2014

A FIREWALL MODEL OF FILE SYSTEM SECURITY

Lihui Hu
Michigan Technological University

A FIREWALL MODEL OF FILE SYSTEM SECURITY

By

Lihui Hu

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY

2014

This dissertation has been approved in partial fulfillment of the requirements for the Degree of DOCTOR OF PHILOSOPHY in Computer Science.

Department of Computer Science

Thesis Advisor: Dr. *Jean Mayo*

Committee Member: Dr. *Steve Carr*

Committee Member: Dr. *Soner Onder*

Committee Member: Dr. *Haiying Liu*

Department Chair: Dr. *Charles Wallace*

Table of Contents

List of Figures

List of Tables

Acknowledgements

First, I would like to give my sincere thanks to my advisor, Dr. Jean Mayo. Her insights, expertise, guidance, patience, and kindness have helped me through every step of my Ph.D. study. Without her valuable ideas and suggestions in my research, I could not have completed this work. Her dedication on work and healthy attitude towards life set an example of excellence as a researcher, mentor, and role model.

Second, it is an honor to have Dr. Steve Carr, Dr. Soner Onder and Dr. Haiying Liu in my degree committee. Thanks to them for their suggestions and feedback through my proposal, oral exam and final defense. I would like to thank Dr. Steve Carr, who inspired me to enjoy the research and knowledge in computer science. Deeply thanks to Dr. Soner Onder for his comments and suggestions on this dissertation.

A special thanks to Dr. Xinli Wang for his help with my research. Thanks to Dr. Nilufer Onder for her encouragement. Many thanks to my fellow graduate students and friends, Yifei Li, Shuhan Ding, Wei Wang, Weiming Zhao, Warren Powers, Alicia Thorsen, who shared laughers and tears on this tough road together with me. Their companies made this journey more fun and bearable.

Finally, a well-deserved thanks to my mother, Yafeng Liu, and my husband, Zhonghai Wang, for their unconditional love, support and trust over the long years of my Ph.D. study. I am lucky to have them in my life.

Preface

This dissertation contains material of the paper "An empirical study of three access control systems" that has been published in the Proceedings of the 6th International Conference on Security of Information and Networks (SIN2013) in chapter 8. The reuse license is listed in Appendix A. Dr. Jean Mayo, Dr. Charles Wallace are the other co-authors. Dr. Jean Mayo contributed the idea of comparisons of access control models, and made suggestions on modifications of the tasks in the study. Dr. Charles Wallace provided the information on how to apply for conducting a user study in MTU, and on general rules of the design of user study. They both reviewed the design of the study and made suggestions and modifications of the paper. I designed and conducted the study, collected and analyzed the data, and wrote the draft of paper.

Abstract

File system security is fundamental to the security of UNIX and Linux systems since in these systems almost everything is in the form of a file. To protect the system files and other sensitive user files from unauthorized accesses, certain security schemes are chosen and used by different organizations in their computer systems. A file system security model provides a formal description of a protection system. Each security model is associated with specified security policies which focus on one or more of the security principles: confidentiality, integrity and availability. The security policy is not only about "who" can access an object, but also about "how" a subject can access an object. To enforce the security policies, each access request is checked against the specified policies to decide whether it is allowed or rejected.

The current protection schemes in UNIX/Linux systems focus on the access control. Besides the basic access control scheme of the system itself, which includes permission bits, setuid and seteuid mechanism and the root, there are other protection models, such as Capabilities, Domain Type Enforcement (DTE) and Role-Based Access Control (RBAC), supported and used in certain organizations. These models protect the confidentiality of the data directly. The integrity of the data is protected indirectly by only allowing trusted users to operate on the objects. The access control decisions of these models depend on either the identity of the user or the attributes of the process the user can execute, and the attributes of the objects. Adoption of these sophisticated models has been slow; this is likely due to the enormous complexity of specifying controls over a large file system and the need for system administrators to learn a new paradigm for file protection.

We propose a new security model: file system firewall. It is an adoption of the familiar network firewall protection model, used to control the data that flows between networked

computers, toward file system protection. This model can support decisions of access control based on any system generated attributes about the access requests, e.g., time of day. The access control decisions are not on one entity, such as the account in traditional discretionary access control or the domain name in DTE. In file system firewall, the access decisions are made upon situations on multiple entities. A situation is programmable with predicates on the attributes of subject, object and the system. File system firewall specifies the appropriate actions on these situations. We implemented the prototype of file system firewall on SUSE Linux. Preliminary results of performance tests on the prototype indicate that the runtime overhead is acceptable. We compared file system firewall with TE in SELinux to show that firewall model can accommodate many other access control models. Finally, we show the ease of use of firewall model. When firewall system is restricted to specified part of the system, all the other resources are not affected. This enables a relatively smooth adoption. This fact and that it is a familiar model to system administrators will facilitate adoption and correct use. The user study we conducted on traditional UNIX access control, SELinux and file system firewall confirmed that. The beginner users found it easier to use and faster to learn then traditional UNIX access control scheme and SELinux.

Chapter 1

Introduction

The traditional protection scheme used within UNIX (and Linux) systems has been proven inadequate to meet modern concerns for system security. The UNIX systems were not designed with a lot of consideration for security. It only had a basic access control scheme which has not been improved much from the original design. The main problems in original UNIX protection system are listed below:

First is the discretionary access control (DAC). DAC is the predominant access control mechanism in traditional UNIX systems. It means which subject can access a certain object and how the subject can access the object are controlled by the owner of that object. A process started by a user can access and modify any object the user has access to. This is dangerous in that attackers can exploit certain vulnerability of that process and let the process execute malicious code. Now the malicious code can operate on any object the process has access to. So we need mandatory access control (MAC) over the DAC to constrict what a subject can do.

Second is the coarse-grained access control. In UNIX systems, access decisions are made based on the identity of the user (UID) and the group(s) the user belongs to (GID). To a certain object, users are divided into three types: the owner, the users in the same group with the owner and all the other users. The possible permissions include only read, write and execute. This coarse-grained access control makes it cumbersome or impossible to apply certain security policies.

Last is the root account. The root account, or the super user, has unlimited privilege and can access and modify any object. The super power the root has been the main target of security attacks since with the root privilege, a user or a process can access, modify or delete any file. With DAC, setuid and setgid mechanism, the situation only gets worse. If attackers get a process run with root privilege to execute their malicious code, they can destroy the whole system. So there should be some mechanism to tone down the super power of the root account.

A lot of research has been done to provide more secure protections to the file system and avert these issues over the years. The most well-known security models that have been implemented are DTE, RBAC and POSIX Capabilities. They all focus on providing finer-grained access control and they all confine the super power of the root account in some way. Linux Security Module (LSM) is the most well-known security framework that supports these more complicated models.

It is already included in some distributions of Linux. But there are some issues with these security models which stops them from being used widely. One reason is that these models are complicated to use. To use these models, we need to design the right capability sets for each process, the right domain and type for each process and file , and the right roles for each user. Then we still need to set up the whole system. For example, to use Capabilities, one needs to decide for each process the right inheritable, permitted, and effective capabilities. The inheritance set adds more complexity to the work. To make DTE work, one needs to carefully define the domains and types, then tag each subject with the right domain and tag each subject with the right type. Even after we get all these done, the work to set up the system is not a trivial job. For RBAC, because of its nature, it is useful more in systems where the roles are easily recognized and they have different responsibilities.

So it is more popular in special organizations such as government, bank and health care systems.

We propose the file system firewall model which can be seen as an extra layer to the current security system in UNIX systems. It incorporates a filter and uses this filter to filter out the illegal access requests, just like the network firewall. When there is an access request, the firewall system will check the filter rules to see if the object of the access request has rules defined. If no matching rules are found, the control will be passed onto the original access control system. On the other hand, if there are some matching rules, the firewall will check the attributes of the access requests against the rules to get an action. The action can be "allow", "deny" or "redirection". Redirection means the access request is diverted from one subject or one object to another subject or object.

This firewall system is built into the kernel to make sure it is efficient and protected from being bypassed or compromised by a rogue application.

Compared with other security models, the firewall model has the following advantages. First, it can specify much more complex access control policies. The conditions on which to perform the specified actions are programmable on multiple attributes. Second, it is a general model that can accommodate different security policies. It can simulate other access control models using different attributes. Most importantly, firewall model is relatively easy to use compared to other access control systems. The adoption of the firewall model is not aggressively all or nothing. It can be done gradually by setting up access control rules on part of the system, which does not affect the rest of the system. It is easy to set up and maintain. Our user study shows that beginner users are more likely to use firewall model than UNIX access control scheme and SELinux and they can learn to use it relatively fast.

