection even though this involves the maximum performance penalties. The errors are detected immediately and the accurate information of the errors is reported to programmers. These features considerably help programmers identify the reason of the errors and fix them. Moreover, the fail-safe mechanism can safely terminate the erroneous modules after they crash. Due to sufficient fail-safety, programmers can make rapid prototyping of the extension modules.

In the beta-test phase, the fail-safe mechanism does not must keep the full capability of the protection. Rather it should run the extension modules as fast as possible so that the test users are satisfied with the performance to some degree. If the extension modules achieve better performance, more test users would use them and find more errors. Since the extension modules are expected to be fairly stable in this phase, only relaxed protection is needed. For example, it has only to detect and recover from a few kinds of errors depending on timing such as deadlocks. Illegal memory accesses like an access of null pointer are expected not to frequently occur.

Finally, the extension modules released as a product need the fail-safe mechanism no longer. Unnecessary protection is removed and they can run as almost efficiently as one directly implemented in the kernel by hand.

4.2 Overview of the CAPELA Operating System

We have implemented the CAPELA operating system on the basis of NetBSD [72] 1.3.2. CAPELA is an extensible operating system with the multi-level protection. CAPELA is running at Intel and SPARC platforms. But since the platform dependent part is a few, CAPELA can be easily ported to the other platforms that NetBSD 1.3.2 supports.

In the CAPELA operating system, the programmers can create the extension modules as programs independent from the kernel in order to extend the functions of the operating system. The extension modules are driven by events hooked in the operating system kernel and achieve the functions of new subsystems by communicating with the kernel. The communication is done through a *protection manager* provided by CAPELA. A protection manager is provided per extension module and plays a role of a gateway between the extension module and the kernel. The protection manager cooperates the kernel and thereby provides a fail-safe mechanism so that the extension module can manipulate the kernel data safely. CAPELA provides multiple protection managers, each of which provides a different protection level to the extension module. Using these protection managers, CAPELA achieves the multi-level protection.

Figure 4.2 depicts the overview of the CAPELA operating system. The implementation of the protection managers for the extension modules located at the user level is largely different from that for the extension modules located at the kernel level. If an extension module is located at the user level, its protection manager uses shared memory for communication with the kernel. The kernel uses an upcall mechanism to invoke the entry points of the extension module when hooked events occur. On the other hand, if an extension module is located at the kernel level, the communication between its protection manager and the kernel is done through the kernel memory. The invocation from the kernel to the extension module is a direct function call.

Changing a Protection Level

CAPELA allows the users to change the protection level of an extension module by exchanging its protection managers. Each protection manager that CAPELA provides can detect and recover from a different kind of error. Depending on the stability of the extension module, the users can select the protection manager that provides an appropriate protection level at that time from all protection managers.

The protection managers are implemented as user-level libraries or kernel libraries. The user-level libraries are used when extension modules are running in the user address space whereas kernel libraries are used when extension modules are running in the kernel address space. When the users change the protection

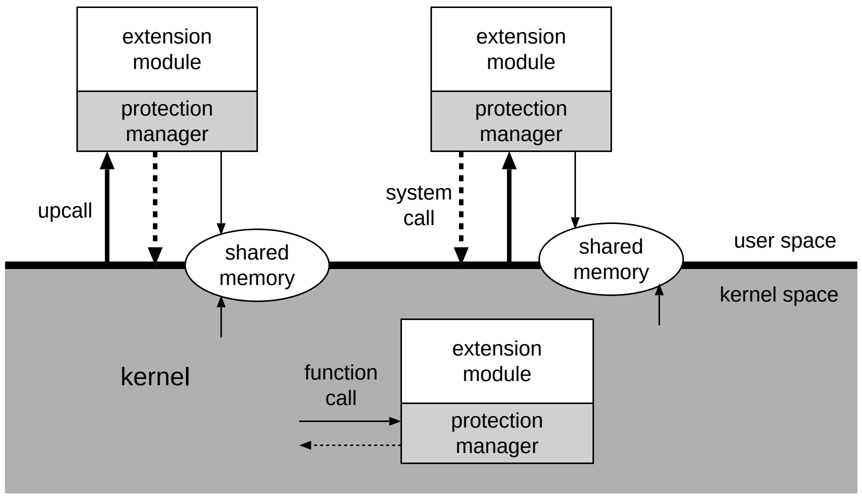


Figure 4.2: The relationship among extension modules, protection managers, and the kernel.

level of an extension module, they need to unlink an old protection manager from the extension module and relink a new protection manager with it. When the users change the protection level so that the extension module continues to run in the same address space, they can simply restart it after relinking. On the other hand, when the users change the protection level so that the extension module moves between the user space and the kernel space, they need extra tasks. If the extension module moves from the user space to the kernel space, the users stop the running extension module and dynamically link it with the kernel using the Loadable Kernel Module (LKM) mechanism. If the extension module moves from the kernel space and the user space, the users unlink the extension module from the kernel and start it as a user-level process. In any case, it is unnecessary to recompile the extension module. However, if recompiling the extension module is allowable, it can make the performance of the extension module better, in particular, when the protection level gets the lowest.

4.3 Implementation of the Protection Manager

The protection manager has three kinds of responsibilities to an extension module. First of all, the protection manager plays a role of a gateway between the kernel and the extension module. If the protection manager receives upcalls from the kernel, it invokes callback functions of the extension module corresponding to the upcalls. Conversely, when the extension module needs to access the kernel functions or the kernel data, the protection manager does that instead. To play this role, the protection manager provides an API to the extension modules.

Second, the protection manager protects the kernel from errors of the extension module. Depending on the protection level it provides, the protection manager protects the memory for the kernel data or replicates the kernel data. Third, the protection manager registers and unregisters the extension module to the kernel. In particular, it safely detaches the extension module from the kernel even if the module terminates abnormally.

4.3.1 API

The protection manager provides an API to which all the extension modules must conform so that the extension modules can interact with the kernel. Since all the protection managers provide the same API, the protection levels of the extension modules can be changed without modifying the extension modules. This API consists of callback functions and manipulation functions for the kernel data. Although forcing all the modules to conform to this API may restrict the programming of the extension modules, that is, extensibility, the advantages of changing the protection level without modification of the extension modules will outweigh the disadvantages of this restriction. The details of this API are described in Appendix C.

Callback Functions

The protection manager provides an API for callback functions invoked by upcalls from the kernel. When an event hooked by an extension module occurs in the kernel, the kernel first notifies the protection manager of the event using an upcall. Second, the protection manager that received the upcall translates the parameters from data structure of low abstraction used in the kernel to one of high abstraction used in the extension modules. Finally, the protection manager invokes a callback function of the extension module.

To define callback functions specific to an extension module, the programmers can override methods of classes for callback. Since CAPELA provides C++ classes for callback, the programmers can inherit them to create their own classes for callback. For example, `FileSystem` class is a class for callback on file systems. To develop a new file system, the programmers can inherit the class and override some methods such as `mount()` and `read()` if necessary.

Manipulation functions for the Kernel Data

The protection manager provides an API to manipulate the kernel data because the extension modules cannot directly access the kernel data in CAPELA for two reasons. The one reason is that the protection manager prevents the kernel data from being illegally accessed. The kernel data is very complex since the execution efficiency is the most important and since pointers are used very frequently. For example, `mbuf`, which is a container to deal with variable data from network, has very complicated structure for both the generality and the execution efficiency. Also, shared memory on which the kernel data is put is often protected by the protection manager. In this case, the kernel data is accessed only after the protection manager removes the protection of the shared memory.

The other reason is that CAPELA makes the extension modules deal with data structure of high abstraction. The protection manager translates the kernel data structure of low abstraction to one of high abstraction in order to hide the complexity of the kernel data structure. For instance, the programmers of the extension modules can manipulate a chain of `mbuf`s using `MbufChain` class instead of manipulating multiple `mbuf`s with pointers. They can manipulate an instance of `MbufChain` class as if they were dealing with one data container. Also the API allows programmers only to increment and decrement a reference count one by one.

The protection manager also provides an API equivalent to one provided by the kernel since the extension modules running at the user level cannot directly use an API provided by the kernel. For example, an NFS [93] server needs to access a local file system like UFS. This means that an API to access a local file system in the kernel is needed if the the NFS server module is running at the user level. To enable a UDP [80] module and a TCP [82, 18] module at the user level to access an IP [81, 84] layer, an API for the operations to the IP layer is needed. The protection manager issues system calls or emulates these facilities to achieve them.

4.3.2 Protection Techniques

CAPELA uses some protection techniques to detect illegal memory accesses and deadlocks. Various combinations of these techniques enable the protection manager to provide various protection levels.

Switching Address Spaces

CAPELA uses the memory protection provided by switching address spaces to detect illegal memory accesses. The extension modules are located either in a user address space or in the kernel address space. If an extension module is located in a user address space, its illegal memory accesses to the kernel memory and the other processes' memory are trapped by hardware of the Memory Management Unit (MMU) as illustrated in Figure 4.3. When MMU detects such illegal memory accesses, CAPELA catches hardware traps from MMU and then can terminate the extension module. Additionally, if an extension module is running in a user space, CAPELA can prevent the module from exhausting resources like CPU and memory by the resource limitation mechanism for user processes such as the `setrlimit` system call. Moreover, CAPELA can use even the access control mechanism that we proposed in Chapter 3 to user-level extension modules.

However, the overheads for enabling this protection are rather large due to increasing the number of context switches and the amount of data copies between address spaces. For instance, two context switches and two copies of the

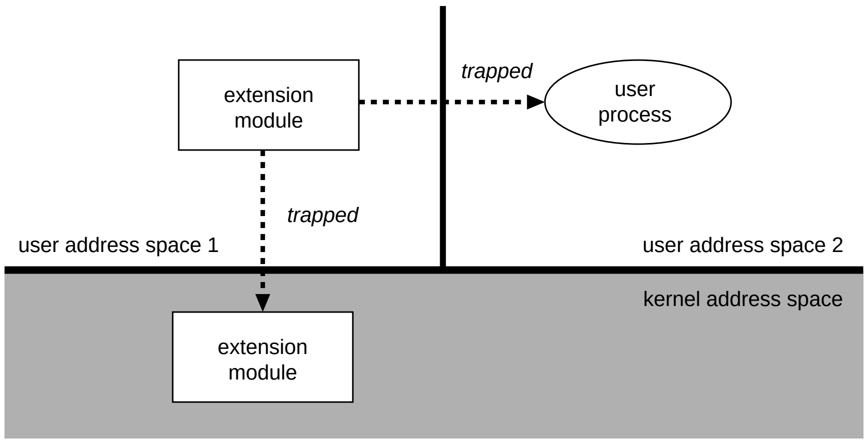
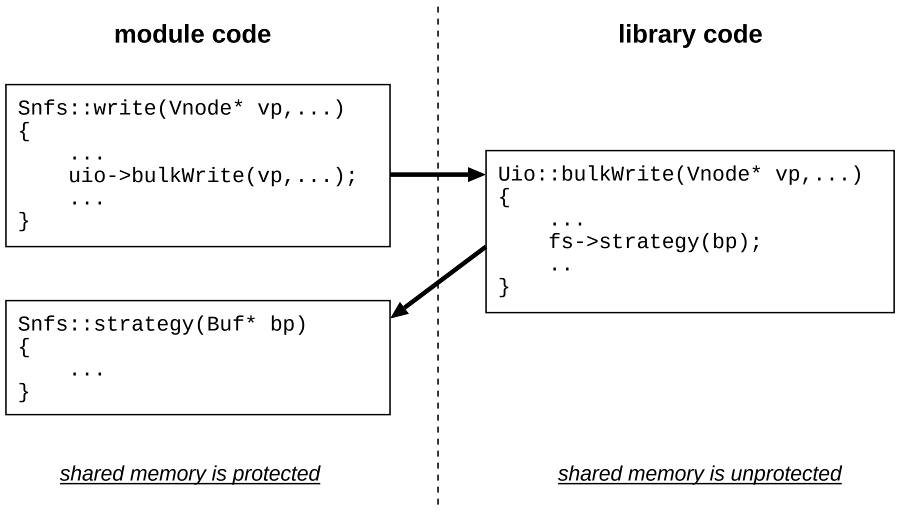


Figure 4.3: The memory protection provided by switching address spaces. If the extension module in the user address space 1 attempts to access either the extension module in the kernel or the user process in the user address space 2, a trap occurs.

data are needed at least to pass data between an extension module running in a user space and another user process. To decrease these overheads, CAPELA also allows the users to locate the extension module in the kernel address space. To embed an extension module into the kernel, the extension module is linked with the kernel dynamically using the Loadable Kernel Module (LKM) mechanism. In this case, only a context switch and a copy are needed at most between an extension module running in the kernel space and a user process.

When an extension module are located in a user address space, the module and the kernel use shared memory to communicate with each other. Since the important kernel data is also put on this shared memory so that the protection manager directly accesses it, corrupting the shared memory causes CAPELA to crash or to get unstable. To prevent the extension modules from illegally accessing the kernel data, CAPELA protects the shared memory using the virtual memory subsystem. While the code fragments of the extension modules are executed, the shared memory is protected by the kernel. On the other hand, while the code fragments of the protection manager are executed, the protection is removed so that the protection manager can access the kernel data on the shared memory. Needless to say, the kernel can always access the shared memory. In other words, the protection of the shared memory is removed when an extension module calls the functions of the protection manager and then the shared memory is protected again when the functions are exited. In contrast, the shared memory is protected when the protection manager calls a callback function of the extension module and then the protection is removed when the callback function is exited. Figure 4.4 depicts in which code fragments shared



```
Snfs::write(Vnode* vp,...)
{
    ...
    uio->bulkWrite(vp,...);
    ...
}
```

```
Uio::bulkWrite(Vnode* vp,...)
{
    ...
    fs->strategy(bp);
    ..
}
```

```
Snfs::strategy(Buf* bp)
{
    ...
}
```

Figure 4.4: An example of changing the protection of shared memory according to the function calls. When the `Snfs::write` method of an extension module is invoked by the kernel, the shared memory is protected. Then if the method calls `Uio::bulkWrite` provided by the protection manager, the protection of the shared memory is removed. While `Uio::bulkWrite` invokes `Snfs::strategy` of the extension module, the shared memory is protected again.

memory is protected.

In addition, while code fragments of the extension modules are running, CAPELA allows the protection manager to select how the shared memory is protected. The strongest protection is that the protection manager unmaps the shared memory. This prevents the kernel data on the shared memory from being accidentally corrupted or read. Trapping illegal reads to the kernel data helps errors to be detected earlier before that reads that do not directly affect the system cause more serious errors. Weaker protection is that the protection manager changes the protection of the shared memory to read-only. This prevents the kernel data only from being accidentally modified. In the SPARC architecture, since changing the protection to read-only is faster than unmapping the memory, this level of protection is useful because it sacrifices the ability of detection but gets better performance instead. In the Intel architecture, however, the overheads of these two protection are almost the same, this protection is not useful. The weakest protection is that the protection manager does not protect the shared memory. This leads good performance but sacrifices the protection completely. Figure 4.5 illustrates three kinds of protections of shared memory.

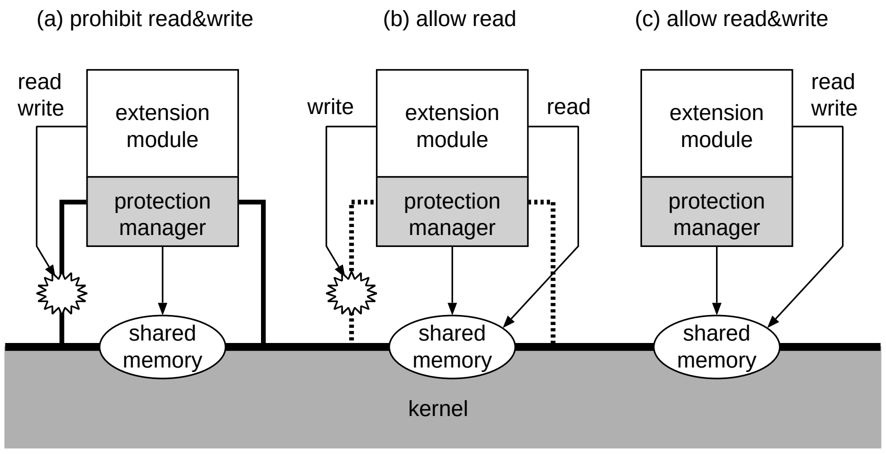


Figure 4.5: Three kinds of protections of shared memory. (a) The extension module cannot directly read and write the shared memory since the shared memory is unmapped. (b) The extension module can only read the shared memory since the shared memory is mapped in a read-only mode. (c) The extension module can freely read and write the shared memory since the shared memory is mapped in a read-write mode.

Replicating the Kernel Data

The protection manager replicates the kernel data on shared memory and makes the extension module access the replicas on the user address space instead of the kernel data on the shared memory. Then the protection manager checks the contents of replicas when it writes them back to the shared memory as illustrated in Figure 4.6 so that it can prevent the extension module from accidentally corrupting the kernel data on the shared memory. For example, if some data fields of an object are corrupted by an buffer overflow of an array in the object, the protection manager can detect that destruction since it checks all data fields of the replica object when it writes them back. If the shared memory is protected by the memory protection provided by address spaces, the extension module cannot corrupt the kernel data on the shared memory directly.

However, even when the shared memory is protected, the kernel data on the shared memory may be corrupted during executing an API that the protection manager provides. For example, if the programmers pass a pointer of a `Mount` object to an API function when they should pass a pointer of a `Vnode` object, the `Mount` object may be treated as the `Vnode` object and the data fields may be corrupted. In the C++ language, which is used for writing the extension modules, using a cast makes that destruction possible since type checking is loose. If the shared memory is protected, replicas are also used for a performance reason. Replicas enable an extension module to directly access the methods of

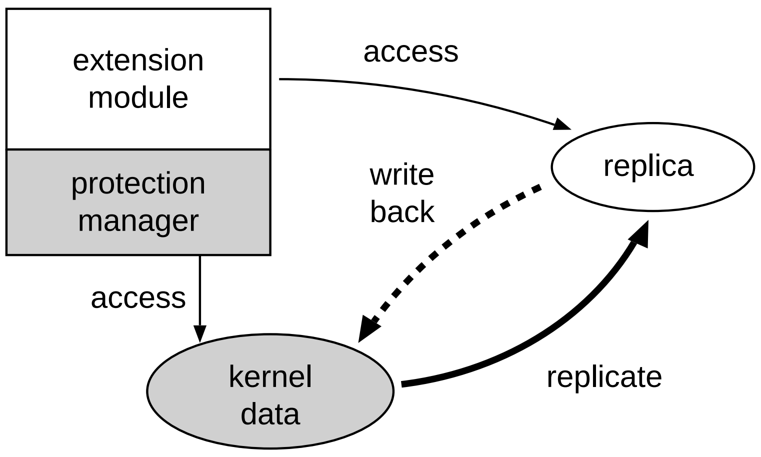


Figure 4.6: The replication and write-back of the kernel data. The protection manager directly accesses the kernel data while the extension module indirectly accesses the kernel data using its replica.

kernel objects. If replicas are not used, the extension module must remove the protection of the shared memory whenever it calls the methods of kernel objects on the shared memory.

To check the replicas, the protection manager uses various knowledges on the kernel data structure that it supports. For example, the knowledges are: that the reference counts are 0 or positive, that the size of each buffer is often limited by the minimum and the maximum, and that some data is put on the shared memory necessarily.

CAPELA allows the protection manager to select what types of kernel data is checked and how the kernel data is checked. For example, if users give up checking for any loops by pointers, the protection manager does not need to traverse pointers and then the overheads of the traverse are reduced although it is possible that the errors are detected lately or cannot be detected.

Wait-for-graph

CAPELA creates a wait-for-graph to detect deadlocks among extension modules and the kernel. When the extension modules lock and unlock a resource, they should issue `lock_and_wait` system call with the argument of `LW_LOCK` and `LW_UNLOCK`, respectively. Likewise, when they wait for a locked resource, they should issue `lock_and_wait` system call with the argument of `LW_WAIT`. Using these information, the kernel of CAPELA detects whether deadlocks occur or not. Since deadlocks are caused by the interaction not only among multiple threads in one extension module but among multiple extension modules and the kernel, the kernel, which manages all extension modules, deals with the detection and recovery. The detection and recovery in the kernel are mentioned in Section 4.4.5.

4.3.3 Implementation Details

This section describes implementation details of how the protection manager enables the extension modules to run at the user level.

Emulation of a CPU Interrupt Level

The protection manager emulates disabling of hardware interrupts so that an upcall to an extension module is not interrupted by another upcall to the same module. In the kernel, the invocation of routines of subsystems is done not only from system calls but also from interrupt handlers. If an interrupt handler invokes a routine of a subsystem when a system call has invoked the same routine, the integrity of the kernel states may be lost. To prevent this, the kernel disables interrupts by changing a CPU interrupt level. However, since the extension modules running at the user level cannot change a CPU interrupt level appropriately, the protection manager uses an upcall enabling flag allocated in the kernel to emulate a CPU interrupt level. If the protection manager needs to disable interrupts, it sets the upcall enabling flag to 1 and then the kernel does not issue any upcalls while the flag is set to 1. This is not complete emulation of a CPU interrupt level but it is sufficient to run subsystems at the user level.

Thread

The protection manager uses threads so that it can simultaneously handle multiple upcalls which depend on each other. The threads that CAPELA provides for the extension modules are different from normal threads in two points. First of all, the contexts of our threads are not switched periodically. Since each upcall should be handled exclusively like the case where the functions of subsystems are executed in the kernel, context switches between threads are done only when a thread yields a CPU to sleep. Likewise, when a thread are woken up by the kernel or the other thread, the thread is first inserted into the runnable queue and then switched after the current running thread terminates or sleeps. Second, our threads change an upcall enabling flag to emulate CPU interrupts, if necessary, when the context is switched.

4.4 Implementation of the Kernel

The kernel of the CAPELA operating system provides several facilities for enabling to create the extension modules at the user level and to achieve the fail-safe mechanism.

4.4.1 Installation and Uninstallation of Extension Modules

CAPELA provides the facility to install and uninstall the extension modules. When installing an extension module for running at the user level, the protection manager issues the `modregist` system call. In this system call, the kernel first creates an object that holds information for the extension module. The object consists of an upcall handler, a process ID for the user-level module, a module name, an upcall enabling flag, information on shared memory, and so on. The upcall handler is used when hooked events occur in the kernel and then the extension module is called from the kernel. The process ID and the module name is used for identifying the module. If the upcall enabling flag is set to 0, upcalls to the extension module is disabled as described in Section 4.3.3. The information of shared memory includes the address of the shared memory for communication between the kernel and the extension module, the size, and so on. Finally, the `modregist` system call invokes an initialization routine depending on a type of the extension module, for example, a file system module, a network subsystem module, and so on.

When an extension module running at the user level terminates normally or abnormally, the protection manager issues the `modunregist` system call. If the protection manager cannot issue this system call due to unexpected signals, the kernel executes this system call instead. This system call first cleans up the kernel to get rid of the impact made by the extension module. This clean-up routine is the same with a recovery routine described in Section 4.4.4.

On the other hand, installation and uninstallation of an extension module running at the kernel level are done by the Loadable Kernel Module (LKM) mechanism. When uninstalling the extension module, the protection manager executes the clean-up routine as well as the uninstallation of extension modules running at the user level.

4.4.2 Shared Memory

CAPELA provides memory shared between the kernel and each extension module running at the user level. System V shared memory architecture is provided in NetBSD 1.3.2, which is the base of CAPELA, but it is for general use. This general shared memory architecture allows the users to limit the access right to the memory only in the style of the UNIX access control. In short, the users can set readable, writable, and executable to only three types of owner, group, and others. To make matters worse, checking the access right to the shared memory is done only once when it is mapped, so that anyone can access the memory with the specified access mode such as read-only or read-write. Such an access model is not suitable for keeping shared memory safe from malicious user processes.