The contributions we have made in our work:

1. Design the file system firewall model.
2. Implement the prototype of file system firewall model on SUSE Linux.
3. Conduct the functional tests and performance tests on the prototype.
4. Theoretical comparisons of file system firewall with TE in SELinux.
5. Study the usability of access control systems and design a user study on access control systems.
6. Conduct a user study on three access control systems, analyze the results and provide suggestions to improve usability access control systems.

This dissertation is organized as follows. Chapter 2 introduces background information about file system protection and network firewalls. Chapter 3 describes the related work. In Chapter 4, we describe the motivations behind the firewall model for file system protection and the definitions of the firewall model. Chapter 5 is the prototype implementation of the firewall model and the performance tests. In Chapter 6, we explain in more details about the redirection feature in file system firewall. The comparison of firewall model is discussed in Chapter 7. Chapter 8 describes the user study on access control systems is. Chapter 9 is the conclusion.

Chapter 2

Background

The main content in this section includes introduction of the UNIX/Linux operating systems and file systems, important concepts in access control systems, security principles and the enhanced access control extensions and current security protections in UNIX/Linux systems. Since our file system firewall model borrows some ideas from the network firewall, we will include an introduction to the network firewall as well.

2.1 UNIX/Linux Operating Systems

UNIX was originally developed in 1969 by a group of AT&T employees at Bell Labs [58]. Today UNIX is still famous for its simplicity, power and safety. As the owner of the UNIX trademark, The Open Group has separated the UNIX trademark from any actual code stream itself [45], so there are multiple different implementations of UNIX systems such as Solaris, HP-UX and IBM's AIX. An open consensus specification, called Single UNIX Specification, defines the requirements for a UNIX system. Only systems that are fully compliant with and certified to the Single UNIX Specification can use the trademark; others are called "UNIX system-like" or "UNIX-like". The most popular UNIX-like systems are GNU/Linux and Berkeley Software Distribution (BSD). Linux is relatively new, having been originally written in 1991. Due to it being free and open source software, it is receiving more and more attention in both the academic and the industrial worlds. Some well-known distributions include Debian, Gentoo, Red Hat/Fedora and SUSE. For BSD, the most well- known BSD descendants are the major open source BSDs: FreeBSD, NetBSD and OpenBSD. Our research are done on Linux systems since it is free software, but it can apply to UNIX systems.

2.2 File System, Super Block, Inode

UNIX/Linux systems view a file as a stream of bytes that can be modified in length, as well as read and written at any specified position. Files can be of different types, such as ordinary files, directory files, special device files, link files, pipes or sockets. These files are all organized in a directory tree. The system in UNIX or Linux that manages files is called the file system [43, 44]. The file system deals with how to name and store files, how to protect files, how to share files and how to let the user create and access files.

UNIX/Linux file systems support many file system types. We list some well-known systems below:

• Ext2: Devised for Linux.

• Ext3: The ext2 file system enhanced with journaling capabilities. Ext3 allows fast file system recovery, and it supports POSIX AccessControl Lists (ACLs).

• Ext4: Ex4 file system was introduced in 2008. It modifies important data structures of the filesystem and provides improved design and better performance. It supports new features such as huge individual file size and overall file system size, bigger directory capacity, etc.

• Isofs (iso9660): Used by CDROM file system.

• Sysfs: A RAM-based (in memory) file system initially based on ramfs. Sysfs is used to export kernel objects to user space.

• Procfs: The proc file system acts as an interface to internal data structures in the kernel. It can be used to obtain information about the system and to change certain kernel parameters at run-time using sysctl command.

• VFAT: Short for virtual FAT (file allocation table). A Linux file system compatible with Microsoft Windows95 and Windows NT long file names on the FAT file system.

• NFS: Network file system.

• NTFS: The standard file system of Windows NT.

These file systems are different, but each of them contains four parts: a boot block, a super block, an inode list and data blocks as shown in the following chart.

Table 1 Filesystem Structure

Boot Super	Block Block	Inode List	Data Blocks

The boot block is located in the first several sectors including the initial bootstrap program to load the operating system [2]. The super block contains all the key information about the file system, such as a magic number to identify the file system type, the number of blocks in the file system, and other key administrative information [2]. The inode (index node) is the basic unit of files in the kernel and it stores the key information about a file, such as file address, link count, access mode, individual owner and group owner of the file. The inode list is created when a file system is created. Each file has a unique inode and is identified by an inode number in the file system where it resides. Two files can have the same inode numbers in two file systems, but not the same file system. The inode list is a static list of inodes. Once the file system is created, the size of the list cannot change. The initial size of the inode list is determined by the administrator and the size of the storage device. Data blocks contain the actual data in the form of directories and files.

2.3 Virtual File System (VFS)

UNIX/Linux systems provide a common interface, VFS, to the different file systems for application programs. VFS is a software layer in the kernel of the operating system. It manages kernel level file abstractions in one file format for all mounted file systems. So now the users can interact with the VFS no matter what type of file system the users are accessing. Even if they access several different file systems, they can still use the common system calls provided by the VFS. The VFS will then translate the system calls to appropriate operations within the appropriate file system. The VFS has a vnode structure for each active node (file or directory). The vnode contains a numerical designator for a

network-wide unique file [44]. The figure below shows how application programs interact with file systems through the VFS.

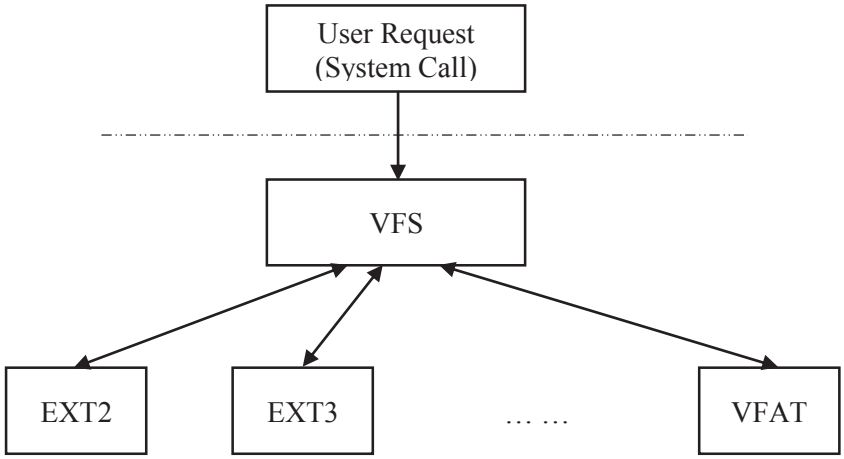

Figure 1: Virtual File System

2.4 Terminology in Access Control

Access control is the part of an operating system that deals with who can access a file and how he can access the file. Before we start to introduce different security models, schemes and extensions in UNIX/Linux systems, we will explain the terminology associated with access control.

2.4.1 Subject and Object

In any of the access control models and frameworks, there are three basic concepts: subject, object and access rights.

Subjects are the active objects, like processes and users.

Objects are protected entities, such as files and processes.

Access rights are the operations one subject can perform over objects or other subjects, such as enable, disable, read, write, execute, etc.

2.4.2 MAC and DAC

Access control can be grouped into two types: Mandatory Access Control (MAC) or Discretionary Access Control (DAC).

MAC is a type of access control in which the security levels and clearances are defined by the administrators, and the control rules between subjects and objects are enforced by the operating system or security kernel. Even if a subject owns a file, the subject cannot pass any permissions to other subjects, that is, the permissions are defined by the system and cannot be changed by a single normal user.

DAC is another type of access control defined as "a means of restricting access to objects based on the identity of subjects and/or groups to which they belong" [18]. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) onto any other subject (unless restrained by mandatory access control) [18]. So the access control to the objects is at the discretion of the user. An object's owner, who is usually also the objects creator, has discretionary authority over who else may access that object. For instance, in UNIX, the owner of a certain file can change the permissions of the file he owns and determine if another can access the file.

Some access control models are MAC, some are DAC and some are the hybrid of the two. MAC is used more often in the government and the military.

2.4.3 Principles of Computer Security

Confidentiality, integrity and availability are the key principles of computer security. Every access control model will have its own focus among these three principles. Confidentiality means that only authorized users can access certain information. An unauthorized access request should be denied. Integrity means that information should only be modified

properly by appropriate people. Availability means that the information is accessible by the authorized users.

2.5 Access Control Models, Schemes and Extensions

2.5.1 Multilevel Security

Multi-level security (MLS) is a security scheme in which both objects and subjects have security levels. Whether a subject can access an object or other subjects depends on the security levels of both the subject and the target of the access request. In 1970s and 1980s, a lot of research attention was paid to MLS and the most famous model that implements MLS are the Biba model and the Bell-LaPadula model. The Bell-Lapadula model focuses on confidentiality, so it is used mostly in military and government organizations. The Biba model focuses on integrity, so it is used mostly in industry.

2.5.2 Access Control Matrix

The access control matrix (ACM) uses a matrix to store the rights a certain subject has over a certain object. Let S be the set of subjects, O be the set of objects and R be the set of rights. A matrix M is used to describe the rights each subject has over each object or subject. Let us make each row is about a subject and each column is about an object, then each matrix entry is the access rights that subject has for that object. That is, if $s \in S$, and $o \in O$, then $M[s, o] \subseteq R$ is the set of rights s has over o.

ACM is very straightforward to understand and it can be used for any specific security policies, but as the number of files and users get larger, the matrix might take too much space. Especially when most of the entries in the matrix are either blank or default, it is really not space-efficient to store the whole matrix. There are two optimizations of ACM model. One is Access Control Lists, and the other is Capabilities.

2.5.3 Access Control Lists

An access control List (ACL) is like one column of the access control matrix. Each object has associated with it a set of pairs, with each pair containing a subject and a set of rights [1]. The subject has the defined rights over the object. One example would be *acl(file a) = {(p1, {read,write}), (p2, {read})}*.

This means p1 can read and write file a; p2 can read file a. UNIX's permission bits can be seen as an abbreviated version of an ACL. The complete version of ACL is supported in some UNIX/Linux systems. ACLs are an addition to the standard UNIX file permissions bits for User, Group, and Other. ACLs provide fine-grained control over who can read, write, and execute files. This can all be done without the use of UNIX groups and their associated administrative overhead. Windows systems also use ACLs.

2.5.4 Capabilities

The Capabilities model is another variant of ACM. If we think of an access control list as a column in the access control matrix, then a capability would be the row of the access control matrix. Each subject has associated with it a set of pairs, with each pair containing an object and a set of rights. The subject associated with this list can access the named object in any of the ways indicated by the named rights [1]. One example could be

cap(process a) = {(file 1, {read,write}), (file 2, {read})}.

This means process a can read and write file 1. And it can read file 2. The capabilities model is supported in Linux starting from kernel 2.2, and it is also supported by Secure Linux Framework.

2.5.5 Domain and Type Enforcement (DTE)

DTE [3, 38, 4, 5, 6, 7, 8, 9, 10, 11] is based on type enforcement (TE). TE groups the system into subjects and objects. An invariant access control attribute called a domain is associated with each subject, and another invariant attribute called a type is associated with each information object [3]. DTE uses two tables, the Domain Definition Table (DDT) and the Domain Interaction Table (DIT). The DDT defines the allowed interactions between

domains and types, that is, subjects to objects. The DIT defines the allowed interactions between subjects. So whether an access request is granted or not depends on the related settings in these two tables. If the access attempt is defined in the table, then the access is granted, otherwise it is denied. TE can be seen as an improved version of ACM, in that it reduces the matrix size by grouping subjects into domains, and objects into types.

On top of TE, in DTE, a domain is defined as a tuple (entry point, access rights to object types, and access rights to subjects in other domains). Entry points are executables. When these executables run, they are in this specified domain. A process running in domain A may transit to another domain only by executing one of B's entry point programs [6]. DTE improves TE in two ways: 1. DTE defines its own language, DTE language (DTEL) to specify the access control configurations. 2. DTE file security attributes are not stored one-to-one with files on disk [7], but using an implicit typing mechanism based on the directory hierarchy [35]. This separates the implementation of these attributes from the file system and makes DTE more practical.

2.5.6 Role Based Access Control (RBAC)

With RBAC [12, 13, 14, 15, 67, 59, 60, 61, 62, 63, 64, 65, 66], administrators create roles according to the job functions performed in a company or organization, grant permissions to those roles, and then assign users to the roles on the basis of their specific job responsibilities and qualifications [59]. A user might have several roles, and can change from one role to another. The roles can have new permissions or revoke old permissions. The central notion of Role-Based Access Control is that users do not have discretionary access to enterprise objects. Instead, access permissions are administratively associated with roles, and users are administratively made members of appropriate roles [12]. No consensus on the requirements of RBAC exists, so there are different variations of RBAC. Some RBAC can set mutually exclusive roles while some might give a user the right to grant access, which makes it discretionary. RBAC can support the least privilege principle and the separation of duties principle.

RBAC is not just one model; instead it has a family of models that form a family tree [12]. Each RBAC belongs to one of the four categories, base model, role hierarchies, constraints and consolidated model. The base model is the minimum requirement for an RBAC system; it is the root of the family tree. Role hierarchies and constraints are separately built on top of the base model, that is, they both include the base model. The role hierarchy model adds hierarchies where inheritance happens between roles. One role may contain other roles. For example, some transactions or operations might be the same for all the users. It will be simple to associate a role with these operations. This role can be included in other roles. This forms a role hierarchy. The constraints model adds constraints to the model. For example, to support "separation of duty" or "mutual exclusion", constraints need to be put on the assignment of users to roles [60]. The consolidated model is built on both hierarchy model and constraints model, so it includes both.

DTE and RBAC are similar in that DTE groups users into domains, while RBAC groups users into roles, and these domains or roles decide what rights they have over objects. People also combine these two together [14, 15] in some cases to provide finer grained access control than plain DTE or RBAC.

RBAC is designed for enterprises or government where we can divide the functionalities into distinct roles, and it is well-accepted and used in government, bank, health care and other commercial organizations. RBAC is built into Solaris (since S8) and supported by Sun, AIX of IBM, HP-UX and SELinux. RBAC is so popular that in 2000, NIST called for a unified standard for RBAC. This standard was revised in 2004, and they are calling for another revision in 2009.

2.5.7 Locks and Keys

The locks and keys technique combines features of access control lists and capabilities [1]. A piece of information (the lock) is associated with the object and a second piece of information (the key) is associated with those subjects authorized to access the object [1]. The key also contains the information about how the subject can access the object. When

a subject wants to access an object, the key is checked to see if it corresponds to any locks of the object. If it matches, access of the allowed type is granted. Cryptography is used in almost all key-lock systems.

2.6 HRU Model and the Safety Problem

One of the most important problems in access control systems is the decidability of safety problem. A protection system is defined to be safe with a permission or a right, only if the permission or right could not be leaked by the system. Harrison, Ruzzo and Ullman [1,131] gave a formal definition of the safety problem on a generalized formal model (HRU model) of protection systems. The results show for HRU model that it is not decidable whether a protection system is safe with a right.

2.6.1 HRU Model

In HRU model [131], a protection system is defined as a state-transition system on a set $\boldsymbol{O}$ of objects, a set $\boldsymbol{S}$ of subjects and an access control matrix $\boldsymbol{A}$. A state $\boldsymbol{T_i}$ is a triple ($\boldsymbol{O_i}$, $\boldsymbol{S_i}$, $\boldsymbol{A_i}$). The protection system has two parts: a finite set of generic rights $\boldsymbol{R}$ and a finite set $\boldsymbol{C}$ of commands. The commands has the following forms.

command $\alpha(\boldsymbol{X_1}, \boldsymbol{X_2}, \ldots, \boldsymbol{X_k})$

if

$\boldsymbol{r_1}$ in $\boldsymbol{A}(\boldsymbol{Xs_1}, \boldsymbol{Xo_1})$ and

$\boldsymbol{r_2}$ in $\boldsymbol{A}(\boldsymbol{Xs_2}, \boldsymbol{Xo_2})$ and

. . .

$\boldsymbol{r_m}$ in $\boldsymbol{A}(\boldsymbol{Xs_m}, \boldsymbol{Xo_m})$

then

$\boldsymbol{op_1}$

$\boldsymbol{op_2}$

. . .

$\boldsymbol{op_n}$

end

The X_1, X_2 , . . ., X_k are formal parameters. Each op_i is one of the following six primitive operations.

Create subject ***s***; create object ***o***; delete subject ***s***; delete object ***o***; enter ***r*** into ***A***(***s***, ***o***); delete ***r*** from ***A***(***s***, ***o***). The states of a protection system change on the execution of defined commands from one state (O_i, S_i, A_i) to another (O_j, S_j, A_j).

2.6.2 The Safety Problem

The safety problem is defined in HRU as "the user should be able to tell whether what he is about to do (give away a right, presumably) can lead to the further leakage of that right to truly unauthorized subjects" [131]. More specifically, given a protection system and generic right r, we say that the initial configuration Q0 is unsafe for r (or leaks r) if there is a configuration Q and a command α such that

–Q is reachable from Q0

– α leaks r from Q

We say Q0 is safe for r if Q0 is not unsafe for r [131].

Two conclusions from HRU are:

1. There is an algorithm which decides whether or not a given mono-operational protection system and initial configuration is unsafe for a given generic right r.
2. It is undecidable whether a given configuration of a given protection system is safe for a given generic right.