New Shared Memory Architecture

To solve this problem, we have implemented a new shared memory architecture with fine-grain access control. Our shared memory architecture has two different points from the generic shared memory architecture. One is that the shared memory is mapped only between an extension module and the kernel so that the shared memory is not accessed illegally by the other user processes. The shared memory is automatically mapped only when the protection manager calls the `modregist` system call. The other different point is that only the protection manager that owns the shared memory is allowed to issue the `shmprotect` system call for changing the protection of the shared memory. The other user processes cannot illegally remove the protection.

For each page table entry of the shared memory that an extension module owns, CAPELA sets a user-accessible bit of a page table entry so that the extension module can access its own shared memory. For the other user processes, on the other hand, CAPELA does not set the bit so that these processes cannot access it. Using this hardware bit, CAPELA allows only an extension module to access its own shared memory. However, the kernel may need to access the shared memory even if the CPU context is not of the extension module in order to refer to the kernel data on the shared memory. Therefore CAPELA makes every shared memory of the extension modules accessible for the kernel. In addition, each shared memory is allocated in a different virtual address so that the kernel deals with every shared memory simultaneously. This access control is illustrated in Figure 4.7.

Our shared memory has the limitation that CAPELA cannot access the shared memory in interrupt handlers. During interrupts, a page table is undefined since any interrupt handlers are executed without binding to any processes. Since our shared memory is swappable virtual memory unlike the kernel memory, which permanently resides on physical memory, the whole system crashes if a page fault occurs for the shared memory.

Shared Memory vs. IPC

Why does CAPELA use shared memory for communication between the kernel and each extension module running at the user level? Many systems like Mach use IPC instead. The reason why we use shared memory is that shared memory has two advantages over IPC. First of all, communication using shared memory gets more efficient than that using IPC. IPC needs to copy data from the kernel address space to an address space of an extension module (the reverse is not always necessary), whereas shared memory needs no copy. Second, shared memory enables to communicate complicated data structure more easily and more efficiently. For example, shared memory allows data structure with pointers as long as the pointers point to the data on the same shared memory. For IPC, on the other hand, the system must serialize complicated data structure by traversing pointers and copying the data to which they point. Otherwise, the system must lazily copy data on demand and this suffers from large overheads.

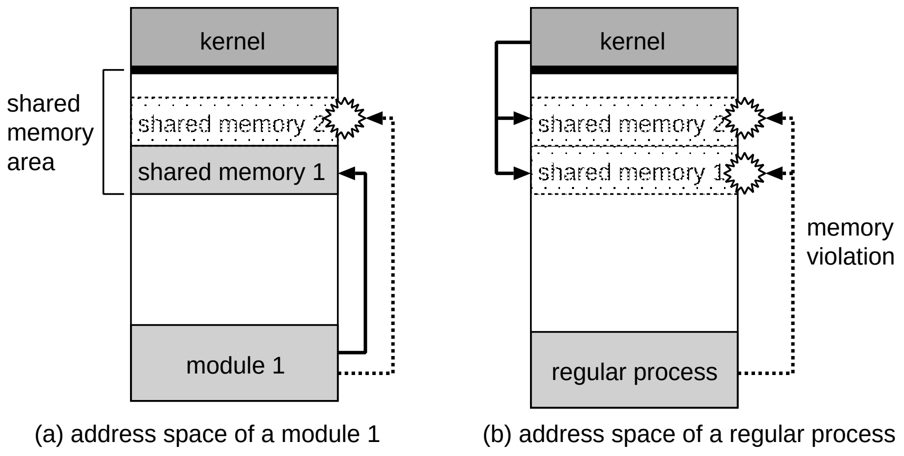


Figure 4.7: Shared memory with unique access control for communication between an extension module and the kernel. In the address space of extension module 1, the module can access its shared memory 1 but cannot access shared memory 2 for module 2. In the address space of a regular process, the process cannot access either shared memory. In either case, the kernel can access both shared memory.

However, shared memory has a disadvantage. Using shared memory is risky because the shared memory can be corrupted by an extension module that owns it if the module has errors. On the other hand, IPC is safer because the extension module cannot corrupt the kernel data directly. We believe that the risk can be avoided. The other user processes are not allowed to corrupt the shared memory. The destruction of the shared memory due to errors of an extension module is prevented if the protection manager protects the shared memory. Until the extension module is enough stable to corrupt no shared memory, the protection manager should protect the memory.

4.4.3 Upcall

When a hooked event occurs in the kernel, CAPELA issues an upcall to an extension module hooking the event if the module is running at the user level. Strictly speaking, the upcall is received by the protection manager. An upcall mechanism is implemented using a similar mechanism to signal's. A routine for calling an upcall handler, which is called trampoline code, are put on a user-level stack of every process like a signal mechanism when CAPELA initializes it. There are two differences between upcalls and signals. First of all, upcalls enable CAPELA to pass parameters to an extension module while signals pass only a signal number. Second, upcalls can be issued only by the kernel while signals can be issued by not only the kernel but also the other user processes using

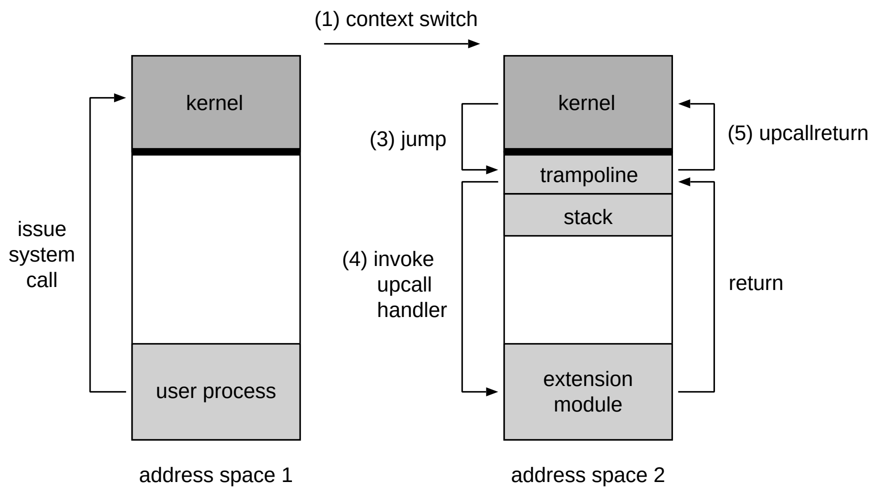


Figure 4.8: The control flow of the execution when a user process issues system calls that are related to an extension module.

the `kill` system call. If a regular user process issues an upcall, the extension module that receives it is confused.

An upcall is handled as illustrated in Figure 4.8.

1. When a process issues a system call and raises an event hooked by an extension module, CAPELA makes the process sleep and then forces switching the CPU context to that of the extension module that receives that upcall.

2. An upcall routine in the kernel puts an upcall handler, the parameters of the upcall and information for returning from the upcall on the user-level stack of the extension module. The parameters are allocated on the shared memory so that the extension module running at the user level can access them.

3. The upcall routine jumps to the trampoline code on the user-level stack. Since the kernel cannot call a routine of a user process directly, the upcall routine rewrites the return address from the routine so that the address points to the trampoline code on the user-level stack. When the routine is exited, the thread of control moves to the trampoline code.

4. The trampoline code invokes the upcall handler of the extension module passed as the parameter.

5. After the upcall handler is exited, the trampoline code issues the `upcallreturn` system call and returns to the kernel.

Upcalls cannot be issued during interrupts because it is impossible to switch the CPU context from an interrupt handler to the other processes and because CAPELA does not allow an interrupt handler to write the parameters of upcalls to the shared memory. To avoid this problem, CAPELA delays upcalls that are issued during interrupts and then the upcalls are processed after the interrupt handler finishes.

4.4.4 Recovery from Illegal Memory Accesses

CAPELA has the ability to recover from an error of an extension module so that the kernel is not affected by the abnormal termination of the erroneous module. As the reasons why the kernel gets unstable due to the abnormal termination, it is considered (1) that the kernel refers to the kernel data on the shared memory owned by the erroneous module and (2) that the erroneous module has modified the kernel state so that the kernel gets unstable.

In the former case, the kernel must modify the references to the kernel data on the shared memory so that the kernel does not access non-existing memory and does not occur a kernel fault after the extension module is removed. First of all, the kernel removes all entries registered by the extension module. For a file system module, for example, the `mount` object for the management of mount information is removed from the mount list. At the same time, the kernel changes the current working directories of all processes so that they do not point to any `vnode` on the file system. In the current implementation, the current directory on the removed file system is changed to the root directory. For a network subsystem module, the `protocol switch` object is removed from the protocol switch table.

In the latter case, the kernel must restore the kernel state modified by the erroneous module so that the kernel is kept stable. To restore the kernel state, the kernel prepares a log per extension module and records the operations to be recovered with which the extension module changes the kernel state. To keep the size of the log small, an operation recorded in the log is deleted when a new operation can cancel out the previous operation. For instance, the increment of the reference count of a `vnode` are cancelled out with the decrement of the same `vnode`. Likewise, the lock of a `vnode` is also cancelled out with the unlock of the same `vnode`. When restoring the unstable kernel states, the kernel examines the log and executes the operations recorded in the log in a reverse order. For example, the reverse operation of increasing the reference count is to decrease it and the reverse operation of locking is to unlock. Figure 4.9 illustrates the record of a lock operation in a log and the recovery based on the log.

4.4.5 Detection and Recovery of Deadlocks

CAPELA periodically checks whether the system falls into a deadlock state. When locking, unlocking, and waiting for resources, the protection manager of an extension module notifies CAPELA of that operation using the `lock_and_wait`

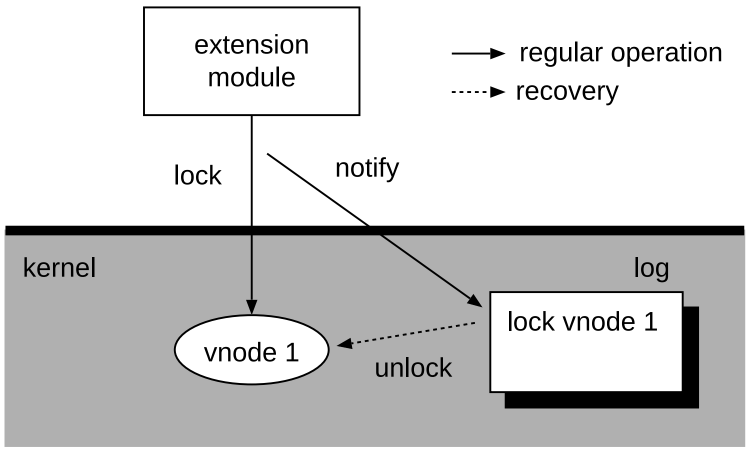


Figure 4.9: Recording operations in a log and the recovery based on the log. When the extension module locks the vnode 1, the operation is recorded in the log. At recovery, the kernel unlocks the vnode 1 from the recorded operation in the log.

system call as explained in Section 4.3.2. Then CAPELA creates a wait-for-graph like Figure 4.10 on the basis of that information. To detect deadlocks, CAPELA checks the wait-for-graph by examining the dependencies of resources. The detection algorithm is as follows:

1. CAPELA marks all resources in the wait-for list of each thread of extension modules and the kernel with *UNTOUCH*. The wait-for list contains resources that the thread is waiting for.

2. For each wait-for list, CAPELA pushes all resources into a stack and marks them with *INSTACK*.

3. For one thread, CAPELA peeks a resource in the top of the stack. If the resource is marked with *EXTRACT*, CAPELA pops it from the stack, marks it with *DELETE*, and repeats the operation of 3. Otherwise, CAPELA marks it with *EXTRACT* and search the thread that locks this resource using the lock lists of all threads. The lock list contains resources that the thread is locking.

4. If CAPELA finds such a thread, CAPELA examines each resource of the wait-for list of the thread found. If CAPELA finds a resource marked with *EXTRACT*, there is a loop in the wait-for-graph; in short, the system is in a deadlock state. If CAPELA finds a resource marked with *UNTOUCH*, CAPELA marks it with *INSTACK*. If any resources in the wait-for list are not marked with *UNTOUCH*, CAPELA backtracks.

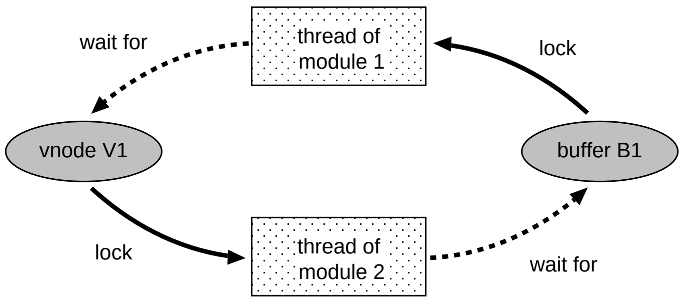


Figure 4.10: The example of a wait-for-graph in CAPELA. The thread 1 locks the buffer `B1`, which the thread 2 waits for, and waits for the vnode `V1`, which the thread 2 locks.

5. On the other hand, if CAPELA does not find out any threads that lock the target resource, CAPELA backtracks and repeats the operation of 3.
6. CAPELA performs this algorithm for all threads until a deadlock is found or all resources are marked with *DELETE*.

Let us apply this algorithm to the example of Figure 4.10. First, the vnode `V1` and the buffer `B1` are entered in stacks for the thread 1 and the thread 2, respectively, and are marked with *INSTACK*. CAPELA peeks `V1` from the stack for the thread 1 and marks it with *EXTRACT*. Then CAPELA search threads that lock `V1` and finds the thread 2. Since there is no resources marked with *EXTRACT* in the wait-for list of the thread 2, CAPELA backtracks. Since the thread 1 has no more resources that it waits for, the search for the thread 1 is finished. Next, CAPELA peeks `B1` from the stack for the thread 2 and marks it with *EXTRACT*. Then CAPELA search threads that lock `B1` and finds the thread 1. Since `V1` in the wait-for list of the thread 1 is marked with *EXTRACT*, CAPELA can detect a deadlock between `V1` and `B1`.

CAPELA allows the protection manager to change the interval between checks for deadlocks. If the users want to detect deadlocks earlier, they can set a shorter value to this interval although this degrades the performance of the whole system. On the other hand, if they want to decrease the overheads, they can set a longer value to the interval although it takes more time to detect deadlocks.

Since the occurrence of deadlocks depends on timing, CAPELA may be able to run the deadlocked extension modules by resolving the deadlock and retrying its execution. To destroy a loop in the wait-for-graph, CAPELA temporarily releases one of the locks in the loop. A thread that has a temporarily released lock is suspended for a while so that another thread can obtain the lock.

4.4.6 System Calls

CAPELA provides several special system calls for the protection manager to use internally. These system calls are used to create extension modules at the user level and to achieve a fail-safe mechanism for the modules. Although not only the protection manager but also the extension modules written by programmers can issue these system calls, the extension modules should not.

The `modregist` system call installs a process that issues this system call as an extension module to the kernel. For the parameters, the type of the extension module, its name, an upcall handler, and private data are passed. The type must be `MLP_FS` for a file system module or `MLP_NS` for a network subsystem module in the current implementation. The name must be able to identify the extension module. The private data is passed to an initialization routine for each module type. CAPELA allows only the super user to use this system call so that malicious users cannot install any extension modules illegally.

The `modunregist` system call uninstalls an extension module that issues this system call from the kernel. In this system call, a kernel state is cleaned up so that the uninstalled module does not leave a bad influence to the kernel. Since this system call allows only the process that has installed an extension module to unregister itself, the other user processes cannot uninstall any extension modules illegally.

The `shmalloc` system call allocates the area of shared memory and returns the address. The maximum size is limited by the size of the shared memory. Only the extension modules installed by the `modregist` system call can use this system call.

The `shmfree` system call releases the area of the shared memory indicated by the parameter. If the address to be released does not point to the shared memory owned by the extension module, this system call returns an error. Like the `shmalloc` system call, this system call does not allow to be used by regular processes.

The `shmprotect` system call changes the protection of shared memory. The argument is the combination of `PROT_READ` for read permission and `PROT_WRITE` for write permission. The protection of shared memory provided by CAPELA must be changed by this system call instead of the `mprotect` system call. In CAPELA, the `mprotect` system call does not allow to change the protection of the shared memory. This system call are also secure because only a process that owns the shared memory can use it.

The `upcallsuspend` system call is used to wait for a signal or an upcall. When receiving an upcall, the protection manager wakes up from this system call and makes the extension module handle it.

The `kernfunc` system call executes a kernel function specified by the parameter. Since this system call may corrupt the kernel data due to the execution of kernel functions, regular user processes cannot use it. Some of defined kernel functions are listed in Appendix D.

The `logctl` system call controls the logging for recovery. If the parameter is `RECLOG_CHECK`, the kernel checks a recorded log to examine whether the kernel is unstable or not. If the parameter is `RECLOG_ROLLBACK`, the kernel checks the log and rolls back if the kernel is unstable. If the size of the log is not zero, it is determined that the kernel state is unstable.

The `lock_and_wait` system call notifies the kernel of locking, unlocking, waiting for, and waking up on a kernel resource such as `vnode`. When `LW_LOCK` and the pointer to a kernel resource are passed to this system call, the kernel inserts the pointer in the lock list. The pointer inserted in the lock list is removed when this system call is issued with the parameters of `LW_UNLOCK` and the same pointer. On the other hand, when `LW_WAIT` and the pointer to a kernel resource are passed, the kernel inserts the pointer in the wait list. The pointer inserted in the wait list is removed when this system call is issued with the parameters of `LW_WAKEUP` or `LW_WAKEUP_ALL` and the same pointer.

4.4.7 Implementation Details

Validity Check for Kernel Function Parameters

To protect the kernel from erroneous modules, CAPELA checks the validity of the parameters of the kernel functions that can be called by extension modules. This validity check examines if the addresses to which the parameters point are safe. If the validity check fails, the kernel function returns an error if possible. The overhead of this check is almost negligible since the comparison of addresses takes less time than the execution of the function in many cases.

Optimized Memory Allocation

The kernel provides an optimized memory allocation routine for the extension modules running at the kernel level. Our extension modules frequently allocate and corrupt memory since they are written in the object-oriented language C++ and deal with many objects. Since the memory allocation routine suffers relatively large overheads, it can become performance bottleneck. To solve this problem, the kernel allocates memory of a fixed size before the extension modules start and does not use the general memory allocation routine after that. If the pre-allocated memory gets short, the kernel allocates more memory again for the extension modules.

4.5 Examples of Extension Modules

CAPELA allows the users to extend the operating system for each subsystem. The extension modules are therefore relatively course-grain. We believe, however, that the suppression of extensibility makes it easier to extend the operating system. If programmers would like to change only a part of a subsystem, they can create a new extension module that delegates the execution to the original subsystem for the routines that they do not need to change. In the current implementation, CAPELA supports to extend file systems and network subsystems.

4.5.1 File System Modules

The file systems are one of subsystems that are developed the most. Since the file systems affects the system performance largely, it is meaningful to improve the file systems and to develop new file systems. In particular, distributed file systems such as AFS [94] and Coda [95] are researched recently. CAPELA helps developers create such distributed file systems as well as local file systems.

In the current implementation of CAPELA, the file systems are implemented on top of the virtual file system (VFS) [64]. A file system module serves the applications while it is mounted on a directory. When a file system is mounted, `FileSystem::mount` method is invoked by the kernel and then prepares for the file system. When a file system is unmounted from the directory, `FileSystem::unmount` method is invoked and then cleans up the file system. The most primitive operations for a file system are to read and write files. When a file is read, `FileSystem::read` method is invoked. This method should read the contents of the file into a file buffer and copy it to a universal I/O buffer. Conversely, when a file is written, `FileSystem::write` method is invoked. This method should copy the contents of a universal I/O buffer to a file buffer and write it to a file on a physical device. Additionally, a file system module should support changing a directory, getting a file status, getting directory entries, and creating a file at least.

In distributed file systems, some of file operations such as read and write invoke local file systems such as UFS to perform requested operations. Since such file systems may modify the state of the local file systems in the kernel, the kernel records accesses to the local file systems in a log so that the kernel can recover the local file systems when an unexpected accident occurs.

4.5.2 Network Subsystem Modules

The network subsystems are indispensable to recent operating systems. When the programmers develop a new distributed file system, they may want to develop a new network protocol so that the distributed file system can communicate between clients and servers most efficiently. As a more close example to us, the present IP version 4 is being changed IP version 6 in order to prepare for the shortage of IP address in the near future.

In the current implementation of CAPELA, the network subsystems are implemented on top of an IP layer or an Ethernet layer. A network subsystem serves the applications while it is bound to a socket. When a network subsystem is bound to a socket, `NetworkSystem::attach` method is invoked by the kernel and then prepares for the network protocol. When a network subsystem is detached from a socket, `NetworkSystem::detach` method is invoked and then cleans up the network protocol. The most primitive operations for network subsystems are to send and receive packets. When a packet is passed from a socket layer above, `NetworkSystem::send` method is invoked. This method should divide a packet if necessary, attach a protocol header, and call an output routine of an IP layer or an Ethernet layer below. Conversely, when a packet is received on an IP layer or an Ethernet layer, `NetworkSystem::input` method is invoked. This method should control the sequence of packets if necessary, and append it to a socket buffer. The data in a socket buffer is read by the `recv` system call.

4.6 Automatic Distribution of Network Modules

Network modules developed as extension modules have to be distributed to the other host with which one host is going to communicate. If CAPELA distributes new network modules on demand, the users can use the most suitable network protocol flexibly. This automatic distribution solves the problem that all hosts that communicate with each other have to get necessary network modules in advance, for example, via FTP or HTTP.

One of the problems to be solved for automatic distribution is how the system assigns a unique protocol number that a network module uses. For standard protocols such as TCP, a protocol number that identifies a network protocol is assigned globally by Internet Assigned Numbers Authority (IANA). However, for non-standard protocol such as new protocols, assigning a fixed protocol number is not realistic. For such non-standard protocols, the users cannot ask IANA to assign a unique protocol number. On the other hand, if the same protocol number were assigned for different protocols, hosts would not be able to communicate using such protocols. Also, a centralized server that dynamically assigns protocol numbers for non-standard protocols is not suitable from the viewpoint of scalability.

To address this problem, CAPELA dynamically resolves a protocol number used by a new network protocol between hosts that communicate with each other. In this section, we first describe a meta protocol used for dynamic resolution of protocol numbers. Then we mention how new network modules are distributed between two hosts.

4.6.1 A Meta Protocol for Dynamic Protocol Number Agreement

We have developed a meta protocol called Dynamic Protocol Number Agreement Protocol (DPNAP), which enables two hosts to communicate using a network protocol whose protocol number is not assigned statically. First of all, CAPELA assigns a protocol number that can identify a network protocol only within the local host. In general, this assignment is different for each host. To agree with the other hosts, CAPELA uses DPNAP and then creates a translation table for looking up a network protocol from a protocol number used within the other hosts. This translation table is referred to when CAPELA receives a packet from network. An input to this translation table is a tuple of a network address and a protocol number. An output is a pointer to the packet handling for the corresponding network protocol.