After the HRU, there has been considerable progress on the safety problem. However, the HRU conclusion #2 still holds. With most protection systems, the safety problem is still undecidable. But for a specific protection model, it might be decidable. Researchers devoted their efforts to find protection models with decidable safety. While at the same time, these models should have sufficient expressive power to define different policies. Several well-known protection models with decidable safety has been presented since HRU,

such as the take-grant protection model [133,135,136,137], Schematic Protection Model [127,132] and Typed Access Matrix Model [134]. Li and Tripunitara proved that the security problem in most DAC models is decidable [129,130].

2.7 Network Firewall

The term "firewall" was in use by Lightoler as early as 1764 to describe walls which separated the parts of a building most likely to have a fire (e.g., a kitchen) from the rest of a structure [70]. In a computer network, a firewall is a system or a group of systems that enforces an access control policy on network traffic as it passes through an access point [71]. A firewall system can be set up to permit or deny the passing of a packet, encrypt, decrypt or break up and recreate a passing packet, or perform access control on application level.

The most widely used firewalls can be grouped into three types, ordered by increasing complexity: packet filtering firewalls, stateful inspection firewalls and application proxies.

Packet filtering firewalls perform access control based on the header of the packets. They apply certain defined rules on the header to determine whether to forward the packet or discard the packet. These firewalls are configured to filter both incoming and outgoing packets. Sometime this type of firewall is called a static firewall.

Stateful inspection firewalls keep track of information about the states of currently established connections. So on top of checking the header of the packets, the rules of this type firewalls include information about the connection states.

Application proxies are the most sophisticated firewalls. The previous two types of firewalls sometimes are called packet filters. In addition to the features of both packet filtering firewalls and stateful inspection firewalls, application proxies can do application level checking such as anti-virus scanning and content filtering. Another important feature

of proxy firewalls is that they break up the packets and recreate them. This feature makes proxy firewalls more secure. One drawback is that proxy firewalls are slow compared with the other two types of firewalls.

2.8 Current File System Protection

In UNIX/Linux systems, almost everything is in the form of a file. The types of file include ordinary files, directories, symbolic links, device files, FIFOs and sockets. The protection scheme needs to protect not only the system files, but also the user sensitive files. For these files, we need to protect both confidentiality and integrity.

2.8.1 Basic Security Scheme

The basic security scheme in Unix/Linux has five parts.

1. Login account and password.

They are used to get the users user ID(UID) and group ID(GID), which would be used later to decide whether the user can access certain files. The account information is saved in a password file and the password is saved after encryption.

2. Permission bits, setuid and setgid bits.

The permission bits on each file defines what rights the owner, users in the same group and all the other users have over this file. The permission sets are set by the root and the owner. Setuid and setgid enable a user to temporarily run an executable with the permissions of the executable's owner or group.

3. The super user.

Also known as the root, the super user has full permissions to access all files on the system. That is, the super user can execute any file that has any of its "execute" permissions turned on, and can access, read, modify or delete any file and any directory. The super user can

change the permissions on any file too. One major type of attacks is to get the power of the super user.

4. Encryption and decryption.
Encryption and decryption are used for authentication and authorization. We can encrypt files and communications using different encryption and decryption algorithms. The password verification also uses encryption. For instance, GNU Privacy Guard (GPG) is free software used to encrypt and decrypt files together with a password for the Linux operating system.

5. Chroot jail.
When the "chroot" system call is called, the calling process's root directory is modified and reset, and any future system calls issued by the process will see that new specified directory as the file system root. This makes it impossible to access files and binaries outside the tree rooted on the new root directory. But there are two problems with "chroot" jail. One is that the process will need various files and program to do some operation. So we need to set up some support environment. The other is that it is possible that the user can break out from the jail.

2.8.2 Unix/Linux Security Modules and Extensions

On top of the basic file system protection scheme, there are some security extensions or security services that are supported in UNIX/Linux system. Some UNIX/Linux systems might also support one or a combination of ACL, Capabilities, RBAC and DTE. We briefly explain each of them in chapter 3. Below are some well- known security extensions and enhanced access control implementations.

1. Linux Capabilities. Linux Capabilities is started as an effort to limit the privileges of a process when it runs with the privileges of the super user. The privileges associated with the super user are broken into distinct units, these are called capabilities. Each capability is just a bit in a bitmap which is either set or unset. A process has three sets of bitmaps called

the inheritable set, permitted set, and effective set. When a process tries to do a privileged operation, the operating system will check the appropriate bit in the effective set of the process to determine if the access should be allowed. The permitted set defines the capabilities the process might have in both effective set and inheritance set. The inheritable set defines the capabilities of the current process that should be inherited by the program executed by the current process. For instance, if the current process A executed program B, then the process of B will inherit the inheritable set of the capabilities of A.

2. Security-Enhanced Linux (SELinux). SELinux was originally a development project from the National Security Agency (NSA), Secure Computing Corporation (SCC) and others. Built on LSM framework, SELinux supports a set of access control policies, such as MAC, DTE and RBAC. SELinux is now a standard component of non-commercial distributions of Linux, such as Fedora, Debian GNU/Linux, Gentoo Linux etc.

3. Domain and Type Enforcement (DTE) Linux. Domain and Type Enforcement for Linux is implemented as a kernel (v2.3.28) [38] patch. Other than the patch, DTE is also supported by SELinux.

Chapter 3

Related Work

The firewall model for file system protection is novel, but a lot of research effort has been devoted to fine-grained access control models or enhance the current access control scheme in Unix/Linux systems. We will describe these research works and research in file system redirection which we implement in our firewall model.

3.1 Linux Security Module

The Linux security module (LSM) [28, 29, 30, 31, 32, 34] framework provides a general framework to support access control modules which might implement an access control model, such as DTE, RBAC, ACL or Capabilities. LSM includes security hooks in the kernel and the security modules which are loaded in the kernel. The so-called security hooks are just lines of code calling functions in the security module, and typically are not lengthy. These hooks can be task hooks, program loading hooks, file system hooks, IPC hooks, network hooks and other hooks.

When there is an access request, the system first does the normal DAC check. Then it turns to the security modules, through the security hooks, to check whether the request should be granted or denied. Then control turns back to the system. Under LSM, the security modules, which implement security policies, can be loaded into the kernel as kernel modules. Thus, to apply a different security policy, we just need to load a different security module. If no security modules are loaded, no extra access control is applied.

Security-Enhanced Linux (SELinux) is built on LSM and it is the most popular security extension so far for Linux. Since it is built on LSM, it can support different security policies.

SELinux is a bundle of kernel modules and user-space configuration tools which implement three different types of MAC [37]: Type Enforcement (TE), Role-Based Access Controls (RBACs) and Multi-Level Security (MLS), which defines access controls against objects based on data classification (sensitivity). SELinux was originally a development project from the National Security Agency (NSA) and it is an implementation of the Flask operating system security architecture. Flask focuses on the concept of least privilege, which gives a process exactly the rights it needs to perform its given task. NSA integrated SELinux into the Linux kernel using the Linux Security Modules (LSM) framework. The SELinux code was integrated upstream to the 2.6.x kernel in 2003.

AppArmor (Application Armor) is another set of security modules built on LSM from Novell. It is a parallel product to SELinux and is also a MAC model. To protect the operating system and applications from both known and unknown vulnerabilities, AppArmor uses application behavior enforcement. AppArmor is built on the assumption that the single biggest attack vector on most systems is application vulnerabilities. If the applications behavior is restricted, the behavior of any attacker who succeeds in exploiting some vulnerability in that application also will be restricted [37]. In AppArmor, each program is associated with a security profile. AppArmor then uses security profiles to restrict the capabilities of the program. AppArmor pre-builds profiles for popular operating-system services and common applications, including Web, e-mail and remote-login applications. AppArmor is also integrated into the LSM module for the Linux 2.6 kernel.

3.2 Rule-Set Based Access Control

Rule-set based access control (RSBAC) [68, 69] is an open source security extension based on the Generalized Framework for Access Control (GFAC) by Abrams and LaPadula. RSBAC is a kernel based access control scheme, and can be configured with different security policies. In the RSBAC system, the access control system of the Linux kernel has three parts, the AEF (access enforcement facility) component, the ADF (access control

decision facility) component and the ACI (Access control information) module. Access requests first go through AEF and AEF will gather information on the requests and send the access requests together with that information to ADF. ACI is responsible to collect information on the security attributes of processes, of users and of all resources that are needed. ADF is the part that implements different security policies, so after AEF sends the access requests to ADF, ADF will check the policy rules together with information obtained from ACI, then make a final decision and send the decision back to AEF. AEF is policy independent, while ADF and ACI are policy dependent. The rules are expressed with a language more like a programming language rather than mathematical notations. Some security policies are implemented in ADF, including MAC and ACL.