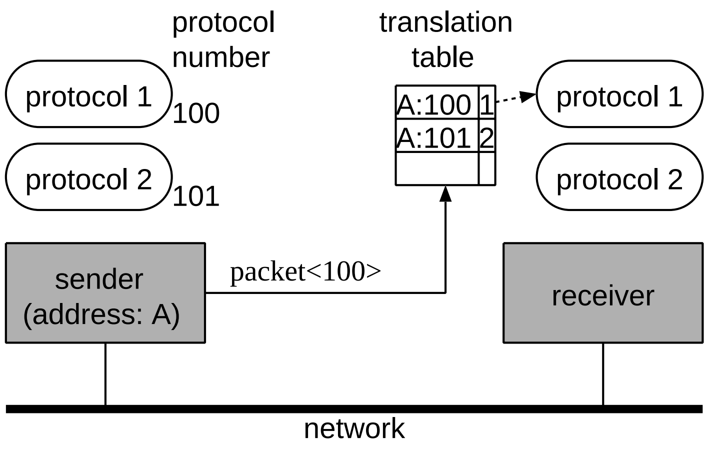


Figure 4.11: Protocol decision by referring to a translation table. When the receiver host receives a packet with the protocol number 100, it decides that the used protocol is 1 from the translation table even if a protocol number for the protocol 1 is not 100 in the receiver host.

Figure 4.11 shows how CAPELA decides a network protocol when receiving a packet. Suppose that a sender host whose network address is A sends a packet with a protocol number 100, which stands for a protocol 1. When receiving the packet, a receiver host refers to its translation table and then can recognize that the packet is sent using protocol 1. Thus the receiver host can carry out the routine for handling packets of a protocol 1 correctly.

Algorithm of DPNAP

When a sender host is going to send the first packet to a receiver host, it sends a *NOTIFY* message of DPNAP to that host before sending the first packet. The *NOTIFY* message consists of a network address of the sender host, a protocol number assigned within the sender host, a name and version of the protocol. The name and version of the protocol must identify the only network protocol uniquely. This global naming includes the same problem with unique assignment of protocol numbers, but it is easier to avoid collision if a developer of a network module gives his protocol as a long and descriptive name as possible. When the receiver host receives a *NOTIFY* message, it sets a new entry to its translation table so that it can look up a pointer to the packet handling routine for the corresponding protocol from a tuple of the network address and the protocol number sent. Figure 4.12 shows how the receiver host can handle a packet of a new protocol.

This algorithm allows the two hosts to communicate without any confusion after the sender host is rebooted. Since rebooting a host may change the as-

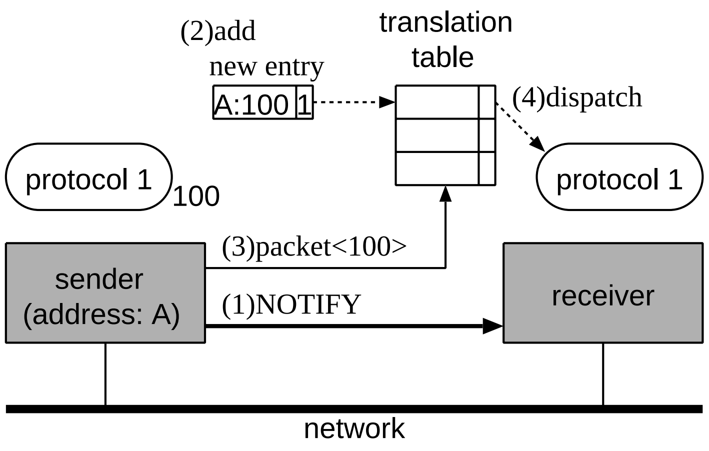


Figure 4.12: A *NOTIFY* message sent before the first packet sending. By this message, the sender host lets the receiver host know that the protocol number 100 from the network address A means the protocol 1.

signment of protocol numbers before rebooting, the integrity of the translation table in the receiver host may be lost. However, before the rebooted sender host sends the first packet to the receiver host, it sends a *NOTIFY* message to the receiver host again. If the receiver host receives the message, it discards an old entry from the translation table, if any, and adds a new entry, so that the integrity of the translation table is preserved.

When the receiver host is rebooted, it may receive packets of a protocol whose entry does not exist in the translation table. The reason is that protocol information sent by the *NOTIFY* message before the first packet is lost. If the receiver host receives a packet of an unknown protocol, it sends a *REQUEST* message of DPNAP to the sender host. The *REQUEST* message consists of a protocol number that is contained in a header of the received packet. If the sender host receives the *REQUEST* message, it sends back a *REPLY* message of DPNAP. The *REPLY* message consists of the name and version of a protocol corresponding to the sending protocol number. When the receiver host receives the *REPLY* message, it adds a new entry to the translation table in the same way with a *NOTIFY* message. Figure 4.13 shows re-negotiation with *REQUEST* and *REPLY* messages.

A host preserves packets of an unknown protocol until it receives a *REPLY* message after it sends a *REQUEST* message. The preserved packets are handled after a necessary entry is added in the translation table by a *REPLY* message.

DPNAP can be used for broadcast and multicast communication as well as one-to-one communication. For hosts that participate in communication from the beginning, a sender host sends a *NOTIFY* message to these hosts before it sends the first packet. For hosts that participate in communication from

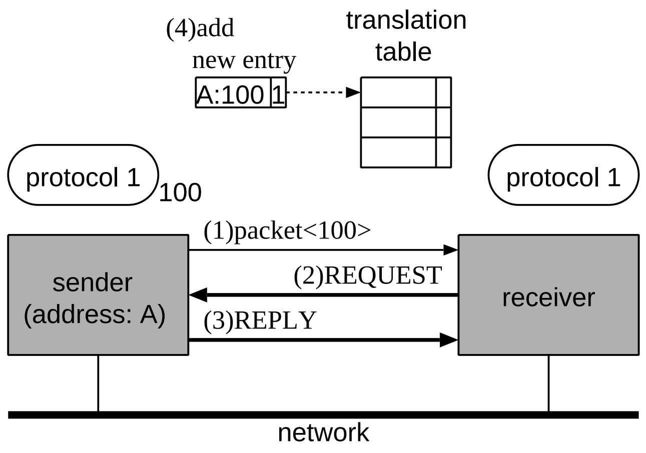


Figure 4.13: Re-negotiation with *REQUEST* and *REPLY* message. The receiver host can reconstruct the translation table so that the protocol number 1 from the network address A corresponds to the protocol 1.

the middle, these hosts send a *REQUEST* message to a sender host when they receive a packet of an unknown protocol since they have not received a *NOTIFY* message. If they receive a *REPLY* message, they can create correct translation tables and communicate with each other.

4.6.2 Automatic Distribution Mechanism

When a host receives a *NOTIFY* or *REPLY* message of DPNAP, it retrieves a network module corresponding to the name and version of a protocol sent by the message as well as adding a new entry to the translation table. If a corresponding network module has not loaded yet, CAPELA requests a user-level daemon called `netmodd` to load a necessary module. `Netmodd` first examines if the specified module exists on a local disk and, if it exists, load it to the system. In the current implementation, `netmodd` uses a loadable kernel modules (LKM) mechanism of NetBSD.

If the specified network module does not exist on a local disk, `netmodd` requests the module of a sender host of a *NOTIFY* or *REPLY* message of DPNAP. `Netmodd` of the sender host sends back the requested module and then `netmodd` of the receiver host loads it to the system as illustrated in Figure 4.14. At this time, CAPELA assigns a protocol number unique within the receiver host to the protocol implemented by the loaded module.

For completely automatic distribution of network modules, the system has to automatically configure a network interface such as a network address when a module is loaded to the system. In UNIX, this configuration is manually done by `ifconfig`. The problem of automatic configuration is how to assign an network

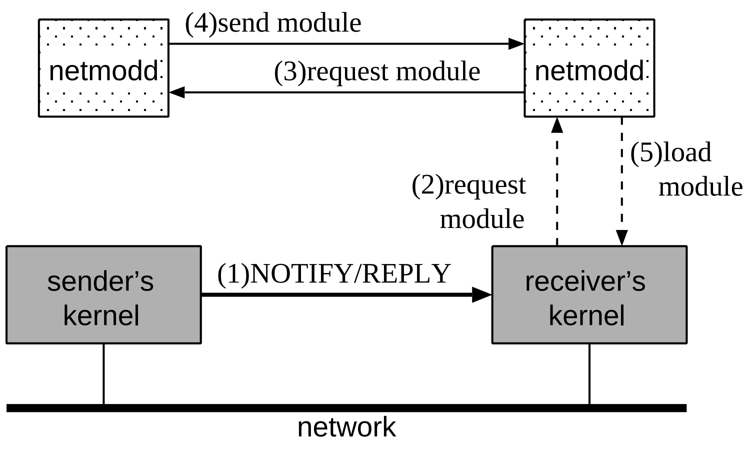


Figure 4.14: Automatic module distribution via network. If the receiver's kernel does not have a necessary module, it dynamically loads it. `Netmodd` retrieves a module from a local disk or the sender host.

address. In the current implementation, CAPELA automatically assigns either an Ethernet address uniquely assigned to a network interface card (NIC) or an IP address already assigned to the NIC for the other protocol.

From the viewpoint of applications, if a necessary network module does not exist in the system when an application issues the `socket` system call, the application is suspended. When the host receives a *NOTIFY* or *REPLY* message of DPNAP, `netmodd` loads a network module to the system and the application is resumed. CAPELA preserves packets sent to the application until it finishes binding a name to a socket and handles these pending packets after that. In UNIX, packets that the operating system receives are discarded if a name is not bound to the corresponding socket yet. This mechanism enables an application to handle all packets that it should receive even if a the sender host sends packets before loading a module is finished in the receiver host.

4.6.3 Change of Socket Interface

The traditional interface of the `socket` system call, which takes a protocol number for the 3rd parameter, is not suitable for the case where a protocol number is assigned dynamically. In our system, in the worst case, a protocol number is assigned when a network module is loaded from a remote host after the `socket` system call is issued. CAPELA provides another version of `socket` system call, which takes the name and version of a protocol instead of a protocol number for the 3rd parameter.

4.7 Summary

This chapter presented the multi-level protection. The multi-level protection enables the users to change the protection level of an extension module depending on its stability. Using the multi-level protection, the CAPELA operating system can construct a dynamic safety net that can make a suitable trade-off between fail-safety and performance. To change the protection level, CAPELA provides various protection managers. The protection manager, which is a gateway between an extension module and the kernel, isolates the extension module from the rest of the system using various protection techniques.

Each protection manager provides different abilities of error detection and recovery. Some protection managers use a protection technique of switching address spaces to detect illegal memory accesses. Some use a technique of replicating the kernel data to detect accidental data corruption. To recover from such protection violations, CAPELA uses a log in which the operations performed by erroneous modules are recorded. Moreover, some protection managers use a deadlock detection mechanism using a wait-for-graph. CAPELA also provides recovery from deadlocks.

The users can change the protection level without any modifications to extension modules due to a common API provided by the protection manager. As long as the programmers conform to this API, this binary-level compatibility among different protection levels is guaranteed since the API hides the differences of how each protection manager protects the extension module. This API consists of callback functions and manipulation functions for the kernel data. Callback functions are used for handling upcalls from the kernel when events hooked by the extension module occur in the kernel. Manipulation functions for the kernel data are used for allowing the extension module to invoke necessary kernel functions.

Additionally, CAPELA automatically distributes the network modules to other hosts. To solve the problem that a protocol number for a new network module cannot be fixed globally, CAPELA dynamically assigns a protocol number using a meta protocol called DPNAP. DPNAP mediates two or more hosts that participate in one-to-one or one-to-many communication so that each host can recognize which protocol is used for a received packet from the IP address and the protocol number locally assigned in each host.

Chapter 5

Experiments

5.1 Process Cleaning

We have developed the Compacto operating system by modifying the Linux 2.2.16 kernel. This section reports the results of our experiments. First we measured basic costs of saving and restoring a process state. Then, to show bare performance numbers of Compacto, we ran the Apache web server [4] directly on top of Compacto and measured the performance of that web server. The machine we used for the experiments is a PC with a Pentium III 933MHz processor[1] and 256MB memory.

5.1.1 Micro Benchmark

Cost of Save/Restoration

We first measured the execution time of the `save_state` and `restore_state` system calls. The majority of the execution time of the `save_state` system call is the cost of turning writable pages in a process into write-protected ones. This cost is listed in Table 5.1. For example, the Apache web server uses approximately 40 writable pages. The other costs included in `save_state` are of saving the states of several resources: 2.12 μsec for signal handlers, 0.90 μsec for the status of open files and sockets, 0.05 μsec for registers, and 0.33 μsec for issuing the system call. The influences of the number of opened files and sockets were negligible. The save cost C_{save} is shown as follows:

$$C_{save} = 0.05N_{mem} + 4.76$$

where N_{mem} is the number of saved memory pages.

For executing the `restore_state` system call, 0.33 μsec is taken for issuing the system call and 0.05 μsec is for restoring registers. The rest of the execution time of the system call depends on the number of the restored resources as

[1]The L1 cache is 16KB (instruction) + 16KB (data). The L2 cache is 256KB.

Table 5.1: The cost of turning writable pages into write-protected ones.

# of pages	12	16	32	64	128	256	512	1024
μsec	1.89	2.22	2.94	4.51	7.99	14.7	25.4	52.7

Table 5.2: The cost of restoring the states of resources (μsec).

# of resources	0	1	2	4	8	16	32	48	64
memory(remap)	0.50	0.63	0.84	1.18	1.90	3.43	8.92	24.1	36.3
memory(copy)	0.50	1.59	3.13	9.15	17.9	35.2	208	990	1468
signal handlers	0.08	0.27	0.33	0.38	0.60	1.05	–	–	–
files/sockets	0.05	0.60	0.69	0.83	1.05	1.49	2.84	–	–

listed in Table 5.2. As for memory pages, the remap strategy needed a smaller cost than the copy strategy. However, the remap strategy may need extra costs. If the `save_state` system call is not issued and the restored memory page is updated again before the next issue of the `restore_state` system call, Compacto must catch a page fault and allocate a shadow page. This cost is listed in Table 5.3. Therefore, comparing with the best case of the copy strategy that a process writes in the same set of pages every time, the remap strategy is 1.8 times slower at maximum.

When the users select the remap strategy, the whole restoration cost C_{remap} is shown as follows:

$$C_{remap} = \begin{cases} 3.48N_{mem} + 0.05N_{sig} + 0.07N_{file} + 1.16 & (0 \leq N_{mem} \leq 18) \\ 4.16N_{mem} + 0.05N_{sig} + 0.07N_{file} - 17.3 & (18 \leq N_{mem} \leq 27) \\ 40.6N_{mem} + 0.05N_{sig} + 0.07N_{file} - 1014 & (27 \leq N_{mem} \leq 64) \end{cases}$$

where N_{mem} is the number of modified memory pages, N_{sig} is the number of modified signal handlers, and N_{file} is the number of files/sockets whose status is changed. On the other hand, when the users select the copy strategy, the whole restoration cost C_{copy} is shown as follows:

$$C_{copy} = \begin{cases} 2.17N_{mem} + 0.05N_{sig} + 0.07N_{file} + 1.2 & (0 \leq N_{mem} \leq 27) \\ 39.7N_{mem} + 0.05N_{sig} + 0.07N_{file} - 996 & (27 \leq N_{mem} \leq 64) \end{cases}$$

Table 5.3: The cost of handling page faults.

# of pages	1	2	4	8	16	32	48	64
μsec	2.21	4.96	13.1	26.6	52.8	214	1023	1483

Table 5.4: The comparison of the cost of the fork-join method and process cleaning (μsec).

Breakdown	Fork-join	Process cleaning
Issue `fork/save_state`	0.3	0.3
Duplicate/save address space	17.3	1.9
Duplicate/save signal handlers	2.6	2.1
Duplicate/save file/socket info.	1.9	0.9
Others	3.6	0.0
Call libc exit function	11.7	0.0
Issue `exit/restore_state`	0.3	0.3
Destroy/restore address space	10.3	0.6
Destroy/restore signal handlers	0.2	0.1
Destroy/restore file/socket info.	0.7	0.1
Others	4.1	0.0
Page fault	2.2	2.2
Sum	55.2	8.5

Comparing with the Fork-Join Method

As noted in Section 2.4.2, the fork-join method is less efficient than process cleaning. One of the reasons is that the `fork` and `exit` system calls are slower than the `save_state` and `restore_state` system calls provided by Compacto. The comparison of the cost of the fork-join method and process cleaning is shown in Table 5.4. The fork-join method also depends on the number of writable memory pages and modified memory pages, and the number of open files and sockets. The numbers in this table were measured in a case where 12 pages were writable, 1 page was modified, and no files and sockets were opened or closed.

One large overhead of the fork-join method is to duplicate the address space. `fork` must create a new address space, copy the page table of the parent process, and turn writable pages of both the parent and child processes into write-protected ones. On the other hand, `save_state` only turns writable pages in the process into write-protected ones. Another large overhead is due to the exit function in the standard C library. Process cleaning does not need to execute that function since it does not terminate a process. The last large overhead is due to destroying the address space. Process cleaning does not need to destroy the address space since it does not create a different address space. These overheads become larger as the program size is larger and the process uses more resources.

The difference of the costs of the fork-join method and process cleaning seems to be too small from the viewpoint of real applications. However, the above cost of the fork-join method does not include the cost that arises from that a server cannot use the process pool technique. The synthetic cost of the two is shown in the next section.

In addition, the fork-join method makes a cache hit ratio worse (1) because memory caches such as L1, L2, and TLB are flushed whenever context switches between a parent process and a child process and (2) because a working set of memory gets larger due to the complicated `fork` and `exit` routines and a cache miss increases in number. This is also one reason why the fork-join method is slower than process cleaning.

5.1.2 Apache Web Server

We also measured the execution performance of a web server running on Compacto. As the web server, we used the Apache 1.3.12, which is implemented with the process pool technique. As client machines, we used PCs with a Celeron 300MHz processor and 64MB memory. The operating system of the client machines is FreeBSD 3.4. To avoid network saturation, the clients and the server are connected through the 100baseT Ethernet and the server machine has two Ethernet ports. We used the WebStone benchmark program [65], which measures the average number of requests that a web server can accept per second with various number of client machines. Only one client program was running on every client machine.

For comparison, we used four different types of Apache server. The POOL server is an Apache server that does not perform process cleaning. The COPY server is an Apache server performing process cleaning with the copy strategy. The REMAP server is an Apache server performing process cleaning with the remap strategy. These three servers use 16 pooled processes. Finally, the FORK server is an Apache server modified so that it uses the fork-join method. It does not use the process pool technique or perform process cleaning. We did not impose access restrictions on any of the four Apache servers in this experiment.

Figure 5.1 and Figure 5.2 show the server performance (the number of acceptable requests per second) and the response time, respectively, in a case where a 0 byte file was requested by the clients. All the servers modified 8 pages of memory, changed one signal handler, opened one file and one socket while handling every request.

Figure 5.3 and Figure 5.4 show the server throughput and the response time, respectively, in a case where various sizes of files were requested, All the servers modified 8.1 pages of memory on average, changed one signal handler, opened one file and one socket while handling every request. The maximum number of modified memory pages was 13 although the server used 44 writable pages. The requested data were copied from our real web server[2]. The requested data were various sizes of HTML files, binary files, and data created by CGI programs. The average size of requested data was 7.6KB (from 73 bytes to 47KB).

[2]`http://www.hlla.is.tsukuba.ac.jp/`

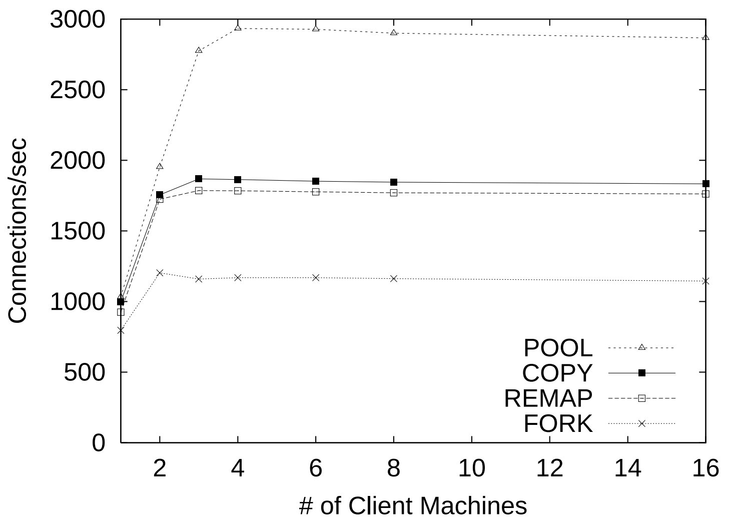


Figure 5.1: The server performance (a 0 byte file was requested).

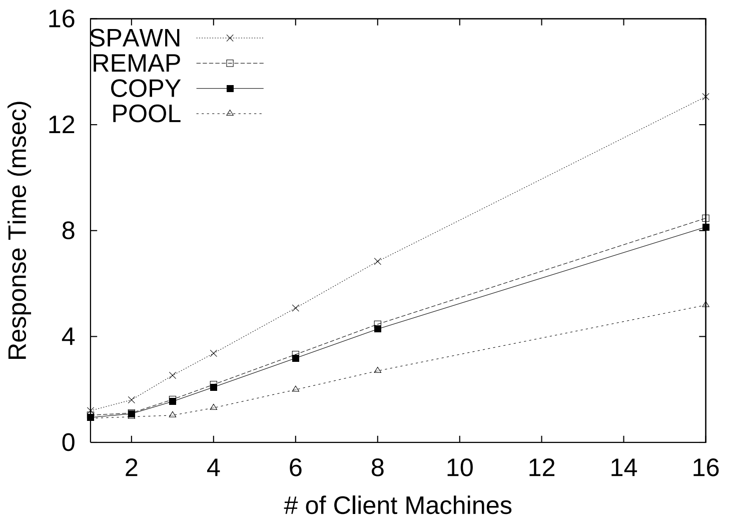


Figure 5.2: The average response time (a 0 byte file was requested).

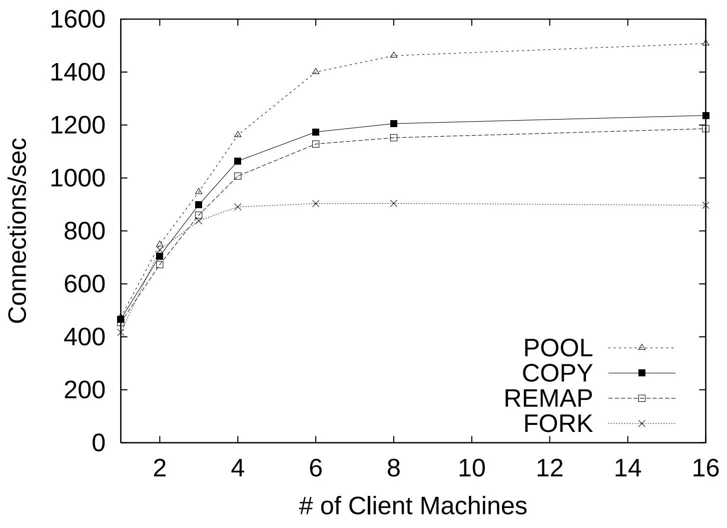


Figure 5.3: The server performance (various sizes of files were requested).

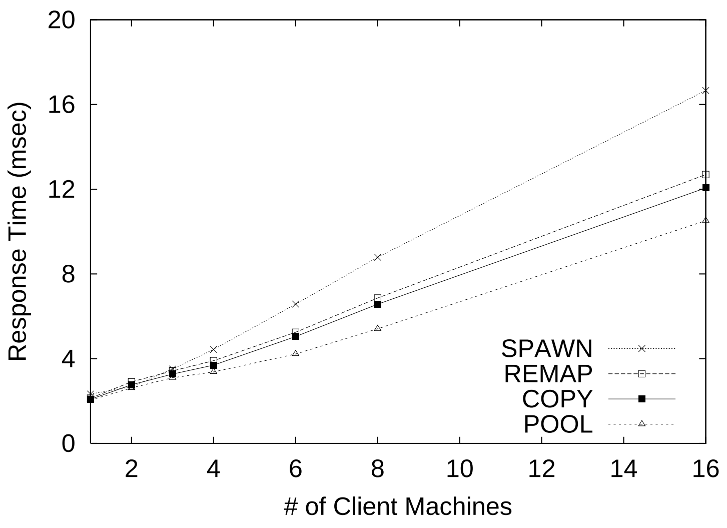


Figure 5.4: The average response time (various sizes of files were requested).

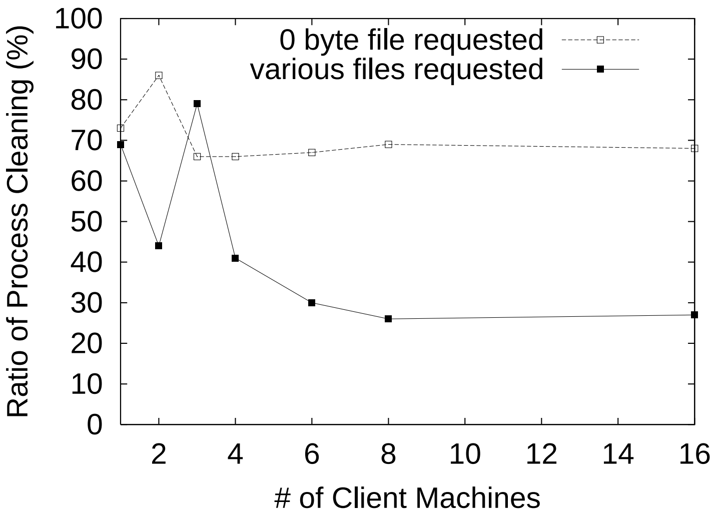


Figure 5.5: The ratio of the performance improvement by process cleaning to the whole improvement. The rest of the performance improvement is due to process pool.

Performance of Process Cleaning

According to Figure 5.1 and Figure 5.3, the COPY server is 60% and 30% faster than the FORK server, respectively. As described in Section 2.4.2, secure servers with the fork-join method, which impose access restrictions depending on each request, have had to create a new child process for every request. Process cleaning achieves great performance improvement over those traditional secure servers. It allows secure servers with process cleaning to handle a request with pooled processes although its performance penalties are 35% in the worst case, which is not negligible, if compared with the POOL server.

The performance improvement of the COPY server over the FORK server is due to both using process cleaning and using a process pool. To show the performance improvement only by process cleaning, we also measured a COPY server where the number of pooled processes is 1 and a FORK server where it handles requests sequentially. Figure 5.5 shows the ratio of the performance improvement by process cleaning to the whole improvement. From the result, in a case where a 0 byte file was requested, the ratio of the improvement by process cleaning and a process pool was 7:3. On the other hand, in a where that various sizes of files were requested, the ratio was 3:7. This result means that the performance improvement by process cleaning is relatively small, comparing with the fork-join method, if most of the execution time of a server is spent for disk and network I/O. Requests for large data cause more disk and network I/O than requests for a 0 byte file.

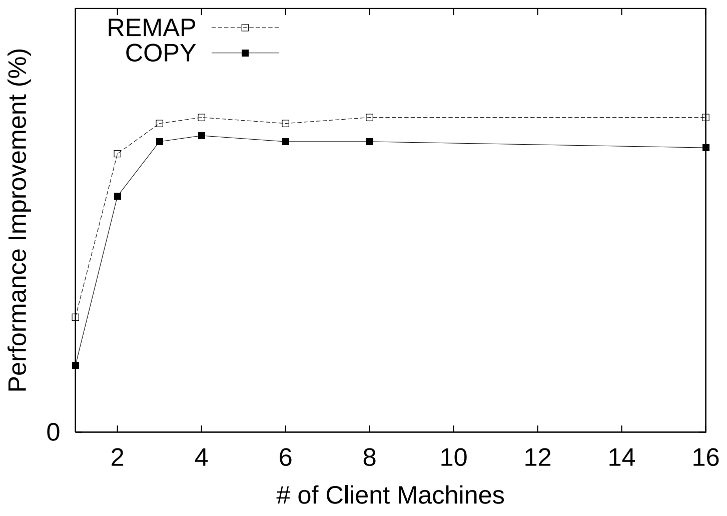


Figure 5.6: The improvement of server performance by optimizing a server program layout (a 0 byte file was requested).

Effectiveness of Optimization

In our experiments, the COPY server was always 5% faster than the REMAP server. In a case where a 0 byte file was requested, the copy strategy achieves the best performance since the same set of memory pages was updated whenever a request was handled. In a case where various sizes of files were requested, the copy strategy is also better than the remap strategy although the memory access pattern of the servers was less advantageous to the copy strategy than in the former case. Although the copy strategy is better than the remap strategy in the case of the Apache web server, the remap strategy may be better in the case of other kinds of servers.

Figure 5.1 and Figure 5.3 show the performance of the Apache servers in a case where the server programs are optimized as described in Section 3.3.3. Relocating the static data segment of the servers reduced the number of memory pages modified for static data from 10 pages to 1 page. As shown in Figure 5.6 and Figure 5.7, the performance of the COPY and REMAP servers, which perform process cleaning, was increased by 40% on average, comparing with using non-optimized server programs. This improvement includes the improvement of a cache hit ratio and the reduction of the overheads due to dynamic linking. In addition, the performance of the FORK server was also increased by the same degree since the fork-join method uses copy-on-write to copy pages for modified static data from a parent process to a child process.

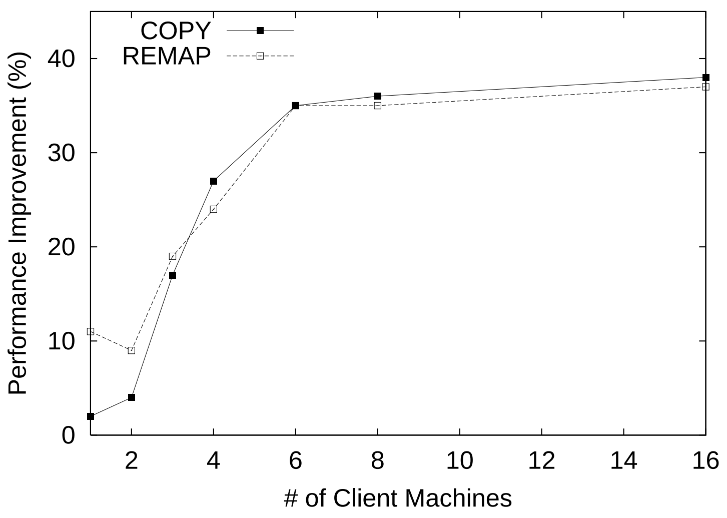


Figure 5.7: The improvement of server performance by optimizing a server program layout (various sizes of files were requested).

5.1.3 FastCGI

We also measured the execution performance of the FastCGI module [91], which runs on top of the Apache web server. The FastCGI module enables the server to use pooled processes for running CGI programs. Without the FastCGI module, the server must spawn a child process whenever a CGI program runs. As a CGI program, we used wwwcount [67] 2.5, which is one of the most popular access counters. Since wwwcount uses data file for a counter of each user or each page, accesses to the data files must be restricted depending on users or requested pages.

Like the experiment in the previous subsection, this experiment compared four web servers: the POOL, COPY, REMAP, and SPAWN servers. Only the FastCGI module used by the COPY and REMAP servers performs process cleaning. The SPAWN server does not use the FastCGI module. It spawns a child process for running a CGI program. The underlying servers of the four are the normal Apache server.

Figure 5.8 and Figure 5.9 show the results. These results are similar to the results in the case of the Apache web server. However, the performance overhead due to process cleaning was smaller in this experiment. The COPY server was only 8% slower than the POOL server. This is because executing a CGI program was a bottleneck and hence the overheads due to process cleaning were relatively small.

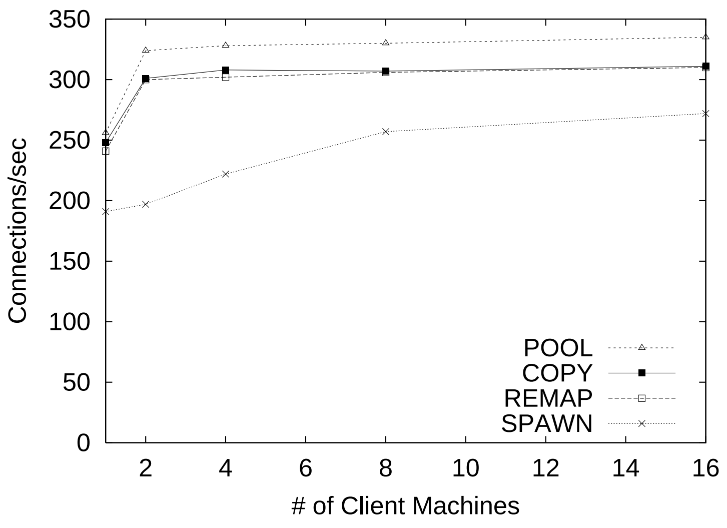


Figure 5.8: The server performance (a CGI program was requested).

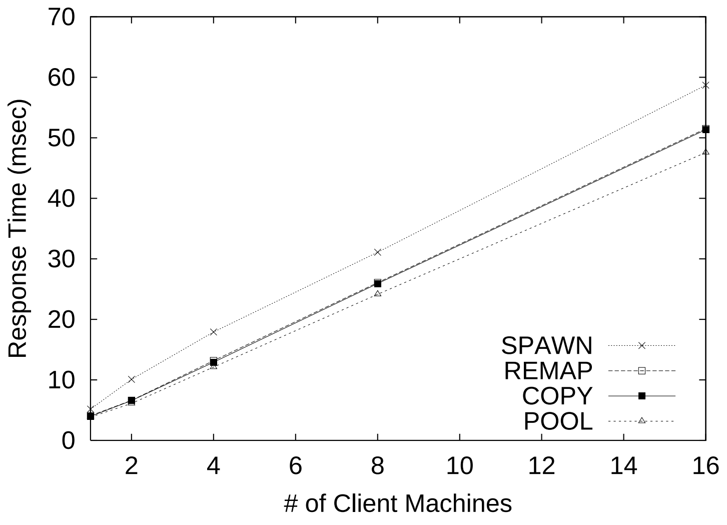


Figure 5.9: The average response time (a CGI program was requested).

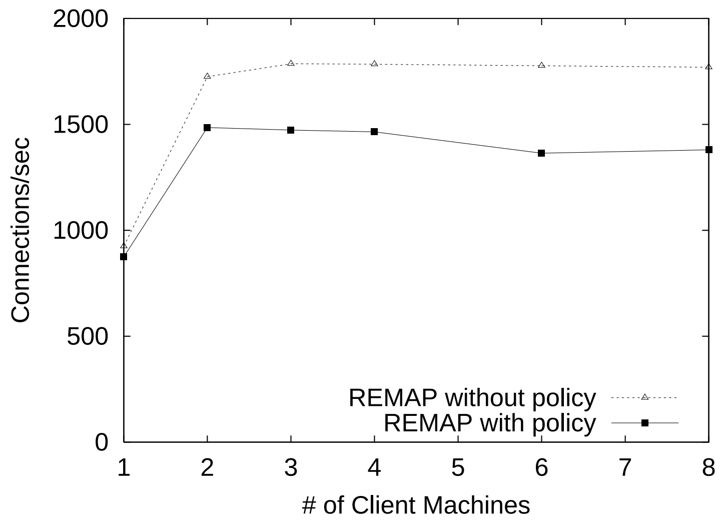


Figure 5.10: The overhead of policy checking. (0 byte file was requested).

5.2 Access Control

5.2.1 Policy Checking

Our Compacto checks rules of security policy whenever a server process issues a system call to restrict the privileges of the process. We applied policy rules described in Appendix B to the Apache web server and measured the overhead of policy checking. For this experiment, we used the REMAP server, which performs process cleaning to securely change policy rules for each request. The overhead of policy checking includes the overheads of both applying policy rules and invalidating the rules. We used the WebStone benchmark program and experimented under the same settings with the previous section.

Figure 5.10 and Figure 5.11 show the results. In a case where a 0 byte file was requested, the performance of the server with policy checking was 18% slower than that of the server without policy checking. On the other hand, in a case where various sizes of files were requested, the slowdown was 11% on average. We believe that these overheads are small to achieve the least privilege for the web server.

5.2.2 Taint Propagation

To examine the overheads for propagating taint information via network, we measured round-trip latency of TCP/IP and UDP/IP and throughput of TCP/IP using a benchmark program called netperf [48]. The overheads are caused by increasing a packet size due to an extra IP option, processing the IP option,

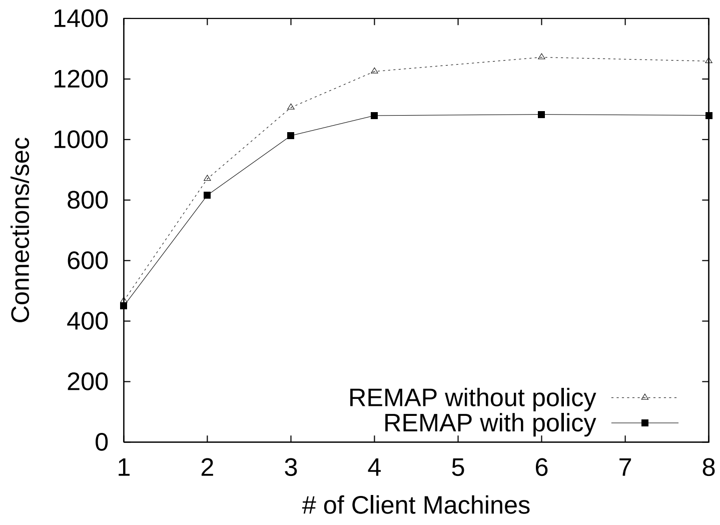


Figure 5.11: The overhead of policy checking. (various sizes of files were requested).

and updating taint information of a socket and a process. For round-trip latency, we measured request/response performance of netperf for various sizes of packets. Request/response performance is the average time that takes from sending a request and to receiving a response. For throughput, we measured stream performance of netperf, where data size that a process sends at a stretch is 64KB. For comparison, we also measured round-trip latency and throughput in a case where Compacto does not propagate taint information.

For this experiment, we used two PCs with a Pentium II 400MHz processor, 128MB memory, and a network interface card of 3COM Fast EtherLink. The operating system of this two machines is our Compacto based on Linux 2.2.11 kernel. Two machines are connected through the 100baseT Ethernet.

The results on round-trip latency are shown in Figure 5.12 for TCP/IP and Figure 5.13 for UDP/IP. Taint propagation makes round-trip latency 6% longer at maximum. Compacto suffers from this maximum latency when the sending packet size is small, but the latency is smaller in an average packet size. For throughput, the results are shown in Table 5.5. The degradation of throughput is 1.2% and very small.

5.3 Multi-level Protection

We experimented to make sure of the usefulness of the multi-level protection. We have three purposes in the experiments. The first purpose is to make sure that the execution performance is improved when the users lower the protection

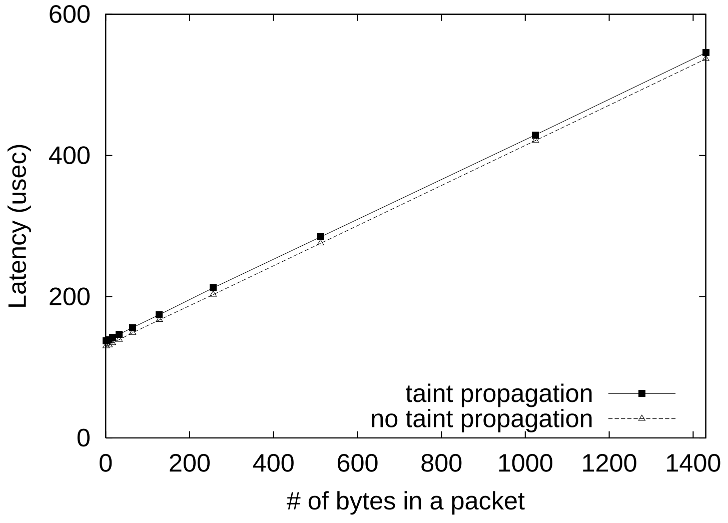


Figure 5.12: The increase of TCP/IP round-trip latency due to taint propagation.

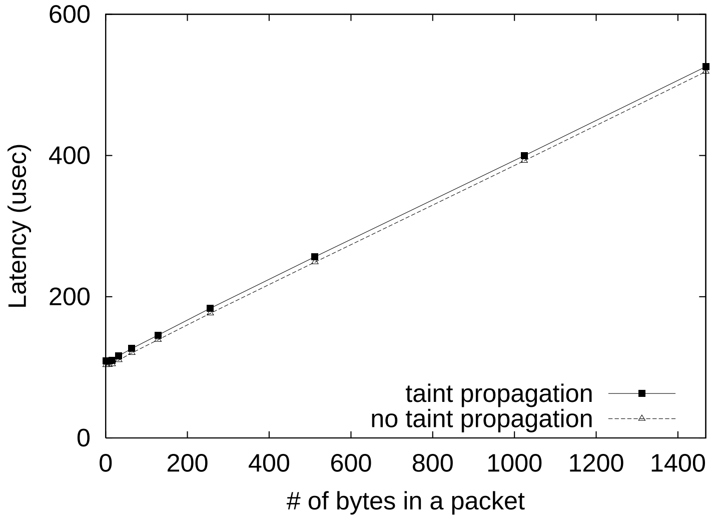


Figure 5.13: The increase of UDP/IP round-trip latency due to taint propagation.

Table 5.5: TCP/IP throughput degradation due to taint propagation.

	Mbps
Taint propagation	67.2
No propagation	68.0

Table 5.6: Five characteristic combinations of protection techniques. Shared memory full protection means that an extension module cannot read and write the shared memory directly. Shared memory partial protection means that an extension module can only read the shared memory directly. Kernel data replication means that the protection manager makes an extension module to access replicas of the kernel data. Address space switch means that an extension module is located at the user level.

Protection technique	1st	2nd	3rd	4th	5th
Shared memory full protection	√				
Shared memory partial protection	√	√			
Kernel data replication	√	√	√		
Address space switch	√	√	√	√	

level of the extension modules. The second is to measure the overhead of the maximum protection level. It is significant that this overhead are not too large although this overhead is not important for debugging of extension modules. The third purpose is to measure the overhead of the minimum protection level. This overhead is caused by making the API of multiple protection managers common. It is considered that the multi-level protection enables the extension modules to be as efficient as hand-crafted modules if this overhead is enough small.

For experiments, we used two PCs, which have a Pentium II processor running at 400MHz. Each PC was equipped with 128MB of RAM and a 10Mbps Ethernet. All experiments with network were performed between two machines on the same Local Area Network (LAN). All measurements were done using the CAPELA operating system.

Since it would have been too difficult to experiment with all combinations of the protection techniques that CAPELA provides, we selected five characteristic combinations, listed in Table 5.6, and experimented with them. These combinations can detect the errors on memory protection listed in Table 5.7. We call the combination yielding the highest protection level *the first level* and call the combination yielding the lowest level *the fifth level*. In our experiments, the protection levels from 1st to 5th are considered as linear.

The first level locates an extension module in a user address space, protects the shared memory from illegal memory reads and writes, and replicates the kernel data. The second level is different from the first level in that it protects

Table 5.7: Detectable errors on memory protection at each protection level.

Type of error	1st	2nd	3rd	4th	5th
Illegal reads on shared memory	√				
Illegal writes on shared memory	√	√			
Accidental kernel data corruption	√	√	√		
Illegal accesses to the kernel	√	√	√	√	

the shared memory only from illegal writes. The third level does not protect the shared memory but illegal accesses to the kernel data can be detected by checking for the replicas of the kernel data. The fourth level does not even replicate the kernel data, so illegal accesses to the kernel data cannot be detected at all. However, it still receives a benefit of the protection as a user process at least. Finally, at the fifth level, an extension module is embedded into the kernel without any protection. The extension module at this level is exactly the same with ones hand-crafted in the kernel from the beginning except the overheads for using the same API among all protection levels.

For comparison, we have also measured hand-crafted version of extension modules in the kernel. These extension modules are implemented as a part of the kernel of NetBSD.

5.3.1 File System Modules

We have developed two file system modules: Simple Memory File System (SMFS) and Simple Network File System (SNFS). SMFS is a RAM disk, whose files reside in memory. The block size that SMFS can read and write from the memory at a time is 512 bytes. SNFS is a simplified NFS [93], which consists of some clients and one server and communicates between the client and the server using Remote Procedure Call (RPC) [99]. RPC is executed using UDP and the block size that SNFS can read and write with RPC at a time is 512 bytes. The server reads and writes real files from a local file system UFS.

Micro Benchmark

We measured the time needed to copy a file on our file system. Since copying a file is one of the most fundamental operations to file systems, we can obtain the potential overheads of each of the protection techniques used in CAPELA. A file was copied from our file system to the same file system. The size of the copied file was 64KB and the block size for each `read` and `write` system call was 8KB. In SNFS, the client communicated with the server through a 10Mbps network.

Figure 5.14 and Figure 5.15 show the results of this experiment. These two figures mean that the performance of the file systems is improved when the protection level is lowered. The reason why the second level suffers larger overheads than the first level is probably that changing the protection of memory

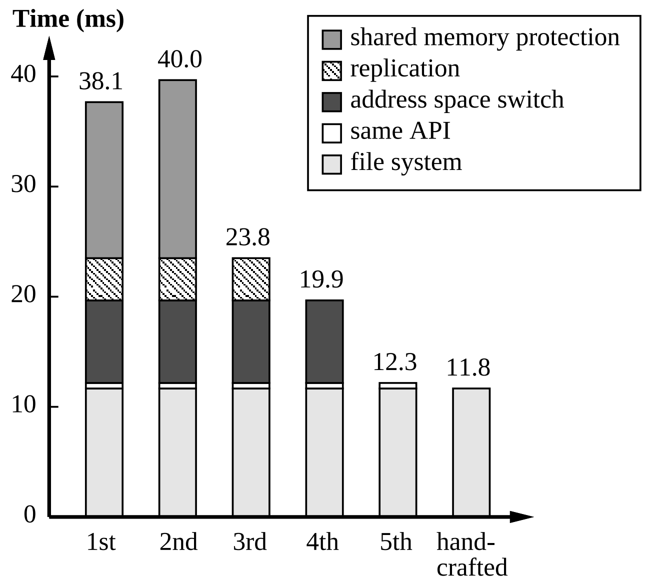


Figure 5.14: The time needed to copy a 64KB file on SMFS and the breakdown of the overheads.

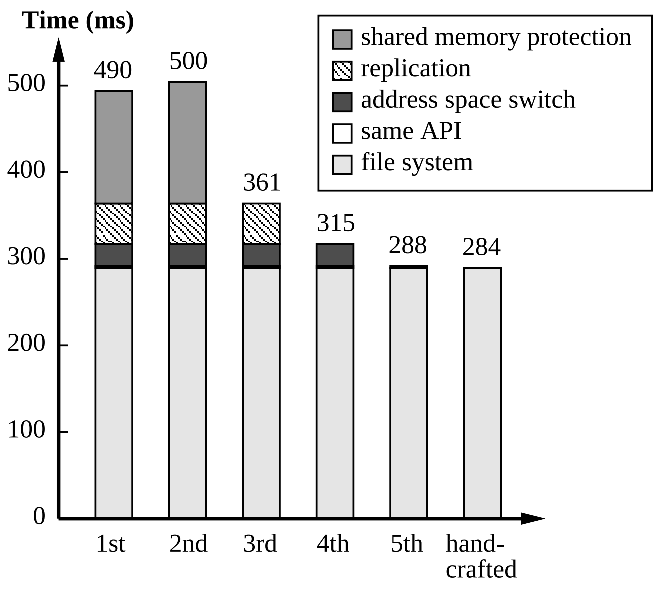


Figure 5.15: The time needed to copy a 64KB file on SNFS through a 10Mbps network and the breakdown of the overheads.