3.3 Attribute Based Access Control

Attribute based access control (ABAC) [77, 78, 79] started in systems where the number of users is huge and the users are dynamic, such as Internet. In these systems, it is hard to use user identity to meditate the access control. Instead, the attributes of the users are used for the users to gain access to certain resources. ABAC is used in RBAC to assign the users to roles according to the attributes of the users [78]. ABAC is extended to the attribute based access matrix model in [79]. In this model, each subject with its attributes is associated with a row and each object with its attributes is associated with a column. Each cell in the matrix is the access rights the subject have over the object. From the paper, the attributes of the subjects look like unchanged since the time of creation.

3.4 Sandboxing

Besides the previous security extensions and models, Sandboxing is another technique that is related to our research. Sandboxing is a technique to use a sandbox, which is a confined restricted environment, to execute an un-trusted program. Even if the un-trusted program is infected, the harm that can be done is localized in the sandbox. The program should never be able to access anything that is outside of the sandbox. To set up a sandbox, required files or services must be included in the sandbox. This, however, is not a trivial

task. The “chroot jail” in UNIX is one of the sandboxing applications. Sandboxing can provide decent security, but comes with the cost that some software could not be executed outside of the sandbox even when harmless. A related technique is a honeypot, which is the opposite of a sandbox. A honeypot is a closely monitored computing resource that we want to be probed, attacked or compromised [50]. It is a technology to gather information about attacks.

3.5 Redirecting FileSystem

The Redirecting File System (RedirFS) [48, 49] is built in the virtual file system (VFS) layer and can be seen as a new layer between the original VFS and the file system drivers. It is implemented as an out-of-kernel module for Linux 2.6. The RedirFS by itself does not provide any additional functionality. If it is loaded into the Linux kernel, it just occupies some memory space and does practically nothing. The RedirFS is intended to be used by the so-called filter which is a Linux kernel module (LKM) that uses the RedirFS framework. Each filter can add some useful functionality to the existing file systems, like transparent compression, transparent encryption, merging contents of several directories into one, allowing writing to a read-only media and others. While the RedirFS is located in the VFS layer, filters can be used generally in all file systems (e.g. ext2, nfs, proc) [48, 49].

3.6 Other Proposals

There are other great ideas about how to improve the access control in Unix/Linux systems.

Marie and Bruce [23] proposed to provide access control to objects in object-oriented software. Their proposal used protection domains and user authentications to protect the objects by restricting which methods a user can operate. An object instance can reside in a secure domain and the methods that operate on that object can only be applied if the user invoking those methods has the appropriate authority to do so [23]. This idea is very similar to RBAC. The methods are similar to the concept of transactions in RBAC. If a user does not have permission to access the methods, the user cannot access the objects. The only

difference is that a user has to have access to the objects as well in this model. In this model, a type controller table is used to specify which user has the access right to certain methods. But this only improves the security over object-oriented software, not on the whole file system.

Trent Jaeger and his colleagues propose using inter-process communication (IPC) redirection [26] to enforce access control policies for micro-kernel systems. A redirection controller is needed perform the access control checks and to decide whether the current IPC communication is allowed. The redirection is used to redirect the IPC communication. Processes are grouped into redirection sets, and each set has its own chief and all the redirections happen between the chiefs. The chief is responsible to check on access and decide whether to forward the IPC to the right process.

Access control based on execution history [20, 25] is another type of access control model. Whether the current access request is granted depends on the execution history of the current subject. An execution monitor is required to keep track of the history of accesses the subject has made. Then it grants or denies the next access request based on the recorded history. With different policies, different data has to be kept. This method can be used to implement certain policies, such as the Chinese Wall policy and low-water-mark policy.

A stack based model is also used for general access control policies. The idea is that the access right that the current piece of code has is the intersection of all the static permissions of pieces of code on the stack. If A calls B, then when B executes, it only have the rights shared by both A and B. This is used widely in Java Virtual Machines (JVM) and Common Language Runtime (CLR).CLR is a counterpart of JVM in Windows2000 and WindowsXP.

Christian Payne [47] proposed cryptographic access control architecture, called the vaults security model, to limit the power of root. The vaults security model has user vaults for users and system vaults for the kernel and administrator. User vaults are for sensitive data and tickets that are used to access cryptographically protected files, so whether a user could

access a certain file depends on whether the user has a ticket for that specific file. The vaults security model is a lock-and-key model. A basic public key infrastructure is used in this model. A highly privileged user, called the prime user, is still needed to perform some administrative work. But unlike the root in the traditional systems, the prime user privileges are only for limited tasks. No other system services are run with the privileges of the prime user. The prime user's privileges are not determined by identity, but are determined by the possession of special tickets.

Chapter 4

File System Firewall Model

We begin this section with the motivation and the definitions of the firewall model and its features. Then we show a formal definition of file system firewall model.

4.1 Motivation

4.1.1 Accommodate Complicated Access Control Requirements

In most access control systems, access decisions are made based on one of the credentials of subjects, such as identity of the user (UID) in DAC, or domain type in DTE. With access control systems applying widely to different information systems, these systems are found inadequate to manage more complex and flexible access control requirements. For example, a health care system may need the affiliation of user accounts, the time of day, and the location of the current computer in the network to decide whether the access requests should be granted.

In any access control system access control decisions are made based on the certain system state(s). The user ID, process domain or account role is all one of the states of subjects in the whole system state. The whole system state that is relevant to access control may include certain states of subjects, objects and system environment. More complex access control requirements need more states to more closely define. One example access control policy may be that only myApp can access the target file F, when size of F is below 9M, and the accessing time is between 9am to 5pm. Under other conditions, the access requests will be denied. When we include the relevant attributes of subjects, objects and environment in the access control systems, access control policies can be specified on one, two or any subset of these attributes. The access control policies are defined on a multiple-

dimension space rather than a single attribute. Including attributes of subjects, objects and environment in the access control systems brings us another flexibility, that is, it makes it possible to associate different access control policies with different objects. An application configuration file may only be modified by its own application, that is, the process subject. A backup program may only be started by certain account on certain time period. A file may not be read when its size changed. These access control requirements are on a different subset of the attributes of the system states.

4.1.2 Integrity Protection of the File System

Early security models are for the military systems, so the confidentiality was the center of the security protection. In the commercial world and the civil world, the integrity of the data can be more important than the confidentiality of the data. We want the protection model to protect not only the confidentiality of the system, but also the integrity of the system. In computer security, there are three goals of integrity protection [75]: prevent unauthorized modifications, maintain internal and external consistency, and prevent authorized but improper modifications.

The traditional UNIX access control scheme can provide some degree of integrity protection of the objects in that only authorized subjects can access the objects, but it could not protect the objects from improper modifications caused either by attackers or by human errors. We have to trust the authorized users for operating on the objects properly. For example, if a user is authorized to modify certain files, the user will have write access to these files. We will have to trust the user to modify the data correctly, not to destroy the files. The internal and external consistency is not supported either. To achieve the consistency, application level programs have to be used.

The enhanced access control models still do not provide direct integrity protection to prevent improper modifications and inconsistency. In Capabilities, the privileges of the root are partitioned to make sure a certain process can only run with a part of the root privileges. The goal is to confine the root privileges, so no integrity control is added. In

DTE, some integrity protection is provided indirectly by making sure subjects in certain domains can only access the objects in specified types. So only authorized subjects can operate on specified objects.

With RBAC, a user is associated with roles which decide whether the user can access certain objects. It still focuses on "only authorized user can operate on specified objects". RBAC can provide internal and external consistency protection, but at the application level, not system level. We just have to trust users or processes to perform authorized and proper modifications. But users can make mistakes and the programs can have bugs and can be exploited by attackers. In these models, the integrity control could not meet the goals of integrity protection.

There are actually some security models that focus on integrity protection. Two well-known integrity models are Biba model and Clark-Wilson model. Biba works better in systems where subjects and objects have clear integrity levels. Clark-Wilson model is an application level model. Both do not quite fit general UNIX systems. We include these two models in this section to make our discussion about integrity complete.

Biba model is a multiple-level security model. In Biba model, subjects and objects have strict integrity levels. Subjects cannot read objects with lower integrity level, cannot write to objects with higher integrity level, and cannot execute another subject with higher integrity level. Information only flows one way. Biba model focuses on protect data from being contaminated. Therefore, Biba model only addresses the first goal of integrity protection, that is, to prevent unauthorized modification.

Clark-Wilson model [76] groups the objects into Unconstrained Data Items (UDIs) and Constrained Data Items (CDIs). CDIs are data that subject to integrity control, while UDIs are not. Two kinds of procedures are defined. One is Integrity Verification Procedure (IVP), which is used to test the integrity constraints on CDIs. The other is Transformation procedures (TP), which changes CDI sets of the system from one valid state to another.

IVP is executed before and after the execution of a TP to conform that the system is in valid states. A user cannot access CDIs directly. It can only have execution right on the associated TPs, which can operate on the CDI sets. Clark-Wilson model defines 5 certification rules and 4 enforcement rules to ensure the integrity of data items. In this model, all the three goals of integrity protection are provided.

Clark-Wilson model addresses the security requirements of commercial applications such as bank systems, hospital systems and other computer systems where all the data are stored in databases and users can only access the data through well-defined transactions. In this model, users do not have access to the data. They just have execution access to certain transactions. Other than using these transactions, users can not access any of the data. In UNIX, it is the user that has access rights to the objects. A program will inherit the access rights from the user who executes the program. Polk [80] describes an approximation of Clark-Wilson model under UNIX system, in which setuid programs are used to bypass this problem. But by doing so, too many setuid programs will be in the system. It comes with other problems as well [1], so it is not practical.

There are other issues with Clark-Wilson model under the UNIX system. TPs are executed serially, rather than several at once [76]. IVP is executed before and after the execution of a TP to conform that the system is in valid states. In UNIX, to make sure processes run serially and maintain the data consistency, extra mechanisms are needed. Since the root can modify the Clark-Wilson triples (user, TP, CDI set), and have execution rights to any TP, separation of duty is hard to achieve.

In the firewall model, integrity protection will include preventing improper modifications which are not provided by any other access control models under UNIX systems. The improper modification might be caused by attacks or human error. For example, an attacker might try to overwrite important configuration files to disable the system. An administrator might add a user account to the password file without specifying the right values by mistake. These improper modification to the objects could not be detected by other models under

UNIX systems. Certain specified integrity constraints will be applied by the firewall model to control how certain files can be modified.

4.1.3 Ease of Administration

We want the protection model to be easy to use for the administrators. In any security systems, the most important aspect is the people. A security model should be friendly to the administrators, otherwise people will have a hard time to adopt it and will eventually stop using it. Sometimes security only appears important to the users after a major successful attack. When nothing happens, a security department does not get as much attention as other departments. It is important that a security model is easy to understand, set up and maintain for administrators.

Firewall model works in the way that administrators are familiar with. Administrators have lots of experience with network firewall. They understand how the network traffic and filters work. They can set up, configure and maintain the firewall with ease. File system firewall model shares the same abstractions with network firewall. File system firewall is an access control model that enforces access control policies to the file systems. Just like a network firewall enforces access control policies between two or more networks. Access requests to files can be allowed and denied. Network traffic can be permitted passing or blocked. The integrity control we discussed earlier that allow only certain modifications to the files is like the content filtering in network firewall. Same as the network firewall, to use the firewall model, they need to set up the configuration on filter rules. In Firewall model, filter rules are specified over the object. This is similar to the ACL model, where each object is associated with an access control list. ACL has been around and supported in certain UNIX systems, so it is not something new to the administrators. All in all, administrators do not need to think differently and we expect them to be comfortable using the file system firewall.

4.1.4 MAC for Specified Subjects and Objects

Many attacks follow the same pattern: exploit some vulnerability to make a running process execute the malicious code of the attacker. With DAC, the attackers can have all the access rights the running process has. The situation gets worse with the super user, setuid and setgid mechanisms. Once the attacker gets a process running with root privilege to execute malicious code, severe damage can be done to the system. This is why the recent access control models try to confine the power of the super user. With MAC over DAC, the access rights are not only at the discretion of the owner or super user, but also are constricted by specified security policies. So we can limit the damage of these attacks.

One goal of most existing security models in UNIX system is to confine the super power of the root and provide MAC control over DAC. POSIX Capabilities model is such a model. It splits the privileges of root into capabilities that certain processes have. Even if the root starts a certain process, the process can only perform operations in the effective set of its defined capabilities. DTE confines the root by making many root programs be executed in restrictive domains that only allow access appropriate to each program's assigned responsibilities [6].

To provide the MAC protection over DAC, the current access control models tag all the subjects and all the objects with security labels and then specify the policies according to these tags and labels. We want to protect sensitive data. However, we have to tag everything in the system. This is a waste of time and resource especially when we take into account that most of the time the sensitive data are only a very small portion of the system. We want our security protection model to support MAC over DAC for only specified processes and files, such as system processes and files and sensitive user data. Security is always a trade-off. We want our model to be flexible for the trade-off. That is, our model can be set up to provide protection for all the subjects and objects in the system, or only sensitive system or user processes and files. By supporting MAC over DAC for only specified subjects and objects, we can get rid of the unnecessary security operations and improve the performance of security systems.

4.1.5 Redirection

In network firewall, packages can be forwarded to different networks. We borrow this idea from network firewall and include it in the file system firewall. Its parallel operation in the file system firewall model is redirection. If an access request is illegal, instead of just denying the access, we can redirect the access request for one object to another phony object with a certain degree of transparency. With redirection, we can monitor, track and study the attacks without any harm to the system. At the same time, attackers get false feedback from the system. So redirection can be used to set up a honeypot. This is a unique feature that we have not found in any other access control models.

4.1.6 Flexibility

One of the features that most current access control models lack is flexibility. If we want to allow permission only for a short period of time, or only under certain conditions, in current security models, there is no direct and straightforward way to make this happen. We include the attributes to form propositions to make this fairly easy in the firewall model. In package filtering firewall, whether a package can pass the firewall or be blocked is decided by checking the header of the packages. If certain value or value combination matches the filter rule, the appropriate action will be taken, that is, to forward or to block. The file system firewall works the same way: according to the values of the attributes, the access request might be allowed or denied. Another type of flexibility the firewall model can offer is the run-time update of the filter rules, just like the network firewall. This feature is not discussed explicitly in most of the current security models. Since each filter rule is for one object only. Changing this particular rule does not affect any other rules. With appropriate implementation, run-time update of the filter rules can be supported.

4.2 The Definitions of Firewall Model

The file system firewall model is an access control model. Access control is one of the fundamental problems besides authentication and authorization in file system security. It enforces security policies, mediates access requests to decide whether the access requests

should be granted or denied. It acts as a security layer on top of the current protection scheme. Any access request sent to the file system has to go through the firewall first. It is an adoption of the familiar network firewall protection model, used to control the data that flows between networked computers, toward file system protection. This model supports decisions of access control based on a set of filter rules on attributes related to access requests: attributes of the subjects, the system and the objects. Access control policies are expressed in a rule set using predicates of attributes. Each rule is specified over an object of access requests. We call the rules filter rules just like in the network firewall. The firewall either pass the access requests to the file system (allowing access or redirection) or drop them (deny access) according to matched filter rules.

4.2.1 File System Firewall (FSF)

File system firewall (FSF) is an access control system we have developed on Unix/Linux systems that decides whether an access request to an object on file system should be granted, denied or redirected based on a set of filter rules on attributes. We name the system file system firewall because of the similarity it shares with a network firewall. A computer network firewall enforces an access control policies on network traffic as it passes through an access point [71] and manages the incoming and outgoing network traffic by analyzing the data packets and decides whether they should be allowed through, discarded, or forwarded based on a rule set [119].

The objects or targets of access requests are normal files on Unix/Linux system: regular files, directories, symbolic links, hard links, named pipes, sockets, and devices. The attributes could be any attributes on subject, object or the system, such as the DTE domain of the subject, size of the target file, or time of day. FSF can be used as an extra layer of protection on top of current access control systems.

4.2.2 Filter Rules

Filter rules are used to specify access control policies. The format of filter rules is listed below. In UNIX-like file systems, a file is identified by a pair of (inode number, partition number). In the filter rules of FSF, the pair of (inode number, partition number) is used to identify an object.

FR = (Object, Access type, Predicates, Action, Action parameter) where
Object is the target of the access request;
Access Type is a set of the access request type, such as read, write, execute and others,
Predicates is the conjunction of a set of predicates defined on the attributes of the request subject, object and the current system;
Action is the action to take when the predicates are all true. It could be "redirect", or "allow", "deny", or any other action; and
Action Parameter defines parameters that might be required to perform the indicated action.

An example only_allow filter rule is:
FR1 = ((inoS1, partS1), {read}, {program = App, hour >9, hour <17}, only_allow, {}).
This rule defines that only program App can read the file identified by {inoS1, partS1} during 9am-5pm. Any other attempt of reading the object by process that is not App or not during work hours will be denied.

An example redirection filter rule is:
FR2 = ((inoS2, partS2), {}, {fileSize > 100M}, redirect, {(inoD, partD)}).
The rule specifies that if the size of target file is more than 100M, then an access request to the file with inode inoS1 in partition partS1 will be redirected to the file with inode inoD in partition partD.

4.2.3 Three Types of Filter Rules

So far, we defined three types of filter rules according to the action type: allow, deny and redirect. Allow rules specify situations under which the access requests should be allowed. Deny rules define situations that access requests should be denied. On one object, either allow rules or deny rules may be defined, but not both. If allow rules are defined, then by default, any access request should be denied unless the access request matches any of these allow rules. Similarly, if deny rules are defined, then by default the access requests should be granted.