Table 5.8: The time needed to copy a 64KB file on SMFS in the SPARC architecture (*msec*).

	1st	2nd	3rd	4th	5th
SMFS/SPARC	504	372	155	90	38

Table 5.9: The ratio of the overheads to the fifth level in SMFS and SNFS.

	1st	2nd	3rd	4th	5th
SMFS	3.11	3.26	1.94	1.62	1.00
SNFS	1.70	1.74	1.25	1.09	1.00

pages often takes more time than unmapping memory pages. From this result, the second level is not useful in the Intel architecture at least. However, in the SPARC architecture, the second level is also meaningful. Table 5.8 shows the result of the same experiment in SPARCstation 5 (MicroSPARCII/85MHz). Since this result is a little old data, which is shown in our literature [55], the overheads of various protection techniques are larger than our latest result. But the second level is faster than the first level.

The ratio of the overheads of each protection level to the fifth level is described in Table 5.9. In SMFS, the first level is 211% slower than the fifth level; and in SNFS, the first level is 70% slower. These overheads are not small and the performance is rather degraded, but we think that it is acceptable for debugging the file systems. The overheads of the fifth level to the hand-crafted version are 3.7% and 1.4% in SMFS and SNFS, respectively, so they are almost negligible.

Macro Benchmark

Next, we measured the time needed to compile a small program on our file system. Since compiling a program is one of the most frequently used applications, we can obtain the realistic overheads of our approach. The program consists of five source files and two header files of the C language and had about 2,000 lines. We compiled the program using `gcc` 2.7.2.2 with no optimizing options. But temporary files were put on the local file system.

Figure 5.16 and Figure 5.17 show the results of this experiment. In practical circumstances like this experiment, the first level is 16% and 19% slower than the fifth level in SMFS and SNFS, respectively. The performance is good enough even for normal use. In addition, the overheads of the fifth level to the hand-crafted version are just 3.1% and 1.4% in SMFS and SNFS, respectively, and are enough small.

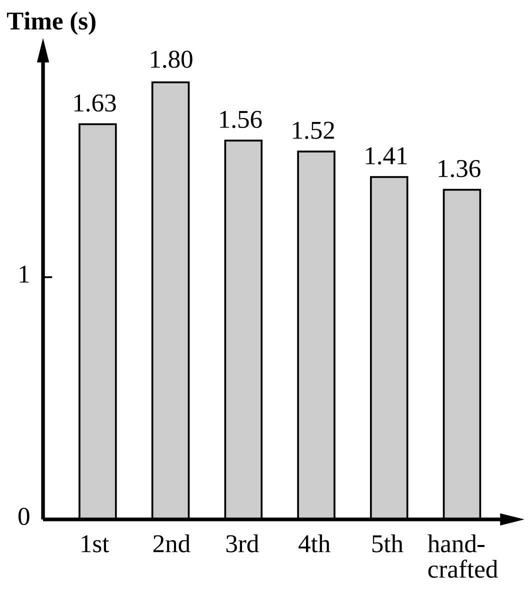


Figure 5.16: The time needed to compile `ps` program on SMFS.

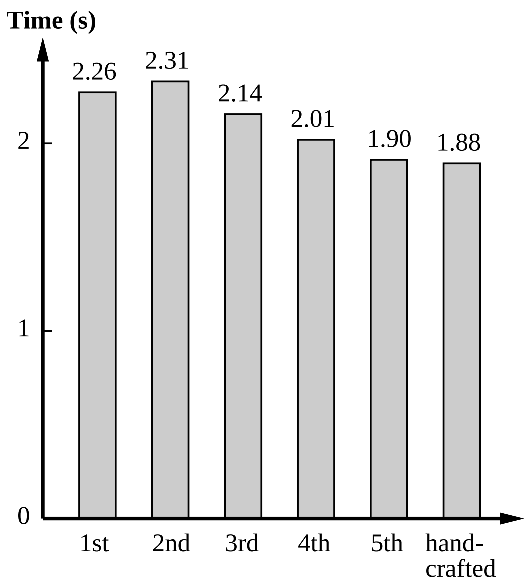


Figure 5.17: The time needed to compile `ps` program on SNFS through a 10Mbps network.

Table 5.10: The ratio of the overheads to the fifth level of round-trip latency in SUDP and STCP.

	1st	2nd	3rd	4th	5th
SUDP	2.82	2.79	1.68	1.56	1.00
STCP	3.20	2.72	1.55	1.39	1.00

5.3.2 Network System Modules

We have developed two network subsystem modules: Simple User Datagram Protocol (SUDP) and Simple Transmission Control Protocol (STCP). SUDP and STCP are exactly the same with UDP [80] and TCP [82, 18], respectively, except that control operations have not implemented.

Micro Benchmark: Round-Trip Latency

Round-trip latency reflects the overhead induced by a protocol when the protocol transfers a packet between two hosts. We sent a packet with data of 1 byte and measured the time needed from sending a packet to the other host to receiving a packet from the other host in order to obtain round-trip latency. We repeated sending and receiving a packet 1,000 times and divided the elapsed time by 1,000 to obtain an average round-trip latency.

Figure 5.18 and Figure 5.19 show the results of this experiment. These two figures mean that the performance of network subsystems is improved when the protection level is lowered. The ratio of the overheads of each protection level to the fifth level is described in Table 5.10. In SUDP, the first level is 182% slower than the fifth level; and in STCP, the first level is 220% slower. These overheads of the maximum protection are, we think, acceptable for debugging. The overhead of the fifth level to the hand-crafted version is 12% and 2.8%. It is considered that the overheads in SUDP are a little large because the send and input routines of SUDP are small and the overhead for using the same API in all protection levels relatively gets large. The improvement to this overhead is mentioned in Section 5.3.3.

Micro Benchmark: Throughput

Throughput is the other indicator to measure the execution performance of network subsystems. Throughput indicates how much data is sent using a protocol per unit time. We did not measure throughput for SUDP because it depends on the windowing and acknowledgment strategies. STCP uses the same strategies with TCP, but SUDP does not provide the strategies like UDP. We calculated throughput from the time needed from sending data of total 1MB to the other host to receiving all the data from the other host. The buffer size of both the `send` and `recv` system calls is 8KB.

Figure 5.20 shows the result of this experiment. Note that a large value means good performance in throughput. Throughput in each protection level

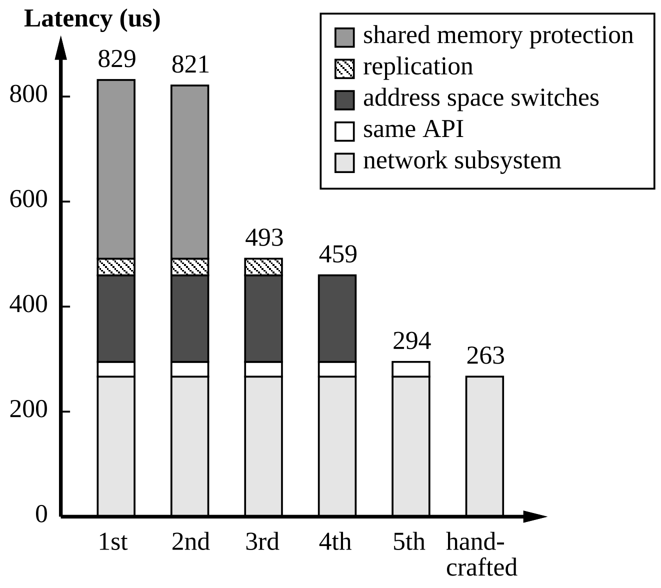


Figure 5.18: Round-trip latency in SUDP through a 10Mbps network and the breakdown of the overheads.

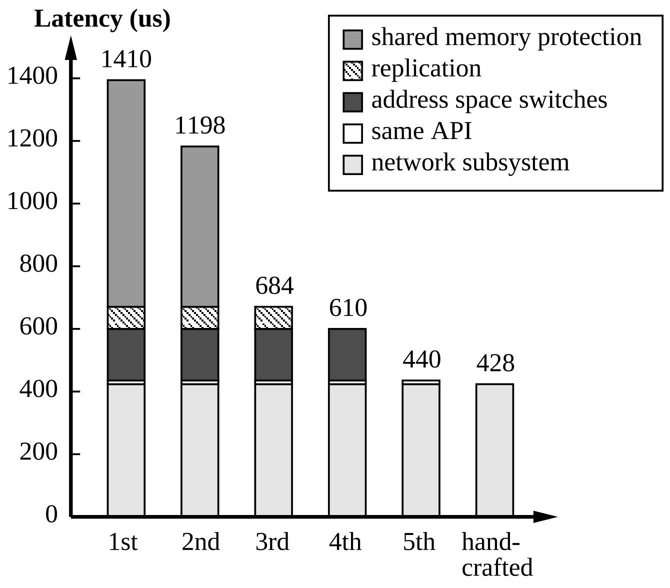


Figure 5.19: Round-trip latency in STCP through a 10Mbps network and the breakdown of the overheads.

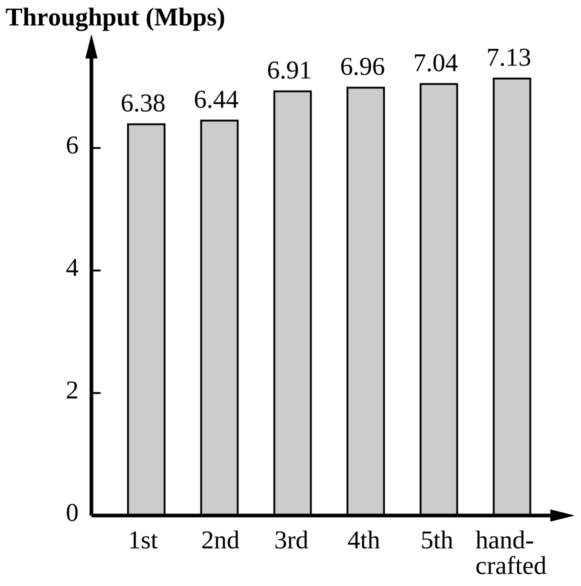


Figure 5.20: Throughput in STCP through a 10Mbps network.

is near the hand-crafted version. The overhead for the maximum protection is 10% and the overhead for using the same API in every protection level is only 1.3%. This good result is caused by the fact that the measurement of the throughput is more practical experiment since more data is transfered through slow network.

Macro Benchmark: SNFS with SUDP

We measured the performance of SNFS, which we have developed as a file system module, with SUDP instead of UDP. Network file systems are one of the most usual and important applications of the datagram protocol, so we can obtain practical overheads of SUDP from this experiment. We measured the time needed to copy a 64KB file on SNFS with SUDP. A file was copied from SNFS to the same SNFS and the block size for each `read` and `write` system call was 8KB. We experimented on SUDP of various protection levels using SNFS of the fifth level, which is embedded into the kernel.

Figure 5.21 shows the results of this experiment. Through a 10Mbps network, the first level of SUDP is 75% slower than the fifth level and the fifth level is 2.7% slower than the hand-crafted version. These overheads are enough small for the purpose of each protection level. This relatively worse result is caused by the fact that SNFS is a small application and processing in SNFS is less than that in SUDP.

We also measured for SNFS of the fourth level, which is running at the user

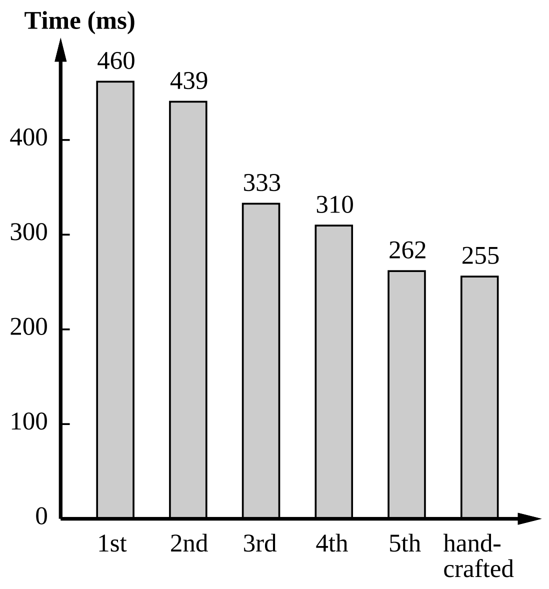


Figure 5.21: The time needed to copy a 64KB file on SNFS of the fifth level with SUDP through a 10Mbps network.

Table 5.11: The time needed to copy a 64KB file on SNFS of the fourth level with SUDP. (*msec*)

	1st	2nd	3rd	4th	5th	Hand-crafted
SNFS/4	517	526	400	351	297	290

level. The result is shown in Table 5.11. From this result, it is confirmed that the interaction between the two user-level modules can work properly and that the overheads do not increase dramatically.

Macro Benchmark: FTP with STCP

We measured the file transfer rate using FTP [83] with STCP instead of TCP. FTP is one of the most frequently used applications of the data-stream protocol, so we can obtain practical overheads of STCP from this experiment. We received a 1MB file by a `get` command of FTP and measured the transfer rate.

The results are shown in Figure 5.22. Note that a large value means good performance in the transfer rate. Like the throughput of STCP, the impact due to the overheads for the maximum protection level and for using the same API are small when a file is transfered through a 10Mbps network. The transfer rate of the first level is 16% lower than that of the fifth level while the transfer rate

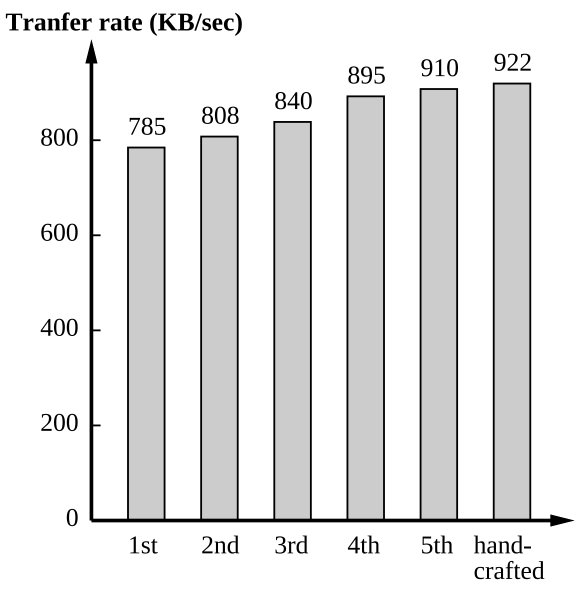


Figure 5.22: The file transfer rate of FTP with STCP through a 10Mbps network.

of the fifth level is only 1.3% lower than that of the hand-crafted version.

5.3.3 Improvement by Source Code Translation

At the fifth level, which is the lowest protection level, the overheads of some of our extension modules are not enough small due to the overheads for using the same API and for modifying no binary codes of the extension modules among protection levels. The details of the overheads are extra memory allocation for creating instances of classes at runtime, dispatches of virtual functions of the C++ language, extra function calls by the fact that inline extraction is impossible, and so on.

If the extension modules are allowed to modify their source code and recompile them, these overheads would be reduced. To automatically modify the source code, translating source code is generally used. Macro is very helpful to translate source code for programs of the C language, but it is not enough powerful for C++ programs because C++ method invocations are difficult to be translated by macro. OpenC++ [22] is useful to translate source codes of C++ programs. Using OpenC++, we can translate C++ classes of higher abstraction to the kernel data structure of low abstraction. This reduces the overheads of translating between them at runtime. For example, suppose the following code fragment.

```
int Smfs::write(Vnode* vp, Uio* uio, int ioflag, Ucred* cred)
{
    uio->bulkWrite(vp, DEV_BSIZE, size, cred);
}

int Uio::bulkWrite(Vnode* vp, int blksize, int filesize,
                   Ucred* cred)
{
    ...
}
```

This code fragment is translated as the below.

```
int Smfs::write(struct vnode* vp, struct uio* uio, int ioflag,
                struct ucred* cred)
{
    Uio_bulkWrite(uio, vp, DEV_BSIZE, size, cred);
}

int Uio_bulkWrite(struct uio* uio, struct vnode* vp, int blksize,
                  int filesize, struct ucred* cred)
{
    ...
}
```

We have translated the source code of the SUDP module and the SMFS module by hand. For the SUDP module, we measured the round-trip latency when sending 1 byte packet through a 10Mbps network. For the SMFS module, we measured the time needed to copy a 64KB file.

The results are shown in Figure 5.23 and Figure 5.24. The translated SUDP module is 5.4% faster than SUDP of the fifth level and the overhead is 6.1% comparing with the hand-crafted version. For the SMFS module, the translated module is 10% faster than the hand-crafted version. It is considered that this is due to CPU caches. In fact, when L1 and L2 caches of CPU are not used, the hand-crafted SMFS module is 4% faster than the translated module. These results indicate that source code translation makes the extension modules almost as efficient as hand-crafted versions even if the module size is too small.

5.3.4 Automatic Module Distribution

When two communicating hosts dynamically negotiate the protocol number for a non-standard protocol using DPNAP, extra processing is needed when a host sends and receives packets. At sending packets, CAPELA must check if it has already sent a *NOTIFY* message of DPNAP. At receiving packets, CAPELA must refer to the translation table in order to examine which protocol is used for the packet. We measured the round-trip latency of two kinds of Null Ethernet Protocol (NEP) modules, which we have developed. One is assigned a

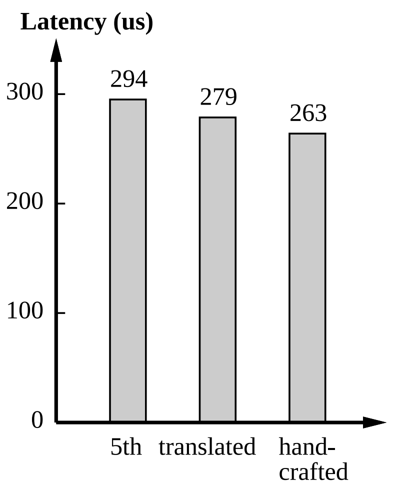


Figure 5.23: Comparison of the latency in SUDP after source code translation.

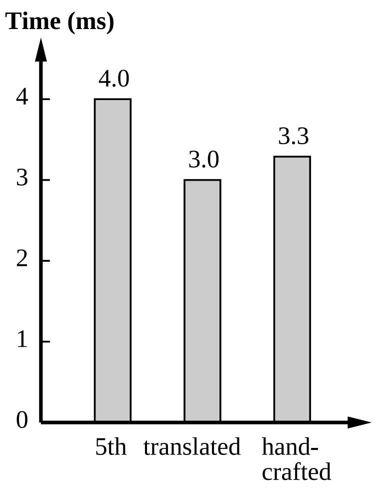


Figure 5.24: Comparison of the time needed to copy a 64KB file on SMFS with all caches hot after source code translation.

Table 5.12: Round-trip latency of f-NEP whose protocol number is fixed and d-NEP whose protocol number is dynamic. (μs)

Protocol	10Mbps	100Mbps
f-NEP	120.5	62.8
d-NEP	121.2	63.9

fixed protocol number (f-NEP) and the other is dynamically assigned a protocol number (d-NEP). NEP is a protocol specialized in a case where two hosts connected in the same Ethernet segment communicate with each other. NEP sends packets directly to an Ethernet device driver and receives packets directly from the device driver. NEP is similar to UDP except a point that it cannot treat packets of large size exceeding the limits of an Ethernet device driver.

Table 5.12 shows round-trip latency of two NEPs. The size of data sent in this experiment was 1 byte. d-NEP suffers overhead of 0.6% in 10Mbps and 1.8% in 100Mbps, comparing with f-NEP. This overhead is almost negligible.

If CAPELA needs to load a network module from a remote host, packet handling is delayed until a necessary module is loaded to the system. When CAPELA loads a NEP module (4,683 bytes) with 10Mbps Ethernet, the delay reached to $250ms$. This delay is very large comparing with the round-trip latency of $0.1ms$ of NEP. However, the time taken for sending a module via network was only $5ms$. Most of that delay is due to saving the received module on a local disk and loading the module from the disk to the system.

5.4 Summary

This chapter mainly presented the results of our experiments on process cleaning and the multi-level protection. From our micro benchmark of process cleaning, it was confirmed that saving and restoring a memory image get large overheads and the overheads increase linearly depending on the amount of memory to be restored. Also, our remap strategy was faster than the copy strategy at restoration time but was slower in total because of extra page faults. Comparing the fork-join method, the overheads of the system calls used by process cleaning were smaller than those of the fork-join method.

To obtain practical overheads of process cleaning, we measured the performance of the Apache web server, which uses a process pool technique. A web server using process cleaning was 35% slower than one using no process cleaning. However, The web server using process cleaning was 30-60% faster than one using the fork-join method. The improvements due to process cleaning and process pool were the same degree on average. For optimization, the copy strategy was 5% faster than the remap strategy in this case. Optimizing the data layout of the server program made the server 40% faster. We also measured the performance of a FastCGI module of Apache and obtained the similar results.

For the multi-level protection, we measured the overheads of that mechanism on our file system modules and network subsystem modules. In terms of a file copy and round-trip latency, the modules of the maximum protection level were 200% slower than those of the minimum protection level at worst. The modules of the minimum protection level were only 4-12% slower than the hand-crafted ones embedded in the kernel from the beginning. In terms of the throughput of our network subsystem, on the other hand, the overhead was 10% at maximum since this is a more practical application of network modules.

As macro benchmarks to obtain practical overheads, we measured the cost of compiling a program and transferring data using FTP. In most cases, the overheads of the maximum protection level were 20% at worst and thus are enough small even for normal use. In terms of our NFS module with our network subsystem, on the other hand, the overhead of the maximum protection level was 75%. The reason why this overhead is relatively large is that our network subsystem is overloaded as well as in the micro benchmarks if it is used with the NFS module.

Chapter 6

Conclusion

This dissertation discussed a dynamic safety net for server software. To achieve the principle of least privilege for user-level servers, it is indispensable to dynamically change a safety net of them so that the safety net fits the client. For operating system modules, it is crucial to protect the whole system from the instability of the modules. Since many modules are also performance critical, it is important that the users can select fail-safety and performance depending on their purpose.

Contributions

Contributions by this dissertation are as follows:

- For user-level servers, process cleaning enables a server process to securely remove the access restrictions even if the process is compromised. Using process cleaning, the operating system cleans up the state of a process including an instruction pointer and a memory image and recovers a hijacked process or a process into which Trojan horse code is injected.

- Process cleaning is faster than the traditional fork-join method, which can achieve similar security effects. While a server process must create and destroy a child process for each request in the fork-join method, a server process is reused in process cleaning. In addition, process cleaning is used together with the process pool technique, which makes the server handle requests in parallel.

- For operating system modules, the multi-level protection enables the users to make a trade-off between fail-safety and performance. The users can change the trade-off transparently by selecting an appropriate one from multiple protection managers, each of which provides a different protection level. To differentiate multiple protection levels, each protection manager has a different ability to detect and recovery from errors.

- When the protection level of modules is minimum, the overheads due to this fail-safe mechanism itself are enough small to use them as product release. The modules are running in the kernel address space and the overheads of the protection manager are reduced to the minimum as well as modules embedded into the kernel by hand.