Redirection is an extra feature of FSF comparing with other access control systems. Instead of merely denying an illegal access request, FSF redirects the access request to another node in the file system. The redirection is transparent to users. This can be used to study and analyze illegal access attempts.

4.3 Features and Usage

4.3.1 MAC Control

Many attacks follow the same pattern: exploit some vulnerability to make a running process executing the malicious code of the attackers. With DAC, once the attackers attack a process running with root privilege to execute malicious code, severe damage can be done to the system. This is the reason that one of the goals of most security extensions on Linux/Unix system is to confine the unlimited powder of root account by providing MAC control. FSF is a MAC model. Whether an access request will pass the firewall is decided by the loaded filter rules. Root account does not have all the privileges as in a traditional access control scheme.

4.3.2 Conditional Access Control with Multiple Attributes

Most access control systems are based on the value of a single attribute, such as user identity or process domain, to make access control decisions. FSF extends this single attribute to multiple attributes with conditions. This puts access control into a system

scenario, and gives administrators more flexibility in dealing with access control requirements more complicated than just the identity of the account. The definition of the system scenario is defined with predicates on attributes in FSF. With a single attribute system, the security policies can be seen defined on the value of attribute as an equality operation. For example, when UID=Andy, appropriate permissions are defined for a certain target. Besides the equality operation, FSF adds ">", "<", "!=" to the definition of filter rules. The attributes can be any attributes in the system, the attributes on the subject or process, the attributes on the object or target resource, and the attributes on the system.

4.3.3 Ease of Adoption and Administration

People are the more important part of an access control systems. No matter how powerful and access control system is, it is useless if no one wants to use it. Adoption to a new access control system is intimidating by its nature. On top of traditional DAC system, many access control systems are recently designed and deployed on Linux/Unix systems. However, despite their new advanced features, some of them are not used as often as expected. Some systems are complex to set up. Other systems are complicated to maintain. The adoptions to these systems have been slow. In FSF, the specification of filter rules is similar to the specification of network packet filters. Both basically include three parts: target of this filter rule, a set of propositions, and an action. Administrators are familiar with network packet filters, which would make the adoption of FSF not as intimidating as some other access control systems. Similar to ACLs, the filter rules are defined on objects with pathnames, which is another aspect administrators have known and might have used. Another advantage of FSF is that it does not have to be defined and setup once and for the whole system before use. It only requires filter rules defined on involved targets.

4.3.4 Redirection

In network firewall, packages can be forwarded to different networks. This idea is applied to the file system firewall as redirection. If an access request is illegal, instead just being denied, it can be redirected to another phony object with certain degree of transparency.

This feature can be used in monitoring, tracking and studying the attacks without any harm to the system. At the same time, attackers get false feedback from the system.

4.3.5 Usage

Like any other access control systems, FSF can be used alone to enforce access control policies. The access control policies are specified on resources. The overall FSF is open authentication. That is, if there are no filter rules defined on a specific resources, the access requests are granted by default. However, it can be open or close authentication when there are filter rules defined on a resource. If allow rules are defined, it is close authentication. Only those access requests that pass the allow rules have access. If deny rules are defined, then it is open authentication again.

FSF can cooperate with other access control systems, and serve as an extra layer of protection on top of existing access control systems. Since it is open authentication, administrators only need to specify filter rules on needed resources.

FSF can be used to simulate or implement other access control systems by using the right attributes. For example, when we choose the user ID as the only cared attribute, we can implement ACL and Capabilities. When we choose domain of the processes as the cared attribute, we can simulate DTE model. Certain extra operations may be needed to collect data on the involved attributes for some access control systems, but the FSF framework remains the same.

4.4 The Operational Specifications of the Firewall Model

We define our firewall model formally in the following operational specifications. We will describe some definitions, then define the operations of the firewall model.

4.4.1 Definitions of Basic Components

The definitions of basic components are listed below.

Definition 1: Request Attributes.

We define *request attribute* RA to be a set of attributes and their associated values related to one specific access request. These attributes could be the attributes about the subject, the object of the access request and the system environment. A request attribute is used to check whether or not to apply certain filter rules. We use an attribute name and its value pair to describe one element of the request attribute.

$$RA = \{(ni, vi): ni \text{ is the name of an attribute, } vi \text{ is the value of } ni\}.$$

Some example system attributes are listed below:

UID: User ID.

GID: User's group ID.

Name (program name): The name of the process that sends the access request.

Time (access time): The time when the access request is sent.

Owner: The resource owner, the owner of the target file.

Bowner: The owner of the binary.

EUID: Effective user ID.

EGID: Effective group ID.

Size: The size of the object.

For instance, one specific request attribute can be:

$$RA\ attr = \{(UID, 100), (GID, 100), (Name, vim), (Time, 2000)\}.$$

In this example the access request has four attributes: the user ID of the request is 100, the user group ID is 100, the program name of the current process is "vim", and the access time is 8pm.

Definition 2: Access Type.

An access request type AT is a set of all possible access rights a subject might have over an object. For instance, it can be a subset of {read, write, execute, create, delete}, such as:

$$AT = \{read, write, execute, create, delete\}.$$

It means that in the current system, all the access rights a subject can have over an object are read, write, execute, create or delete.

Definition 3: Access Request.

An access request AR is a sequence of values that defines the access request. We need all the information in the access request to decide whether the access request is granted, denied or redirected. A general form is defined as (*object, access type, request attribute*), where *object* is the target object of the access request, *access type* is a set of access request type, *access type* $\subseteq$ T, and *request attribute* is a request attribute.

One simple example is:

$$AR\ ar = (/etc/passwd, \{read\}, \{(UID, 100), (GID, 200), (NAME, vi)\}).$$

This means there is this access request with 100 as UID and 200 as GID is trying to use the program *vi* to read the file *"/etc/passwd*".

Definition 4: Attribute Predicate.

We define *P(n)* to be the predicate of the attribute with the name n in the request attribute. We use theses predicates to build the filter rules. For example, *P(UID)* is the predicate on UID, such

As

$$P(UID) : UID > 100.$$

We define *PS* to be a predicate set:

$$PS = \{P(n) | n \text{ is the name of an attribute}\}.$$

Definition 5: Filter Rule.

A *filter rule* R is a sequence of values. A general form applies to each filter rule only with different values for the elements in the sequence. The filter rules specify the security policies. Each filter rule has a specified access request object file name, an access type, a set of predicates, an action, and the parameters of the action. If the access request matches the rule, then the action defined in this filter rule will follow. All filter rules have the same form:

(*object, access type, predicate, action, action parameter*) where

object is the object file of the rule,
access type is a set of the access request type, such as read,write,
predicate is a set of predicates defined on the request attributes,
action is the action we take if the predicates are all true. It could be "redirect", or "grant"/"deny,
action parameter defines parameters the action might need.
One example could be:

R rule = (/etc/passwd, {read}, {UID! = 0}, redirect, {/phony/ect/passwd}).

In this example, the policy is that if the UID is not 0, then we redirect the read access request to the file "/phony/ect/passwd".
Let N be the total number of rules, and all filter rules in a system is a set:

$$F = \{(oi, ati, PSi, ai, api) | 0 < i\ N\}.$$

4.4.2 Operational Specifications

We define three types of operations of the firewall model: redirection, grant or deny access and content filtering. In network firewall, similar operations are forward a package, pass or drop a package and content filtering accordingly. These operations correspond to the

three types of filter rules in our firewall model: redirection rules, access rules and content filtering rules.

Operation: Redirection.

Description: For an access request, if the object file (the object) of the access request matches the object in a filter rule, all the predicates are evaluated to be true and the action of the rule is "redirect", then the access request is redirected to another file (the target) specified in the action parameter of the rule.

Let the access request be AR ar = (*o, at, v*) , N be the total number of rules, all filter rules be

F = *{(oi, ati, PSi, ai, api)|0 < i ≤ N}* and *|PSi| = Ci* . If ∃k, so that *0 < k ≤ N, o = ok* , at ∈ ati,

ak = redirect, and *P(n1)^P(n2)^· · ·^P(nCi) = True*, where each of *n1, n2, · · · nCi* is the name of an attribute in v of the access request *ar*, then the access to file o is redirected to the file target *apk*.

Operation: Grant/Deny Access.

Description: This is the basic access control rule. Only when all the predicates are evaluated to be true, the access request is granted/denied as specified in the rule.

Let the access request be *AR ar = (o, at, v)*, N be the number of rules, all filter rules be

F = {(oi, ati, Pi, ai, api), *0 < i ≤ N }* and *|PSi| = Ci*. If ∃k, *o = ok* , at ∈ ati, *ak = grant||deny*, *apk*

is empty, and *P(n1) ^P(n2)^ · · · ^P(nCi) = True*, where each of *n1, n2, · · · nCi* is the name of one attribute in v of the access request ar, then the access to file o is granted or denied as specified in *ak*.