Future Directions

One of our future research directions is to implement middleware like CORBA [75] using process cleaning. It may not be easy since some kinds of CORBA runtimes cannot be implemented with our access control mechanism. For example, Orbix [46] and omniORB [59] have multi-threaded runtimes, which allocate a new thread for every request. On the other hand, in our system, access control and process cleaning are performed by the unit of process. It is relatively easy to extend the unit of access control to a thread, but the current design of process cleaning does not allow to selectively restore part of the process state used by a particular thread. To work these CORBA runtimes with our access control mechanism, they must be modified not to use multi-threads. The data shared among threads must be moved to the memory shared among processes, which an attacker may use for hijacking the whole CORBA server. Since our system does not protect CORBA runtimes from attacks exploiting this shared memory, the programmers of CORBA runtimes must be responsible for that there is no vulnerability around the shared memory.

The other approach for multi-threaded middleware is to extend access restrictions and process cleaning to be a per thread basis. However, it is doubtful whether this extension is achieved securely and is useful even if it is secure. A pure thread is not protected from the other threads in the same process and thus access control per thread is useless. It is easy that a thread hijacks the execution of another thread. Theoretically, it is possible to protect a thread from the other threads in a lightweight manner using proposed techniques [47, 100, 23], but most global variables must be still shared between threads and these are not secure. Moreover, the cost of thread creation and destruction is still expected to be high since the operating system must manage the memory page table and purge memory pages and caches at destruction. Even if a fast and protected thread could be implemented, it is unknown whether the combination of a thread pool and cleaning of a thread is faster than the fork-join method using a protected thread.

To detect more sensitive errors of extension modules, it is hopeful to provide various protection level at the kernel level as well as at the user level. In the current implementation, the extension modules embedded into the kernel run without any protection. Since the differences between the user level and the kernel level are large, it is useful for debugging to enable the users to change the protection level at the kernel level. If the extension modules run with the least protection in the kernel, the fail-safe mechanism may be able to detect timing-critical errors including deadlocks. To achieve this facility, we can use the efforts of the extensible operating systems that protect extension modules at the kernel level such as VINO. However, the fail-safe mechanism used in such operating systems tends to suffer large overheads in some applications. For example, the overhead of software fault isolation used in VINO is 200% at maximum.

A more difficult direction is to indicate guidelines of time when the users change the protection level of extension modules. If the users lower the protection level in spite of an unstable module, the module would make the whole system crash. But if the users do not change the protection level in spite of a stable module, the system performance would not be improved. We believe that statistical information on a stabilization process of extension modules is useful. In fact, we usually regard programs in which critical software flaws are not found for a long time as stable. Statistics is a good indication for extension modules as well although the decision of changing the protection level requires more carefulness.

Bibliography

[1] Accetta, M., R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," in *Proceedings of the USENIX 1986 Summer Conference*, pp. 93–112, June 1986.

[2] Acharya, A. and M. Raje, "MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications," in *Proceedings of the 9th USENIX Security Symposium*, Aug. 2000.

[3] Anderson, D., T. Frivold, and A. Valdes, "Next-Generation Intrusion-Detection Expert System (NIDES)," Tech. Rep. SRI–CSL–95–07, Computer Science Laboratory, SRI International, May 1995.

[4] Apache HTTP Server Project, "Apache HTTP Server." `http://www.apache.org/`.

[5] Armand, F., "Give a Process to your Drivers!," in *Proceedings of the EurOpen Autumn 1991 Conference*, Sep. 1991.

[6] Asaka, M., A. Taguchi, and S. Goto, "The Implementation of IDA: An Intrusion Detection Agent System," in *Proceedings of the 11th FIRST Conference*, June 1999.

[7] AusCERT, "Vulnerability in NCSA/Apache CGI Example Code." AusCERT Advisory AA-96.01.

[8] Badger, L., D. D. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat, "A Domain and Type Enforcement UNIX Prototype," in *Proceedings of the 5th USENIX Security Symposium*, June 1995.

[9] Badger, L., D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat, "Practical Domain and Type Enforcement for UNIX," in *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, May 1995.

[10] Baratloo, A., T. Tsai, and N. Singh, "Transparent Run-Time Defense Against Stack Smashing Attacks," in *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[11] Berman, A., V. Bourassa, and E. Selberg, “TRON: Process-Specific File Protection for the UNIX Operating System,” in *Proceedings of the USENIX Technical Conference*, 1995.

[12] Bershad, B. N., T. E. Anderson, E. D. Lazowska, and H. M. Levy, “Lightweight Remote Procedure Call,” *ACM Transactions on Computer Systems*, vol. 8, pp. 37–55, Feb. 1990.

[13] Bershad, B. N., S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, S. Chambers, and C. Eggers, “Extensibility, Safety and Performance in the SPIN Operating System,” in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 267–284, Dec. 1995.

[14] Biba, K. J., “Integrity Considerations for Secure Computer Systems,” Tech. Rep. ESD–TR–76–372, USAF Electronic Systems Divison, Hanscom Air Force Base, 1977.

[15] Bobrow, D. G., J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, “TENEX, a Paged Time Sharing System for the PDP-10,” *Communication of ACM*, vol. 15, pp. 1135–1143, Mar. 1972.

[16] Boebert, W. E. and R. Y. Kain, “A Practical Alternative to Hierarchical Integrity Policies,” in *Proceedings of the 8th National Computer Security Conference*, 1985.

[17] Bryce, C. and G. Muller, “Matching Micro-Kernels to Modern Applications Using Fine-Grained Memory Protection,” in *Proceeding on the 7th IEEE Symposium on Parallel and Distributed Processing*, pp. 272–279, 1995.

[18] Cerf, V. and R. Kahn, “A Protocol for Packet Network Intercommunication,” *IEEE Transactions on Communications*, vol. 22, pp. 637–648, May 1974.

[19] CERT, “Sendmail Daemon Mode Vulnerability.” CERT Advisory CA-96.24.

[20] Chen, J. and B. N. Bershad, “The Impact of Operating System Structure on Memory System Performance,” in *Proceedings of the 14th Symposium on Operating System Principles*, pp. 120–133, Dec. 1993.

[21] Cheriton, D. R. and K. J. Duda, “Logged Virtual Memory,” in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 26–39, Dec. 1995.

[22] Chiba, S., “A Metaobject Protocol for C++,” in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 285–299, Oct. 1995.

[23] Chiueh, T., G. Venkitachalam, and P. Pradhan, "Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions," in *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pp. 140–153, Dec. 1999.

[24] Chou, A., J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating Systems Errors," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pp. 73–88, Oct. 2001.

[25] Clark, D. D. and D. R. Wilson, "A Comparison of Commercial and Military Security Policies," in *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pp. 184–194, Apr. 1987.

[26] Cowan, C., S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor, "SubDomain: Parsimonious Server Security," in *Proceedings of the 14th USENIX Systems Administration Conference*, Dec. 2000.

[27] Cowan, C., C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," in *Proceedings of the 7th USENIX Security Symposium*, pp. 63–78, Jan. 1998.

[28] Cowan, C., P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," in *Proceedings of the DARPA Information Survivability Conference and Expo*, Jan. 2000.

[29] Denning, D. E., "An Intrusion-Detection Model," *IEEE Transactions on Software Engineering*, vol. SE–13, pp. 222–232, Feb. 1987.

[30] Engler, D. and M. F. Kaashoek, "DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation," in *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pp. 53–59, Aug. 1996.

[31] Engler, D. R., M. F. Kaashoek, and J. O'Toole Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management," in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 251–266, Dec. 1995.

[32] Etoh, H. and K. Yoda, "GCC Extension for Protecting Applications from Stack-Smashing Attacks." `http://www.trl.ibm.com/projects/security/ssp/`, 2000.

[33] Ferraiolo, D. and R. Kuhn, "Role-Based Access Controls," in *Proceedings of the 15th NIST-NCSC National Computer Security Conference*, pp. 554–563, 1992.

[34] Ford, B. and J. Lepreau, "Evolving Mach 3.0 to a Migrating Thread Model," in *Proceedings of the 1994 USENIX Conference*, pp. 97–114, 1994.

[35] Fraser, T., “LOMAC: Low Water-Mark Integrity Protection for COTS Environments,” in *Proceedings of the 2000 IEEE Symposium on Securtiy and Privacy*, May 2000.

[36] Goldberg, I., D. Wagner, R. Thomas, and E. A. Brewer, “A Secure Enviroment for Unstrusted Helper Applications,” in *Proceedings of the 6th USENIX Security Symposium*, July 1996.

[37] Golub, D., R. Dean, A. Forin, and R. Rashid, “Unix as an Application Program,” in *Proceedings of the Summer 1990 USENIX Conference*, pp. 87–95, June 1990.

[38] Gosling, J. and H. McGilton, “The Java Language Environment.” `http://www.javasoft.com/docs/white/langenv/`, May 1996.

[39] Guedes, P. and D. Julin, “Object-Oriented Interfaces in the Mach 3.0 Multi-Server System,” in *Proceedings of IEEE Second International Workshop on Object Orientation in Operating Systems*, Oct. 1991.

[40] Härtig, H., M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter, “The Performance of μ-Kernel-Based Systems,” in *Proceedings of the 16th Symposium on Operating Systems Principles*, pp. 66–77, Oct. 1997.

[41] Heberlein, L. T., G. V. Dias, K. N. Levitt, B. Mukherjee, J. Wood, and D. Wolber, “A Network Security Monitor,” in *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy*, pp. 296–304, 1990.

[42] Hickman, K. E. B. and T. ElGamal, “The SSL Protocol,” tech. rep., Netscape Communications Corp., Jun. 1995.

[43] Housley, R., W. Ford, W. Polk, and D. Solo, “Internet X.509 Public Key Infrastructure Certificate and CRL Profile.” RFC 2459, 1999.

[44] Hsieh, W. C., M. E. Fiuczynski, C. Garrett, S. Savage, D. Becker, and B. N. Bershad, “Language Support for Extensible Systems,” in *First Workshop on Compilr Support for Systems Software*, Feb. 1996.

[45] IBM, “Lotus Notes.” `http://www.lotus.com/`.

[46] Iona Technologies, “Orbix.”

[47] Isa, H. R., W. R. Shockley, and C. E. Irvine, “A Multi-Threading Architecture for Multilevel Secure Transaction Processing,” in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 166–180, May 1999.

[48] Jones, R., “Netperf Benchmark.” `http://www.netperf.org/netperf/NetperfPage.html`.

[49] Kaashoek, M. F., D. R. Engler, G. R. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie, "Application Performance and Flexibility on Exokernel Systems," in *Proceedings of the 16th Symposium on Operating Systems Principles*, pp. 52–65, Oct. 1997.

[50] Kent, S. and R. Atkinson, "IP Authentication Header (AH)." RFC 2402, 1998.

[51] Kent, S. and R. Atkinson, "IP Encapsulating Security Payload (ESP)." RFC 2406, 1998.

[52] Kent, S. and R. Atkinson, "Security Architecture for the Internet Protocol." RFC 2401, 1998.

[53] Kourai, K. and S. Chiba, "A Secure Access Control Mechanism against Internet Crackers," in *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems*, pp. 743–746, Apr. 2001.

[54] Kourai, K. and S. Chiba, "A Secure Mechanism for Changing Access Restrictions of Servers," *Journal of Information Processing Society of Japan*, vol. 42, pp. 1492–1502, June 2001.

[55] Kourai, K., S. Chiba, and T. Masuda, "Multi-Level Protection: A New Fail-Safe Mechanism for Extensible Operating Systems," *Journal of Information Processing Society of Japan*, vol. 39, pp. 3054–3064, Nov. 1998.

[56] Kourai, K., S. Chiba, and T. Masuda, "Operating System Support for Easy Development of Distributed File Systems," in *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Systems*, pp. 551–554, Oct. 1998.

[57] Li, K., J. F. Naughton, and J. S. Plank, "Real-Time Concurrent Checkpoint for Parallel Programs," in *Proceedings of the 2th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 79–88, Mar. 1990.

[58] Liedtke, J., "On μ-Kernel Construction," in *Proceedings of the 15th ACM Symposium on Operating System Principles*, pp. 237–250, Dec. 1995.

[59] Lo, S. and S. Pope, "The Implementation of a High Performance ORB over Multiple Network Transports," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pp. 157–172, 1998.

[60] Lunt, T., A. Tamaru, F. Gilham, R. Jagannathan, P. Neumann, H. Javitz, A. Valdes, and T. Garvey, "A Real-Time Intrusion Detection Expert System (IDES) - final technical report," tech. rep., Computer Science Laboratory, SRI International, 1992.

[61] Madsen, D., "An Operating System Analog to the Perl Data Tainting Functionality," in *Proceedings of the 23rd National Information Systems Security Conference*, October 2000.

[62] Maeda, C. and B. N. Bershad, "Protocol Service Decomposition for High-Performance Networking," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pp. 244–255, Dec. 1993.

[63] McCanne, S. and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-Level Packet Capture," in *Proceedings of the Winter 1993 USENIX Conference*, pp. 259–269, Winter 1993.

[64] McKusick, M. K., "The Virtual Filesystem Interface in 4.4BSD," *Computing Systems*, vol. 8, pp. 3–25, Winter 1995.

[65] Mindcraft, "WebStone Benchmark." `http://www.mindcraft.com/webstone/`.

[66] Mogul, J. C., R. F. Rashid, and M. J. Accetta, "The Packet Filter: an Efficient Mechanism for User-Level Network Code," in *Proceedings of the 11th Symposium on Operating System Principles*, pp. 39–51, Nov. 1987.

[67] Muquit, M. A., "WWW Homepage Access Counter and Clock." `http://www.muquit.com/muquit/software/Count/Count.html`.

[68] Necula, G. C., "Proof-Carrying Code," in *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Jan. 1997.

[69] Necula, G. C. and P. Lee, "Safe Kernel Extensions without Run-Time Checking," in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, Oct. 1996.

[70] Necula, G. C., S. McPeak, and W. Weimer, "CCured: Type-Safe Retrofitting of Legacy Code," in *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, Jan. 2002.

[71] Nelson, G., *System Programming with Modula-3*. Prentice Hall, 1991.

[72] `http://www.netbsd.org`.

[73] NetBSD Security Alert Team, "at(1) Vulnerabilities." NetBSD Security Advisory NetBSD-SA1998-004.

[74] NIST, "Secure Hash Standard." FIPS PUB 180–1, 1995.

[75] Object Management Group, "Common Object Request Broker Architecture Specification 2.2." `http://www.omg.org/`.

[76] OpenSSL Project, "OpenSSL." `http://www.openssl.org/`.

[77] Openwall Project, "Non-Executable User Stack." `http://www.openwall.com/linux/`.

[78] Porras, P., “STAT – A State Transition Analysis Tool For Intrusion Detection,” Master’s thesis, Computer Science Department, University of California, Jun. 1992.

[79] Porras, P. and P. Neumann, “EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances,” in *Proceedings of the 20th National Information Systems Security Conference*, Oct. 1997.

[80] Postel, J., “User Datagram Protocol.” RFC 768, Aug. 1980.

[81] Postel, J., “Internet Protocol.” RFC 791, Sep. 1981.

[82] Postel, J., “Transmission Control Protocol.” RFC 793, Sep. 1981.

[83] Postel, J. and J. Reynolds, “File Transfer Protocol (FTP).” RFC 959, Oct. 1985.

[84] Postel, J., C. Sunshine, and D. Cohen, “The ARPA Internet Protocol,” *Computer Networks*, vol. 5, pp. 261–271, July 1981.

[85] Povey, D., “Enforcing Well-Formed and Partially-Formed Transactions for Unix,” in *Proceedings the 8th USENIX Security Symposium*, Aug. 1999.

[86] Rashid, R., A. Tevanian, M. Young, D. Golub, and R. Baron, “Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures,” in *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 31–39, Oct. 1987.

[87] Reynolds, F. and J. Heller, “Kernel Support for Network Protocol Servers,” in *Proceedings of the USENIX Mach Symposium*, pp. 149–162, Nov. 1991.

[88] Rivest, R., “The MD5 Message-Digest Algorithm.” RFC 1321, 1992.

[89] Rozier, M., V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser, “Overview of the Chorus Distributed Operating System,” in *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pp. 39–69, Apr. 1992.

[90] Rozier, M., V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser, “CHORUS Distributed Operating System,” *Computing Systems*, vol. 1, pp. 305–370, Dec. 1988.

[91] Saccoccio, R., “FastCGI.” `http://www.fastcgi.com/`.

[92] Saltzer, J. H. and M. D. Schroeder, “The Protection of Information in Computer Systems,” *Proceedings of the IEEE*, vol. 63, pp. 1278–1308, Sep. 1975.

[93] Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem," in *Proceedings of the USENIX 1985 Summer Conference*, pp. 119–130, June 1985.

[94] Satyanarayanan, M., J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West, "The ITC Distributed File System: Principles and Design," in *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pp. 35–50, Dec. 1985.

[95] Satyanarayanan, M., J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Transactions on Computers*, vol. 39, pp. 447–459, Apr. 1990.

[96] Seltzer, M. I., Y. Endo, C. Small, and K. A. Smith, "An Introduction to the Architecture of the VINO Kernel," Tech. Rep. TR–34–94, Harvard University Computer Science, 1994.

[97] Seltzer, M. I., Y. Endo, C. Small, and K. A. Smith, "Dealing With Disaster: Surviving Misbehaved Kernel Extensions," in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pp. 213–227, Oct. 1996.

[98] Small, C., "MiSFIT: A Minimal i386 Software Fault Isolation Tool," Tech. Rep. TR–07–96, Harvard University Computer Science, 1996.

[99] Sun Microsystems, Inc., *Remote Procedure Call Protocol Specification*, Feb. 1986.

[100] Takahashi, M., K. Kono, and T. Masuda, "Efficient Kernel Support of Fine-Grained Protection Domains for Mobile Code," in *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pp. 64–73, June 1999.

[101] Tanenbaum, A. S. and R. Renesse, "Distributed Operating Systems," *ACM Computing Surveys*, vol. 17, pp. 419–470, Dec. 1985.

[102] Thekkath, C. A., T. D. Nguyen, E. Moy, and E. D. Lazowska, "Implementing Network Protocols at User Level," in *Proceedings of the ACM SIGCOMM'93 Symposium on Communications Architectures and Protocols*, pp. 64–73, Sep. 1993.

[103] Wagner, D., J. Foster, E. Brewer, and A. Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," in *Proceedings of the Network and Distributed Systems Security Symposium*, pp. 3–17, Feb. 2000.

[104] Wahbe, R., S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient Software-Based Fault Isolation," in *Proceedings of the 14th Symposium on ACM Operating Systems Principles*, pp. 203–216, Dec. 1993.

[105] Yahoo!, “Yahoo! Calendar.” `http://calendar.yahoo.co.jp/`.

Appendix A

Security Policy Rules

A.1 Policy Rule Format

This section shows the limitable system calls and how to describe a policy rule for them. (The following tables are partial.)

A.1.1 File Management Rules

- `chdir` *dir*
  Allow or deny to change the current directory to a directory specified by *dir*.

- `chmod` *path* `with` *mode*
  Allow or deny to change the access permission of a file specified by *path* to *mode*.

- `chown` *path* `with` *uid*`:`*gid*
  Allow or deny to change the owner and group of a file specified by *path* to a user and group specified by *uid* and *gid*.

- `chroot` *path*
  Allow or deny to change the root directory to *path*.

- `close` *path*
  Allow or deny to close a file specified by *path*.

- `creat` *path* `with` *mode*
  Allow or deny to create a file specified by *path* with file creation mask *mode*.

- `fcntl` *path* `with` *cmd*
  Allow or deny to perform a file control operation specified by *cmd*.

- `flock` *path*
  Allow or deny to lock a file specified by *path*.

- `getdents` *dir*
  Allow or deny to get entries of a directory specified by *dir*.

- `ioctl` *path* `with` *req*
  Allow or deny to control a device specified by *path* with a request *req*.

- `link` *from* `to` *to*
  Allow or deny to link (hard link and symbolic link) from *from* to *to*.

- `mknod` *path* `with` *mode* `for` *major*`:`*minor*
  Allow or deny to create a character or block special file specified by *path* with access permission *mode*. *Major* and *minor* specify the device device.

- `mkdir` *path* `with` *mode*
  Allow or deny to create a directory specified by *path* with access permission *mode*.

- `open` *path* `with` *flags*
  Allow or deny to open a file, directory, or device specified by *path* with read-only, read-write, or append specified by *flags*.

- `read` *path*
  Allow or deny to read from a file descriptor corresponding to a file specified by *path*. This rule is applied only to `read` related to file systems, not network.

- `rename` *from* `to` *to*
  Allow or deny rename a file name from *from* to *to*.

- `rmdir` *path*
  Allow or deny to delete a directory specified by *path*.

- `truncate` *path*
  Allow or deny to truncate the length of a file specified by *path*.

- `umask`
  Allow or deny to change the file creation mask.

- `unlink` *path*
  Allow or deny to delete a file specified by *path*.

- `utime` *path*
  Allow or deny to change access and modification times of a file specified by *path*.

- `write` *path*
  Allow or deny to write to a file descriptor corresponding to a file specified by *path*. This rule is applied only to `write` related to file systems.

- `APPEND` *path*
  This is a macro for read-only access and is extracted to the following 3 rules:
  - `open` *path* `with O_WRONLY|O_APPEND`
  - `write` *path*
  - `close` *path*
- `READ_ONLY` *path*
  This is a macro for read-only access and is extracted to the following 3 rules:
  - `open` *path* `with O_RDONLY`
  - `read` *path*
  - `close` *path*
- `READ_WRITE` *path*
  This is a macro for read-only access and is extracted to the following 4 rules:
  - `open` *path* `with O_RDWR`
  - `read` *path*
  - `write` *path*
  - `close` *path*
- `WRITE_ONLY` *path*
  This is a macro for read-only access and is extracted to the following 3 rules:
  - `open` *path* `with O_WRONLY`
  - `write` *path*
  - `close` *path*

A.1.2 Network Management Rules

- `accept` *addr[/mask][:port]* `from` *addr2[/mask2][:port2]*
  Allow or deny to accept a connection from the peer process with *addr2/mask2:port2* to this process with *addr/mask:port*.
- `accept_unix` *path*
  Allow or deny to accept a connection from the peer process using a UNIX domain socket specified by *path*.
- `closesock` *addr[/mask][:port]*
  Allow or deny to close the socket bound to *addr/mask:port*.

- `closesock_unix` *path*
  Allow or deny to close a UNIX domain socket specified by *path*.

- `connect` *addr[/mask][:port]*
  Allow or deny to connect to a peer process whose network address is included in *addr* with netmask *mask* and whose port is *port*.

- `connect_unix` *path*
  Allow or deny to connect to a peer process using a UNIX domain socket specified by *path*.

- `protocol` *proto*
  Allow or deny to create a socket using a protocol specified by *proto*. *Proto* is `tcp`, `udp`, or `icmp`.

- `recv` *addr[/mask][:port]* `from` *addr2[/mask2][:port2]*
  Allow or deny to receive data from the peer process with *addr2/mask2:port2* to this process with *addr/mask:port*. This rule is also applied to `read` related to network.

- `recv_unix` *path*
  Allow or deny to receive data from the peer process using a UNIX domain socket specified by *path*.

- `send` *addr[/mask][:port]* `to` *addr2[/mask2][:port2]*
  Allow or deny to send data from this process with *addr/mask:port* to the peer process with *addr2/mask2:port2*. This rule is also applied to `write` related to network.

- `send_unix` *path*
  Allow or deny to send data to the peer process using a UNIX domain socket specified by *path*.

- `setsockopt` *opt*
  Allow or deny to set a socket option specified by *opt*.

- `semctl`
  Allow or deny semaphore control operations.

- `semop`
  Allow or deny semaphore operations.

- `msgrcv`
  Allow or deny to receive IPC messages.

- `msgsnd`
  Allow or deny to send IPC messages.

- `msgctl`
  Allow or deny IPC message control operations.

- `shmat`
  Allow or deny to map SYSV shared memory segments.
- `shmdt`
  Allow or deny to unmap SYSV shared memory segments.
- `shmctl`
  Allow or deny to control SYSV shared memory.
- `COMM` *addr[/mask][:port]* `with` *addr2[/mask2][:port2]*
  This is a macro for a server side process to use Internet communication and is extracted to the following 4 rules:
  - `accept` *addr/mask:port* `from` *addr2/mask2:port2*
  - `recv` *addr/mask:port* `from` *addr2/mask2:port2*
  - `send` *addr/mask:port* `to` *addr2/mask2:port2*
  - `closesock` *addr/mask:port*
- `COMM_UNIX` *path*
  This is a macro for a server side process to use a UNIX domain socket and is extracted to the following 4 rules:
  - `accept_unix` *path*
  - `recv_unix` *path*
  - `send_unix` *path*
  - `closesock_unix` *path*

A.1.3 Process Management Rules

- `exec` *path*
  Allow or deny to execute a new program specified by *path*. The program cannot be examined using the string but the calculated SHA-1 value.
- `exit`
  Allow or deny to exit the process.
- `ioperm`
  Allow or deny to set the permissions of I/O ports.
- `kill` *sig* `for` *progname*
  Allow or deny to send a signal specified by *sig* to a process specified by *progname*.
- `nice`
  Allow or deny to change process priority to higher.
- `ptrace`
  Allow or deny to trace system calls of the process.

- `setgid` *group*
  Allow or deny to set a group ID corresponding to *group* to a process. This rule is also applied when users execute a program with a setgid bit on.
- `setgroups`
  Allow or deny to set a list of supplementary group IDs.
- `setpgid`
  Allow or deny to change the process group ID.
- `setrlimit` *res*
  Allow or deny to change the limit of a resource specified by *res*.
- `setsid`
  Allow or deny to run the process in a new session.
- `setuid` *user*
  Allow or deny to set a user ID corresponding to *user* to a process. This rule is also applied when users execute a program with a setuid bit on.
- `sigaction` *sig*
  Allow or deny to change a signal handler corresponding to a signal *sig*.

A.1.4 Memory Management Rules

- `mmap` *path* `with` *prot*
  Allow or deny to map a file specified by *path* to memory with a protection mode specified by *prot*.
- `mprotect` *path* `with` *prot*
  Allow or deny to change the protection of a memory-mapped file specified by *path* to a new protection mode specified by *prot*.
- `munmap` *path*
  Allow or deny to unmap a memory-mapped file specified by *path*.

A.1.5 System Administration Rules

- `delete_module` *name*
  Allow or deny to unload a module specified by *name*.
- `init_module` *name*
  Allow or deny to load a module specified by *name*.
- `mount` *path* `to` *dir*
  Allow or deny to mount a device specified by *path* to a directory specified by *dir*.
- `quotactl` *cmd*
  Allow or deny to control disk quotas with a command *cmd*.

- `reboot`
  Allow or deny to reboot the system.
- `setdomainname`
  Allow or deny to change the domain name.
- `sethostname`
  Allow or deny to change the host name.
- `settime`
  Allow or deny to set the system time.
- `sysctl` *name*
  Allow or deny to write a system parameter specified by *name*.
- `umount` *path*
  Allow or deny to unmount a file system specified by *path*.

A.2 Conditions

Compacto uses attributes of a process as conditions for applying policy rules. All conditions are listed in Table A.1.

Table A.1: All conditions for policy rules.

Directive	Condition
`by-prog` *prog*	Apply if the program name of the process is *prog*
`via-prog` *prog*	Apply if the program name of one of the process and the ancestors is *prog*
`by-user` *user*	Apply if the owner's user name of the process is *user*
`via-user` *user*	Apply if the owner's user name of one of the process and the ancestors is *user*
`by-group` *group*	Apply if the owner's group name of the process is *group*
`via-group` *group*	Apply if the owner's group name of one of the process and the ancestors is *group*
`at-tlevel` *min–max*	Apply if the taint level of the process is between *min* and *max*
`of-tlevel` *min–max*	Apply if the taint level of the related resource is between *min* and *max*
`by-auth-user` *user*	Apply if the user who Compacto has authenticated in the latest is *user*

Appendix B

Examples of Security Policy

The following policy rules are applied to the server just after the `save_state` system call is issued and removed when the `restore_state` system call is issued.

B.1 Apache

This section enumerates policy rules applied to the Apache web server named `httpd`. The server is installed in `/usr/local`. The network address of the server is 192.168.0.248 and the port is 80.

Policy rules applied just before `save_state`

The file name for a lock (`accept.lock.1467`) is determined from the process ID of Apache.

```
# for files
allow READ_ONLY "/usr/local/apache/htdocs/*"
allow mmap "/usr/local/apache/htdocs/*" with PROT_READ

allow fcntl "/usr/local/apache/logs/accept.lock.1467"
    with F_SETLKW

allow open "/dev/null/" with O_RDONLY
allow getdents "/usr/local/apache/htdocs/"

allow write "/usr/local/apache/logs/access_log"
allow write "/usr/local/apache/logs/error_log"

# for network
allow COMM 192.168.0.248:80 with *:*

# for CGI
```

```
allow chdir "/usr/local/apache/cgi-bin/"
allow exec "/usr/local/apache/cgi-bin/Count.cgi"

allow sigaction SIGUSR1
allow sigaction SIGCHLD

deny * by-prog "httpd"
```

Policy rules applied just after `accept`

The following rules are applied when the server is connected from a client host with network address 192.168.0.1 and port 3172. The `accept` system call is prohibited and receiving and sending packets are allowed only between this server and the client connected.

```
deny accept

allow recv 192.168.0.248:80 from 192.168.0.1:3172
deny recv

allow send 192.168.0.248:80 to 192.168.0.1:3172
deny send
```

Policy rules applied just after reading a request

The following rules are applied when the server receives a request from the client when the contents of the request is "`GET /~kourai/menu-ja.html`." For simplicity, the user's HTML files are located in `/usr/local/apache/htdocs/home/`. Opening, memory-mapping, and closing only the requested HTML file are allowed. Receiving any other packets is not allowed.

```
deny recv

allow READ_ONLY "/usr/local/apache/htdocs/home/kourai/menu-ja.html"
deny READ_ONLY "/usr/local/apache/htdocs/*"

allow mmap "/usr/local/apache/htdocs/home/kourai/menu-ja.html"
    with PROT_READ
deny mmap "/usr/local/apache/htdocs/*"
```

B.2 wwwcount

This section enumerates policy rules applied to the wwwcount CGI program named `Count.fcgi`. This program is modified so that it uses the process pool technique using the fastcgi module of Apache. Wwwcount communicates with Apache using a UNIX domain socket.

Policy rules applied just before `save_state`

The path used for a UNIX domain socket is dynamically set.

```
# for files
allow READ_ONLY "/usr/local/apache/etc/Counter/conf/count.cfg"

allow APPEND "/usr/local/apache/etc/Counter/logs/Count2_5.log"

allow READ_WRITE "/usr/local/apache/etc/Counter/data/*"
allow flock "/usr/local/apache/etc/Counter/data/*"

allow READ_ONLY "/usr/local/apache/etc/Counter/digit/*"

# for UNIX domain socket
allow COMM_UNIX "/tmp/fcgi/a3ebac7fe059c1e568812eace40c0e62"

deny *
```

Policy rules applied just after `accept`

The following rules are applied when wwwcount is connected from the web server. The data file for storing the count number and the image used for the access counter are specified in a request from the web server. In this example, the data file is `kourai.dat` and the image is type K.

```
allow READ_WRITE "/usr/local/apache/etc/Counter/data/kourai.dat"
deny READ_WRITE "/usr/local/apache/etc/Counter/data/*"

allow flock "/usr/local/apache/etc/Counter/data/kourai.dat"
deny flock "/usr/local/apache/etc/Counter/data/*"

allow READ_ONLY "/usr/local/apache/etc/Counter/digits/K/strip.gif"
deny READ_ONLY "/usr/local/apache/etc/Counter/digits/*"
```

Appendix C

API for Extension Modules

The following API is provided by the protection manager. If the programmers of the extension modules conform to this API, they can change the protection level of the extension modules without modifying the binary code.

C.1 API for Callback Functions

The API for callback functions is invoked when events that the extension modules hook in the kernel occur. All the methods are virtual functions of C++ and the programmers can override them if necessary.

C.1.1 `FileSystem` Class

All file system modules must inherit this class.

- `void init()`
  This is invoked to initialize data for the file system.

- `int mount(Mount* mp, const String& path, FsOption* opt,`
  `          Nameidata* ndp, Proc* p)`
  This is invoked to mount the file system. `path` indicates a directory on which the file system is mounted. Specific data for the file system is passed by `opt`. In this method, the programmers should create the root vnode and private data of the file system.

- `int start(Mount* mp, int flags, Proc* p)`
  This is invoked just after `mount()`.

- `int unmount(Mount* mp, int mntflags, Proc* p)`
  This is invoked to unmount the file system. In this method, the programmers should call `Mount::vflush()` to remove all vnodes in the vnode list of the file system and then free the root vnode and private data of the filesystem, if necessary.

- `int statfs(Mount* mp, Statfs* sbp, Proc* p)`
  In this method, the programmers should set statistics information of the file system to `sbp`.

- `int root(Mount* mp, Vnode** vpp)`
  In this method, the programmers should set a pointer to the root vnode to `*vpp`.

- `int vget(Mount* mp, int num, Vnode** vpp)`
  In this method, the programmers should set to `*vpp` a pointer to a vnode whose has a node number of `num`. A reference count of the vnode must be increased.

- `int sync(Mount* mp, int waitfor, Ucred* cred, Proc* p)`
  In this method, the programmers should make all vnodes in the vnode list of the file system synchronize using `fsync()`.

-
  ```
  int fhtovp(Mount* mp, struct fid* fhp, MbufChain* nam,
             Vnode** vpp, int* exflags, Ucred** credanon)
  ```
  In this method, the programmers should translate the file handle `fhp` to the vnode and set a pointer to it to `*vpp`.

- `int vptofh(Vnode* vp, struct fid* fhp)`
  In this method, the programmers should translate the vnode `vp` to the file handle `fhp`.

-
  ```
  int quotactl(Mount* mp, int cmds, uid_t uid, caddr_t arg,
               Proc* p)
  ```
  This is invoked by `quotactl` system call. The high 24-bit of `cmds` indicates a quota main command and the low 8-bit indicates the type of quota.

- `int access(Vnode* vp, mode_t mode, Ucred* cred, Proc* p)`
  In this method, the programmers should check an access right of a file specified by `vp`.

- `int getattr(Vnode* vp, Vattr* vap, Ucred* cred, Proc* p)`
  In this method, the programmers should set an attribute of a file specified by `vp` to `vap`.

- `int setattr(Vnode* vp, Vattr* vap, Ucred* cred, Proc* p)`
  In this method, the programmers should change the attribute of `vp` according to `vap`.

- `int lock(Vnode* vp)`
  In this method, the programmers should set a flag for lock if necessary.

- `int unlock(Vnode* vp)`
  In this method, the programmers should clear a flag for lock if necessary.

- `int islocked(Vnode* vp)`
  In this method, the programmers should check whether a flag for lock is set if necessary.

- `int advlock(Vnode* vp, caddr_t id, int op, struct flock* fl,`
  `int flags)`
  This is invoked to do advisory locks. Advisory locks mean that locks are enforced for only processes that request locks.

- `int lookup(Vnode* dvp, Vnode** vpp, CompName* cnp)`
  In this method, programmers should look up a pathname specified by `cnp` and set a pointer to a vnode looked up to `*vpp`. `dvp` is the vnode of a directory on which the file system is mounted.

- `int open(Vnode* vp, int mode, Ucred* cred, Proc* p)`
  This is invoked to open a file specified by the vnode `vp`. In this method, the programmers should do something specific to the file system if necessary.

- `int close(Vnode* vp, int fflag, Ucred* cred, Proc* p)`
  This is invoked to close a file specified by `vp`. In this method, programmers should do something specific to the file system if necessary.

- `int create(Vnode* dvp, Vnode** vpp, CompName* cnp, Vattr* vap)`
  This is invoked to create a file. `dvp` is the vnode of a directory to create a new file. `cnp` holds the name of a newly created file. The attribute of the file is passed by `vap`. In this method, programmers should set a pointer to a newly created vnode to `*vpp`.

- `int remove(Vnode* dvp, Vnode* vp, CompName* cnp)`
  This is invoked to remove a file by `unlink` system call. `vp` is the vnode for a removed file and `dvp` is the vnode for a parent directory of the file. `cnp` indicates the name of a file to be removed.

- `int read(Vnode* vp, Uio* uio, int ioflag, Ucred* cred)`
  This is invoked by the `read` system call to read data from a file. In this method, programmers should copy the buffer of the vnode `vp` to `uio`.

- `int write(Vnode* vp, Uio* uio, int ioflag, Ucred* cred)`
  This is invoked by the `write` system call to write data to a file. `uio` holds data to be written. If the append flag `IO_APPEND` is set in `ioflag`, the programmers should append data to the end of a file specified by `vp`. In this method, programmers should copy data of `uio` to the buffer of `vp`.

- `int bwrite(Buf* bp)`
  This is invoked to write the contents of `bp` to a file.

- `int strategy(Buf* bp)`
  This is invoked to read from or write to a file. If the read flag `B_READ` is set, the programmers should read from a file to `bp`; otherwise, they should write the contents of `bp` to a file.

- `int bmap(Vnode* vp, daddr_t bn, Vnode** vpp, daddr_t* bnp,`
  `         int* runp)`
  In this function, the logical block number `bn` should be translated to the physical block number and a pointer to it is set to `*bnp`.

- `int truncate(Vnode* vp, off_t length, int flags, Ucred* cred,`
  `             Proc* p)`
  This is invoked to change the length of a file. `length` indicates a new length of a file. `vp` is the vnode for a file whose size is changed.

- `int fsync(Vnode* vp, Ucred* cred, int waitfor, Proc* p)`
  In this method, the programmers should make dirty buffers on `vp` synchronize.

- `int update(Vnode* vp, struct timespec* access,`
  `           struct timespec* modify, int waitfor)`
  In this method, the programmers should change an access time and a modified time of a file specified by `vp`.

- `int inactive(Vnode* vp)`
  This is invoked to inactivate `vp` so that it is not used. In this method, programmers should write dirty buffer back by `Vnode::vgone()` if necessary.

- `int reclaim(Vnode* vp)`
  This is invoked to reuse `vp`. In this method, the programmers should call `Vnode::cachePurge()` and `freePrivateData()` for `vp`.

- `int abortop(Vnode* dvp, CompName* cnp)`
  This is invoked to abort an operation of the file system due to errors. In this method, the programmers should free memory for the pathname of `cnp` if necessary.

- `int mkdir(Vnode* dvp, Vnode** vpp, CompName* cnp, Vattr* vap)`
  This is invoked to make a new directory. `dvp` is the vnode for a directory where a new directory is made. `cnp` indicates the new directory name to be created and `vap` holds the attribute of a new directory. In this method, programmers should set a pointer to a vnode for a new directory to `*vpp`.

- `int mknod(Vnode* dvp, Vnode** vpp, CompName* cnp, Vattr* vap)`
  This is invoked by the `mknod` and `mkfifo` system calls in order to create a special file as `mkdir()`.

- `int rmdir(Vnode* dvp, Vnode* vp, CompName* cnp)`
  This is invoked to remove a directory. `dvp` is the vnode for a directory where a directory is made. `vp` is the vnode to be removed and `cnp` indicates the directory name to be removed.

- `int readdir(Vnode* vp, Uio* uio, Ucred* cred, int* eofflag,`
  `            off_t* cookies, int ncookies)`
  This is invoked by the `getdent` system call to read a block of directory entries. In this method, the programmers should copy directory entries from `vp` to `uio`.

- `int link(Vnode* dvp, Vnode* vp, CompName* cnp)`
  This is invoked to make a hard link to a file. `dvp` is the vnode for a directory where a hard link is created. `vp` is the vnode for an existing file to which a hard link is made. `cnp` indicates the new file name for a hard link. In this method, the programmers should create a new file for a hard link.

- `int symlink(Vnode* dvp, Vnode** vpp, CompName* cnp, Vattr* vap,`
  `            char* target)`
  This is invoked to make a symbolic link as `link()`.

- `int readlink(Vnode* vp, Uio* uio, Ucred* cred)`
  This is invoked to read the contents of a symbolic link. `vp` is the vnode for a symbolic link to be read. In this method, the programmers should copy the contents to `uio`.

- `int ioctl(Vnode* vp, u_long command, caddr_t data, int fflag,`
  `          Ucred* cred, Proc* p)`
  This is invoked by the `ioctl` and `fcntl` system calls.

- `int pathconf(Vnode* vp, int name, register_t* retval)`
  This is invoked by the `fpathconf` system call. In this method, the programmers should set the return value to `retval`.

C.1.2 `NetworkSystem` Class

All network subsystem modules must inherit this class. In the current implementation, programmers can write network protocols over IP such as UDP and TCP.

- `void init()`
  This is invoked to initialize the network subsystem. It is used to initialize global data for the network subsystem.

- `int attach(Socket* so, long proto, Proc* p)`
  This is invoked by the `socket` and `accept` system calls. In this method, the programmers should create a new protocol control block.

- `int detach(Socket* so, Proc* p)`
  This is invoked by the `close` system call. In this method, the programmers should destroy the protocol control block.

- `int bind(Socket* so, SockAddr* nam, Proc* p)`
  This is invoked by the `bind` system call. In this method, the programmers should call `InPcb::bind()`.

- `int listen(Socket* so, Proc* p)`
  This is invoked by the `listen` system call. In this method, the programmers should call `InPcb::bind()` if the local port has not been allocated yet. Next, they should change the protocol state to a listen state.

- `int connect(Socket* so, SockAddr* nam, Proc* p)`
  This is invoked by the `connect` system call. In this method, the programmers should call `InPcb::bind()` if the local port has not been allocated yet. Next, they should call `InPcb::connect`.

- `int disconnect(Socket* so, Proc* p)`
  This is invoked by the `close` system call. In this method, the programmers should close the socket `so` soon if the connection is not established. Otherwise, they should make the socket a disconnecting state using `Socket::setDisconnecting()`, flush the receive buffer of the socket, and then close the socket.

- `int accept(Socket* so, SockAddr* nam, Proc* p)`
  This is invoked for a newly created socket `so` by the `accept` system call. In this method, the programmers should set the address of the socket with which `so` has been connected to `nam` using `InPcb::setPeerAddr()`.

- `int shutdown(Socket* so, Proc* p)`
  This is invoked by the `shutdown` system call. In this method, the programmers should make the socket `so` a state where the socket cannot send more packet, and close it.

- `int recvd(Socket* so, long flags, Proc* p)`
  This is invoked when a packet is received. In this method, the programmers can send an acknowledgment to the received packet.

-
  ```
  int send(Socket* so, MbufChain* m, SockAddr* nam,
           MbufChain* ctrl, Proc* p)
  ```
  This is invoked by the `send` system calls. `m` is the data to be sent and `nam` indicates the address to which the data is sent. In this method, the programmers should call `InPcb::connect()`. Next, they should attach a protocol header to `m` and call `Ip::output()` to pass the packet to a lower layer.

- `int recvOOB(Socket* so, MbufChain* m, long flags, Proc* p)`
  This is invoked to read out-of-band data present on `so`. In this method, the programmers should copy the data to `m`.

-
  ```
  int sendOOB(Socket* so, MbufChain* m, SockAddr* nam,
              MbufChain* ctrl, Proc* p)
  ```
  This is invoked to send the out-of-band data `m` by the `send` system call. In

this method, the programmers should attach a protocol header to `m` and call `Ip::output()`.

- `int sockAddr(Socket* so, SockAddr* nam, Proc* p)`
  This is invoked to the local address of the socket `so`. In this method, the programmers should set the address to `nam` using `InPcb::setSockAddr`.

- `int peerAddr(Socket* so, SockAddr* nam, Proc* p)`
  This is invoked to the address with which the socket `so` is connected. In this method, the programmers should set the address to `nam` using `InPcb::setPeerAddr`.

- `int abort(Socket* so, Proc* p)`
  This is invoked to abort sending packets. In this method, the programmers should drop pending packets.

- `int control(Socket* so, u_long cmd, caddr_t data, Proc* p)`
  This is invoked to process protocol specific ioctl by the `ioctl` and `fcntl` system calls. In this method, the programmers can call `Ip::control()`.

- `int sense(Socket* so, struct stat* ub, Proc* p)`
  This is invoked by the `fstat` system call. In this method, the programmers should set the socket status to `ub`.

- `int connect2(Socket* so, Socket* so2, Proc* p)`
  This is invoked by the `socketpair` system call. In this function, the programmers should connect `so` and `so2` if necessary.

- `int input(MbufChain* m, int hlen)`
  This is invoked when the IP layer receives a packet. `m` is the packet with an IP header, IP options, and a header and options of this protocol. In this method, the programmers should trim all headers and options and chain it to the receive buffer of the socket.

- `void* ctlinput(int cmd, SockAddr* sa, void* data)`
  This is invoked to process control information from lower layers.

-
  ```
  int ctloutput(int op, Socket* so, int level, int optname,
                MbufChain*& mp)
  ```
  This is invoked to process control information by the `setsockopt` and `getsockopt` system calls. In this method, the programmers should process control information.

- `void drain()`
  This is invoked when memory is in short supply. In this method, the programmers should free memory as much as possible.

-
  ```
  void sysctl(int* name, u_int namelen, void* oldp,
              size_t* oldlenp, void* newp, size_t newlen)
  ```
  This is invoked by the `__sysctl` system call.

- `void sysFastTimeout()`
  This is invoked every 200 miliseconds.
- `void sysSlowTimeout()`
  This is invoked every 500 miliseconds.
- `void soUpcall(Socket* so)`
  This is invoked when `so` is woken up for read by `input()`. In this method, the programmers can read packets without blocking by `Socket::receive()`.

C.2 API for Manipulating the Kernel Data

We describe an API for manipulating the kernel data in the current implementation. Basically, we explain all methods, but the description of some methods are omitted. Also, the below is an API necessary for developing sample modules and is not complete.

C.2.1 Common Classes

The following classes are commonly used in every subsystem module.

`System` Class

This is the class for the whole system.

- `struct timeval getCurrentTime()`
  This returns the current time in the timeval format.
- `struct timespec getCurrentTimeSpec()`
  This returns the current time in the timespec format.
- `int sleep(caddr_t id, int priority, const char* msg)`
  This makes the current thread of the extension module sleep with `priority` until it is woken up on `id` by `wakeup()`.
- `void wakeup(caddr_t id)`
  This wakes up the thread of the extension module made slept by `sleep()` on `id`.
- `void timeout(void (*ftn)(void *), void* args, int msec)`
  This creates a new thread that executes the function `ftn` with `args` and makes it sleep for `msec` miliseconds.

`Interrupt` Class

This is the class for handling the level of interrupts.

- `void disableSoftNet()`
  This disables interrupts from protocol stacks.

- `void enableAll()`
  This enables interrupts disabled by `disableSoftNet()`.

`Proc` Class

This is the class for handling a process. This class is currently used only for an identifier and has no methods.

Miscellaneous Classes

- `String`
  This is the class for handling strings.
- `PList`
  This is the class for handling a pointer list. This class has an internal class `Iterator`. It is used to traverse the list.
- `List`
  This is the class for a special list. Programmers cannot directly use or inherit this class. This class has an internal class `Iterator`. The methods are the same with `PList` class.
- `TailQueue`
  This is the class for a special tail queue. Programmers cannot directly use or inherit this class. This class has the same methods with `List` class, but the internal algorithm is different.
- `CircleQueue`
  This is the class for a special circular queue. Programmers cannot directly use or inherit this class. This class has the same methods with `List` class, but the internal algorithm is different.

C.2.2 Classes for File System

The following classes are used in file system modules.

`Mount` Class

This is the class for handling the mount structure per file system.

- `void setPrivateData(void* data)`
  This sets the file system specific data to `data`.
- `VnodeList* getVnodeList()`
  This returns the list of vnodes that the file system has.
- `void getNewFsid(const String& name)`
  This generates a new identifier of the file system from `name` and sets it to the structure for statistics information of the file system.

- `Vnode* getNewVnode()`
  This creates a new vnode and return it. The vnode is inserted in the vnode list. To create a new vnode, programmers cannot use the constructor of `Vnode` directly.

- `int vflush(Vnode* skipvp, int flags)`
  This removes all vnodes in the vnode list except `skipvp`. For `skipvp`, the root vnode of the file system is often specified.

- `Mount* getVfs(fsid_t* fsid)`
  This is a static method. This returns the mount structure of the file system identified by `fsid`.

- Miscellaneous
  `getPrivateData`, `freePrivateData`, `setFlags`, `getStatfs`, `clearFlags`, `checkFlags`

`Statfs` Class

This is the class for handling the statistics information of the file system.

- `void setName(const String& name)`
  This sets the name of the file system to `name`.

- `void setPath(const String& path)`
  This sets the directory on which the file system is mounted to `path`.

- `void setFsName(const String& fsname)`
  This sets the file system specific name to `fsname`. For example, `fsname` is a hostname from which the file system is mounted for NFS.

- `long getFsid()`
  This returns a part of the identifier for the file system.

- `long getFullFsid()`
  This returns the full identifier for the file system.

- Miscellaneous
  `getName`, `getPath`, `getFsName`, `setFlags` `getFlags`, `changeFlags`, `setBlockSize`, `getBlockSize`, `setIOSize`, `getIOSize`, `setBlockNum`, `getBlockNum`, `setFreeBlockNum`, `getFreeBlockNum`, `setAvailBlockNum`, `getAvailBlockNum`, `setNodeNum`, `getNodeNum`, `setFreeNodeNum`, `getFreeNodeNum`

`Vnode` Class

This is the class for handling the file node.

- `void setPrivateData(void* data)`
  This sets the file system specific data for this vnode to `data`.

- `void setType(enum vtype type)`
  This sets the type of this vnode. For example, `VREG` is for a regular file, `VDIR` is for a directory, and so on.

- `void setFlags(u_long flag)`
  This sets the bits of `flag` in the flag. The type of the flag is `VROOT` for the root vnode, and so on.

- `void lock()`
  This sets the lock flag.

- `void wait(int priority = -1, const char* msg = "")`
  This makes the current thread sleep until `wakeup()` is called for the vnode. `priority` and `msg` are passed to `System::sleep()`.

- `void wakeup()`
  This wakes up the thread made sleep by `wait()`.

- `void vref()`
  This checks that the reference count of users is positive and then increments it.

- `void vput()`
  This calls `FileSystem::unlock()` and then calls `vrele()`.

- `void vrele()`
  This decrements the reference count of users. If the count gets 0, it calls `FileSystem::inactive()`.

- `void vhold()`
  This increments the reference count of pages and buffers.

- `void holdRele()`
  This checks that the reference count of pages and buffers is positive and then decrements it.

- `int vget(int lockflag)`
  This increments the reference count of users and calls `FileSystem::lock()` if `lockflag` is 1.

- `int vinvalBuf(int flags, Ucred* cred, Proc* p)`
  This flushes out and invalidate all buffers associated with the vnode.

- `void vclean(int flags)`
  This disassociates the file system from the vnode. This calls `FileSystem::inactivate()` if necessary, and then calls `FileSystem::reclaim()`.

- `void vgone()`
  This calls `vclean`.

- `void cachePurge()`
  This flushes the cache for the vnode.

- `void vwakeup(Buf* bp)`
  This updates outstanding I/O count and does wakeup if requested.

-
  ```
  int vaccess(mode_t file_mode, uid_t uid, gid_t gid,
              mode_t acc_mode, Ucred* cred)
  ```
  This checks for the access right of the vnode.

- `BufList* getCleanList()`
  This returns the list of clean buffers associated with the vnode.

- `BufList* getDirtyList()`
  This returns the list of dirty buffers associated with the vnode.

- `int lookup(Nameidata* ndp)`
  This looks up the pathname. Programmers should pass `ndp` whose vnode for a starting directory and component name are set.

- Miscellaneous
  `getPrivateData`, `freePrivateData`, `getType`, `setTag`, `getTag`, `getUseCount`, `getNumOutput`, `getWriteCount`, `getHoldCount`, `clearFlags`, `checkFlags`, `getMount`, `getDevice`, `unlock`, `isLocked`, `isWaiting`, `incrUseCount`, `decrUseCount`, `incrNumOutput`, `decrNumOutput`, `decrHoldCount`, `destroy`

`Vattr` Class

This is the class for an attribute of a vnode.

- `void setUid(uid_t uid)`
  This sets the owner user ID to `uid`.

- `void setBytes(u_quad_t bytes)`
  This sets the size of disk space held by a file to `bytes`.

- `void setMode(u_short mode)`
  This sets the file access mode to `mode`. The mode is `VREAD`, `VWRITE`, and/or `VEXEC`.

- Miscellaneous
  `getUid`, `setGid`, `getGid`, `getBytes`, `setBlockSize`, `getBlockSize`, `setGeneration`, `setFlags`, `changeFlags`, `setType`, `setFsid`, `setDevice`, `getDevice`, `getMode`, `setNLink`, `getNLink`, `setFileid`, `getFileid`, `setSize`, `getSize`, `setAccessTime`, `getAccessTime`, `setChangeTime`, `getChangeTime`, `setModifiedTime`, `getModifiedTime`, `create`, `destroy`

`Nameidata` Class

This is the class for pathname lookup.

- `void setPathLength(int len)`
  This sets the length of a pathname to `len`.
- `void setStartDir(Vnode* vp)`
  This sets the vnode for a starting directory.
- `Vnode* getVnode()`
  This returns the vnode of the result of pathname lookup.
- `Vnode* getParentVnode()`
  This returns the vnode of a directory where the result vnode is.
- `getCompName`, `getStartDir`, `getParentVnode`, `create`, `destroy`

`CompName` Class

This is the class for a component name used in pathname lookup.

- `const String& getName()`
  This returns the pathname to be looked up.
- `void setNameiOp(u_long op)`
  This sets the operation for which the pathname lookup is done. The operation is `LOOKUP`, `CREATE`, `DELETE`, or `RENAME`.
- `void setPathName(const String& path)`
  This sets the pathname to `path`.
- Miscellaneous
  `getNameLen`, `setFlags`, `checkFlags`, `changeFlags`, `setUcred`, `getUcred`, `setProc`, `getProc`, `freePathname`

`Buf` Class

This is the class for handling the file buffer.

- `void lock()`
  This sets the lock flag.
- `void setBusy()`
  This sets the busy flag.
- `void wait(int priority = -1, const char* msg = "")`
  This makes the current thread sleep until `wakeup()` is called for this buffer.
- `void wakeup()`
  This wakes up the thread made sleep by `wait()` for this buffer.

- `void setFlags(long flag)`
  This sets the bits of `flag` in the flag. The type of the flag is `B_ASYNC` for asynchronous I/O, `B_DELWRI` for delayed I/O, `B_READ` for reading a buffer, and so on.
- `void setResident(long resid)`
  This sets the size of data remaining in this buffer.
- `void setDirtyOffset(int off)`
  This sets the offset of dirty region in this buffer.
- `void setDirtyEnd(int end)`
  This sets the end of dirty region in this buffer.
- `void bremFree()`
  This removes this buffer from the free list.
- `void brelse()`
  This release this buffer on to the free list.
- `void brelVp()`
  This disassociates this buffer from a vnode.
- `void bgetVp(Vnode* vp)`
  This associates this buffer with the vnode `vp`.
- `int bwrite()`
  This writes out the contents of this buffer.
- `int bioWait()`
  This waits for operations on this buffer to complete. This method returns an error if it fails the operations.
- `void bioDone()`
  This marks I/O complete on this buffer. If the operations are not asynchronous, it wakes up a thread waiting for this buffer.
- `int bread(Ucred* cred, int async = 0)`
  This reads a disk block to this buffer.
- `void clrBuf()`
  This clears the data area of this buffer by zero.
- `void copyIn(caddr_t addr, int size = -1, off_t offset = 0)`
  This copies memory of `addr` to this buffer with the offset `offset` by `size`.
- `void copyOut(caddr_t addr, int size = -1, off_t offset = 0)`
  This copies the data of this buffer with the offset `offset` to memory of `addr` by `size`.

- `Buf* getBlock(Vnode* vp, daddr_t lblkno, int size)`
  This is a static method. This returns a new buffer with the size of `size`. If a buffer associated with the vnode `vp` and the logical block number `lblkno` exists, this returns it.

- Miscellaneous
  `unlock`, `isLocked`, `clearBusy`, `isBusy`, `isWaiting`, `setPhysicalBlockNum`, `getPhysicalBlockNum`, `setLogicalBlockNum`, `getLogicalBlockNum`, `setVnode`, `getVnode`, `setReadUcred`, `getReadUcred`, `setWriteUcred`, `getWriteUcred`, `getSize`, `getResident`, `setError`, `setProc`, `setDevice`, `getDirtyOffset`, `getDirtyEnd`, `setValidOffset`, `getValidOffset`, `setValidEnd`, `getValidEnd`, `getData`

`Uio` Class

This is the class for handling a universal I/O buffer.

-
  ```
  int read(Vnode* vp, daddr_t lblkno, int xfersize, off_t offset,
           int blksize, Ucred* cred)
  ```
  This calls `FileSystem::strategy` and then copies the contents of the file buffer read to this buffer. `lblkno` indicates the logical block number to be read. The file is read from `offset` by `xfersize`.

- `int bulkRead(Vnode* vp, int blksize, int filesize, Ucred* cred)`
  This reads the contents of a file to this buffer by `filesize`, repeating to call `read()`.

-
  ```
  int write(Vnode* vp, daddr_t lblkno, int xfersize, off_t offset,
            int blksize, int filesize, Ucred* cred)
  ```
  This fills the contents of this buffer to a file buffer and then calls `FileSystem::strategy` to write out the buffer to a file.

- `int bulkWrite(Vnode* vp, int blksize, int filesize, Ucred* cred)`
  This writes out the contents of this buffer to a file by `filesize`, repeating to call `write()`.

- `void setOffset(off_t off)`
  This sets the offset at which the operation should start.

- `void setResident(int resid)`
  This sets the size of data remaining in this buffer.

- `void setIovec(struct iovec* iov, int iovcnt = 1)`
  This sets the I/O vector array to this buffer. The size of array is `iovcnt`.

- `int copyIn(caddr_t cp, int size)`
  This fills this buffer with the contents of `cp` by `size`.

- `int copyOut(caddr_t cp, int size)`
  This copies the contents of this buffer to `cp` by `size`.

- Miscellaneous
  `getOffset`, `getResident`, `setProc`, `getProc`, `getIovec`, `getIovCount`

`DirEntry` Class

This is the class for a directory entry.

- `DirEntry(u_int32_t fileno, u_int16_t reclen, u_int8_t type,`
  `          const String& name)`
  This is a constructor of `DirEntry` class.
- `int pack(Uio* uio)`
  This copies the directory entry to `uio`.
- Miscellaneous
  `length`, `getType`, `getFileNo`, `getNameLen`, `getName`

`Ucred` Class

This is the class for credentials.

- `void setGroupMembers(gid_t* groups, int ngroups)`
  This sets the members of the group to the array `groups` with the size of `ngroups`.
- `Bool isGroupMember(gid_t gid)`
  This returns True if `gid` is a member of the group.
- `void crHold()`
  This increments the reference count.
- Miscellaneous
  `setUid`, `getUid`, `setGid`, `getGid`, `getGroupMembers`, `getGroupNum`

Miscellaneous Classes

- `VnodeList`
  This is the class for a list of vnodes. The methods are the same with `List` class.
- `BufList`
  This is the class for a list of buffers. The methods are the same with `List` class.
- `BufQueue`
  This is the class for a queue of buffers. The methods are the same with `TailQueue` class.

C.2.3 Classes for Network Subsystem

The following classes are used in network subsystem modules.

`MbufChain` Class

This is the class for handling a mbuf chain.

- `int length()`
  This returns the length of this mbuf chain.
- `int remain()`
  This returns the remaining size for dissect.
- `int build(caddr_t cp, int size)`
  This fills this mbuf chain with the data `cp` with the size of `size`.
- `int build(SockBuf* sb, int off, int size)`
  This fills this mbuf chain with the data of the socket buffer `sb`. The data is copied from the offset `off` by `size`.
- `int buildAsBytes(caddr_t* cp, int size)`
  This reserves the space for data of `size` in this mbuf chain and returns a pointer to the data to `*cp`.
- `int buildAsIovec(struct iovec** iov, int size)`
  This reserves the space for I/O vector array of `size` in this mbuf chain and returns a pointer to the array to `*iov`.
- `int append(MbufChain* mc)`
  This appends the mbuf chain `mc` to the end of this mbuf chain.
- `int prepend(MbufChain* mc)`
  This appends this mbuf chain to the end of the mbuf chain `mc`.
- `int dissect(caddr_t cp, int size)`
  This copies the data in this mbuf chain from the dissect point by `size`. The dissect point is advanced by `size`.
- `int advance(int size)`
  This advances the dissect point by `size`.
- `int trimHead(int size)`
  This trims the data from the head of this mbuf chain by `size`.
- `int checkSum(int len)`
  This calculates checksum for the front of this mbuf chain by `len`.
- `MbufChain* create(int type = MT_DATA)`
  This is a static method. This creates a new mbuf chain with the type of `type`. If creating a packet header, users must pass `MT_HEADER` as `type`.
- Miscellaneous
  `resetDissect`, `buildAsWord`, `dissectAsPointer`, `padding`, `trimTail`, `destroy`

`InPcb` Class

This is the class for a internet protocol control block.

- `void setLocalAddr(struct in_addr addr)`
  This sets the local address of the socket with this protocol control block to `addr`.
- `void setForeignAddr(struct in_addr addr)`
  This sets the address of a socket connected with the socket with this protocol control block to `addr`.
- `void setPrivatePcb(caddr_t ppcb)`
  This sets the protocol specific protocol control block to `ppcb`.
- `void setState(int state)`
  This sets the state of this protocol control block to `state`. The state is `INP_ATTACHED`, `INP_BOUND`, or `INP_CONNECTED`.
- `void setOptions(MbufChain* m)`
  This sets the IP options to `m`.
- `void setSockAddr(SockAddr* nam)`
  This fills `nam` with the address and port of the socket with this protocol control block.
- `void setPeerAddr(SockAddr* nam)`
  This fills `nam` with the address and port of a socket connected with the socket with this protocol control block.
- `RtEntry* getRtEntry()`
  This returns the routing table entry. If the entry does not created, it creates a new routing table entry.
- `int bind(SockAddr* nam, Proc* p)`
  This binds `nam` to the socket with this protocol control block.
- `int connect(SockAddr* nam)`
  This connects the socket with this protocol control block with a socket specified by `nam`.
- `void disconnect()`
  This disconnects the socket with this protocol control block from the connected socket.
- `InPcbQueue* getInPcbQueue()`
  This is a static method. This returns the queue of protocol contorol blocks of the network subsystem.

- `InPcb* lookup(struct in_addr faddr, u_int16_t fport,`
  `               struct in_addr laddr, u_int16_t lport)`
  This is a static method. This looks up a protocol control block using a foreign address `faddr`, a foreign port `fport`, a local address `laddr`, and a local port `lport`.

- Miscellaneous
  `getLocalAddr`, `getForeignAddr`, `setLocalPort`, `getLocalPort`, `setForeignPort`, `getForeignPort`, `getSocket`, `getIp`, `getRoute`, `getOptions`

`Ip` Class

This is the class for handling IP.

- `void setSrcAddr(struct in_addr src)`
  This sets the source IP address to `src`.

- `void setDstAddr(struct in_addr dst)`
  This sets the destination IP address to `dst`.

- `int output(MbufChain* m, MbufChain* opt, Route* ro, int flags,`
  `           MbufChain* mopt)`
  This is a static method. This attaches an IP header and the IP options `opt` to the packet `m` and passes it to a lower network layer, e.g. ethernet device driver.

- `int ctloutput(int op, Socket* so, int level, int optname,`
  `              MbufChain*& mp)`
  This is a static method. This processes a socket option for IP

- `void stripOptions(MbufChain* m)`
  This is a static method. This strips out IP options from the packet `m`.

- `int control(Socket* so, u_long cmd, caddr_t data, IfNet* ifp,`
  `            Proc* p)`
  This is a static method. This processes generic internet control operations. The arguments `cmd` and `data` are the same with those of `ioctl` system call.

- Miscellaneous
  `getSrcAddr`, `getDstAddr`, `setTtl`, `getTtl`, `getTos`, `getOptLen`

`Socket` Class

This is the class for handling a socket.

- `void setOptions(short options)`
  This sets the bits of `options` in the socket option. The type of the option is `SO_DONTROUTE`, `SO_REUSEADDR`, and so on.

- `void setState(short state)`
  This sets the bits of `state` in the socket state. The type of the state is `SS_NBIO` for non-blocking operations, `SS_ASYNC` for asynchronous I/O, and so on.
- `Bool hasQueueingSpace()`
  This returns True if there is a space to queue a new connection. The limit is determined by `listen` system call.
- `InPcb* allocateInPcb()`
  This allocates a new protocol control block.
- `SockBuf* getSndBuf()`
  This returns the send buffer.
- `Socket* getNewConn(int connstatus)`
  This creates a new socket with the state of `connstatus`.
- `int reserve(u_long sndcc, u_long rcvcc)`
  This reserves buffer spaces for send and receive to `sndcc` and `rcvcc`, respectively.
- `int abort()`
  This aborts sending packets.
- `void cantSendMore()`
  This makes this socket not send any more packets.
- `void wakeupRead()`
  This wakes up a thread waiting for socket read.
- `int connect(SockAddr* addr)`
  This connects this socket with a socket specified by `addr`.
- `int shutdown(int how)`
  This shuts down part of a full-duplex connection.
- `int close()`
  This disconnects if this socket is connected and then close this socket.
- `int send(SockAddr* addr, MbufChain* m, int flags)`
  This sends a packet `m` to the address `addr`.
- `int receive(SockAddr** paddr, MbufChain** m, int* flagsp)`
  This reads a pakcet from the receive buffer and the pointer is set in `m`.
- `void setUpcall(NetworkSystem* ns)`
  This sets up so that `NetworkSystem::soUpcall()` for `ns` is called by a socket upcall. The socket upcall is done when this socket is woken up for read.

- `Socket* create(int dom, int type, int proto)`
  This is a static method. This creates a new socket with the arguments of `socket` system call and returns it.
- `Socket* create(int fd)`
  This is a static method. This returns a socket corresponding to `fd`.
- Miscellaneous
  `getOptions`, `checkOptions`, `clearState`, `checkState`, `getProtoSw`, `getRcvBuf`, `setConnecting`, `setConnected`, `setDisconnecting`, `setDiconnected`, `setError`, `cantRecvMore`, `wakeupWrite`, `clearUpcall`, `destroy`

`SockBuf` Class

This is the class for handling a socket buffer.

- `void setHighWatermark(u_long hiwat)`
  This sets the maximum buffer size to `hiwat`.
- `int getSpace()`
  This returns the size of space in this buffer.
- `MbufChain* getMbufChain()`
  This returns data in this buffer as a mbuf chain.
- `int reserve(u_long cc)`
  This checks for the value of `cc` and sets the maximum buffer size to it if acceptable.
- `void drop(int len)`
  This drops data from the front of this buffer by the size of `len`.
- `Bool needNotify()`
  This returns True if I/O is possible. It is used to to notify the other sockets.

-
  ```
  Bool appendAddr(SockAddr* asa, MbufChain* m0,
                  MbufChain* ctrl)
  ```
  This appends the address `asa`, the data `m0`, and control data `ctrl` to this buffer.
- `void append(MbufChain* m)`
  This appends the data `m` to this buffer.
- Miscellaneous
  `getHighWatermark`, `setLowWatermark`, `getLowWatermark`, `size`

`SockAddr` **Class**

This is the class for an socket address.

- `void setAddr(in_addr addr)`
  This sets the IP address to `addr`.
- `Bool isNullHost()`
  This returns True if this socket address points to null address.
- Miscellaneous
  `length`, `getAddr`, `setPort`, `getPort`, `isMulticast`, `isBroadcast`

`IfAddr` **Class**

This is the class for an interface address of internet.

- `struct in_addr getAddr()`
  This returns the internet address.
- Miscellaneous
  `getIfNet`, `getSockAddr`, `getBroadAddr`, `create`, `destroy`

`IfNet` **Class**

This is the class for a network interface.

- `u_long getMtu()`
  This returns the value of maximum transmission unit (MTU).
- Miscellaneous
  `checkFlags`

`Route` **Class**

This is the class for a route.

- `SockAddr* getDstAddr()`
  This returns the destination address.
- `void allocateRtEntry()`
  This allocates a new routing table entry.
- Miscellaneous
  `setRtEntry`, `getRtEntry`, `freeRtEntry`

`RtEntry` Class

This is the class for a routing table entry.

- `Bool checkFlags(short flags)`
  This returns True if the bits of `flags` are set in the flag. The type of the flag is `RTF_UP` for an usable route, `RTF_HOST` for a host entry, and so on.
- `SockAddr* getGwAddr()`
  This returns the gateway address.
- `SockAddr* getNetmask()`
  This returns the network mask.
- Miscellaneous
  `getFlags`, `getIfNet`, `getIfAddr`, `getRtMetrics`, `getDstAddr`

`RtMetrics` Class

This is the class for route metrics.

- `u_long getMtu()`
  This returns the maximum packet size called maximum transmission unit (MTU).
- `Bool checkLocks(u_long locks)`
  This returns True if the bits of `locks` are set in the lock flag. The type of the flag is `RTV_MTU`, `RTV_RPIPE`, `RTV_RTT`, and so on.
- Miscellaneous
  `setPecvPipe`, `setThreshold`, `getThreshold`, `setRtt`, `getRtt`, `setRttVar`, `getRttVar`, `getSendPipe`

Appendix D

Kernel Functions for the Protection Manager

Some of kernel functions for the parameter of the `kernfunc` system call are listed in Table D.1.

Table D.1: Kernel functions available to the protection manager.

command	description
`KF_UCINTR`	This changes an upcall enable flag according to the argument.
`KF_WAKEUPALL`	This wakes up a kernel thread waiting for the identifier specified by the argument.
`KF_GET_NEW_FSID`	This returns a unique file system identifier.
`KF_VREF`	This increments a reference count of a vnode of a file system in the kernel. This is used for NFS to access file systems in the kernel.
`KF_IP_OUTPUT`	This calls the output routine of an IP layer.
`KF_SOSEND`	This sends a packet. The mbuf chain passed as the argument is copied to the kernel memory so that it can be accessed in interrupt handlers of a network device driver.
`KF_RTALLOC`	This allocates a new routing table entry.
`KF_SOSETUPCALL`	This sets an upcall handler for a socket. The upcall is done when the socket receives packets.
`KF_KERNFS_*`	This executes a function provided by a file system in the kernel. `*` is `LOCK`, `UNLOCK`, `GETATTR`, `READ`, `WRITE`, and so on. This is used for NFS to access file systems in the kernel.