Operation: Content Filtering.

Description: The target file of the current access request can be modified only in a specified manner.

This operation is used to provide integrity protection to the file system.

Let the access request be *AR ar = (o, at, v)*, N be the number of rules, all filter rules be *F = {(oi, ati, Pi, ai, api), 0 < i ≤ N }* and *|PSi| = Ci . If ∃k, o = ok , at = write, ak = Content*, apk is not empty, and P(n1) ^ P(n2) ^ · · · ^ P(nCi) = True, where each of n1, n2, · · · nCi is the name of one attribute in v of the access request ar, then the file can only be modified as specified in apk. Otherwise, no change is allowed.

Chapter 5

Prototype Implementation and Performance Test

In this chapter we discuss the possible approaches to implement the prototype of FSF and show the reasons of our approach. Then we explain our implementation and give the results of performance test.

5.1 Possible Approaches

We study the existing frameworks that could be used to implement our firewall model. The two most plausible approaches are stackable file systems and LSM. Unfortunately, we cannot build our firewall model using either of these frameworks. We will examine these two frameworks and show why this is the case.

5.1.1 Stackable File System

A stackable file system [42, 51, 52, 53, 54, 55, 56, 57] or stackable layers file system is a file system that is constructed from a number of independently developed layers. Each layer is bounded by a symmetric interface, meaning the interface is syntactically identical above and below [51]. Layers combine in stacks, either linear or tree shaped collections. Each layer is built on the functionality of those beneath it. The goal of stackable file system is to simplify the process of file system development by providing an extensible file system interface [57]. Each layer has its own page cache, inode and dentry caches.

Stackable file systems have some really good features, such as extensibility and flexibility. The best feature of the stackable file system is its extensibility. New services can be added into the stackable file system architecture as another layer. More importantly, each layer can be provided by different parties because of the symmetric interfaces defined in the

stackable file system. No kernel changes are needed when another layer is added. The stackable file system does need some kernel support, but once it is done, adding other layers would not require any more kernel changes.

Another great feature is flexibility. The symmetric interface defined in the stackable file systems makes the order of the layers irrelevant. This means a layer can be placed either on top or below another layer, and the functionality provided by all these layers would not change. Adding or removing a layer would not affect other layers.

But these great features come with a price. Below are some issues with the stackable file system. The biggest issue is the cache coherence problem. It includes both data coherence and meta-data coherence [55]. The data pages, the vnodes and dentries are cached at each layer, so the independent changes in different layers might cause incoherence problems between different layers. Extra mechanisms are needed to make sure they are consistent. Also with caching at each layer, additional overhead is incurred to deal with caching in each layer and cache consistency between layers. Even though we could add as many layers as we need with the infrastructure, the overhead caused by making the caches consistent would be growing with more layers and would hinder the performance.

A template wrapper file system - Wrapfs [52, 53, 54, 57] is introduced to simplify the process to add an extra layer to the Linux kernel. Wrapfs takes care of interfacing with the rest of the kernel; it provides the developer with simple hooks to modify or inspect file data, file names, and file attributes [54]. It provides interfaces with both VFS layer and the lower real file systems layer. Stackable file systems can be written from Wrapfs. So a security layer can be added into the operating system by adding and modifying codes from Wrapfs.

However, the firewall functionality does not easily layer with Wrapfs. The interface provided by Wrapfs sits above the on-disk file system, just below the VFS. The Wrapfs file system uses the inode and dentry objects passed down from VFS. If an operation is redirected, then the inode number for the target file will be determined in a file system

layer beneath the VFS; the VFS will use the object file. Hence, the VFS layer and the firewall layer will have different inode and dentry for this target file. This introduces an inconsistency that requires significant additional overhead to address. So stackable file system is not a good choice for our model.

5.1.2 Linux Security Module (LSM)

LSM [28, 29, 30, 31, 32, 34] is another possible way we could use to build our firewall system. LSM is designed to provide support for access control that supplements the traditional UNIX controls. The LSM framework contains hooks into the kernel so that an access control decision can be made by a module loaded onto the kernel. The LSM will call the appropriate module components in order to make access control decisions. To use LSM, one just needs to implement the security module and call the security service using the appropriate hooks.

Unfortunately LSM the LSM framework cannot support the redirection function. The LSM can only support yes or no decisions to grant or deny an access request. However, for redirection, we must change the inode of the current access request. Additionally, the security hooks in LSM support evaluation after the traditional UNIX access control evaluation. Efficient redirection should happen right after identification of the inode of the object file. At this point, the inode and dentry of the object are replaced with the inode and dentry of the target; then it is not necessary to modify each individual function (that requires the inode and dentry of an object file) in the kernel in order to do the redirection. The translation from object to target inode and dentry is done before the traditional UNIX checks. Since the LSM security check is after the traditional UNIX checks, it is less efficient to use the LSM.

5.2 Components

FSF has basically three components: filter unit and filter rule database in kernel, and filter rule management to load filter rules into database or remove filter rules from the database.

The filter unit and the filter rule database are in the kernel. The filter unit is responsible of checking the filter rule database, collecting data on appropriate attributes, and get an action to an access request at an accessing point.

In addition to the two components that reside in the kernel, FSF requires filter rules management program that administrators can use load new filter rules into the kernel or remove filter rules from the kernel.

5.3 Design Concern

A protection model in UNIX systems not only needs to provide the protection to the file system, but also needs to be practical. We want the prototype to have the following properties.

One property is that the prototype should show decent performance. The overhead caused by the firewall model should be reasonable.

The other property is that the prototype should make as few changes to the kernel as possible. The firewall model has to be built into the kernel. This way the performance will be better, since we avoid the frequent context switching between kernel space and user space during the access control process. Another advantage is that it cannot be bypassed. Less changes makes it easy to incorporate the model into the different versions of the kernel and makes it simple to debug and maintain the system.

5.4 The Prototype of the Firewall Model

The firewall model is implemented in the VFS layer of the Linux kernel. The implementation supports ordinary, directory, symbolic link, named pipe and socket file types. The firewall can additionally support redirection across file systems. The goal for the current version of prototype was to test the chosen implementation structure and to determine whether or not performance is acceptable. Redirection is fully supported. The

grant and deny actions fit easily within the current implementation structure and require less overhead than redirection. We will implement it next. The redirection functionality is implemented before the traditional DAC permission checks. So any access requests granted by the firewall still need the permission of the file system before it could access the file.

5.4.1 The Implementation of the Prototype

This prototype implementation is not a fully implementation of FSF. It is rather a basic start of the fully implementation, but it is sufficient to test out the idea of FSF. Our implementation of the prototype is on the Linux 2.6.37.1 kernel. We made minimal changes to the kernel. Only a few kernel files are involved in the modification. We have implemented FSF successfully on three version of Linux kernel: 2.6.18.2-34, 2.6.24.1 and then 2.6.37.1.

Two components of FSF reside in the kernel, the filter unit and the rule management module. The filter unit checks with the rule database, collects data on appropriate attributes, and determines if an access request should be accepted, denied, or redirected according to the filter rules loaded in memory. The rule management module is in charge of loading the rules into the rule database or deleting rules from the database. It also supports addition and removal of rules at run time. The rule management module is implemented as a device driver. The functions to load or delete rules are implemented as ioctl operations. Authorized processes can call these ioctl functions to load or delete the rules. In addition to the two components that reside in the kernel, the implementation of FSF requires at least two components that reside outside the kernel. The first is the filter rules stored in files. Second, a set of application programs, that call the ioctl functions provided by the rule management module, are needed to manage the rule set in memory. The filter rules are defined on path name in the rule files. However, in kernel, it is actually defined and stored as a pair of (inode number, partition number).

Each rule in the filter rule file has three parts: the object, a set of propositions and the action to take if all the propositions are true for the current access request. The action is described

by a name (*allow*, *deny*, *redirect*) and associated parameters (e.g., a target for the redirection). For example, the object might be /etc/passwd, the propositions might be "the user is not root and the program requesting access this file is /usr/bin/emacs" and the action might be *deny*. This rule mediates access to etc/passwd. When the object of an access request is /etc/passwd, this rule will apply. If the propositions are all true the equest will be denied. Otherwise the access request still must pass the traditional access control restrictions.

The operations currently supported for propositions are: ">" (greater than), "<" (less than), "=" (equal) and "!=" (not equal). The application programs are tools used to load all the rules from the configuration file into the memory and to add or remove certain rules from the memory at run time.

Theoretically FSF can support any attributes in the system. In the prototype, we include some commonly-used attributes:
UID: UID of the subject.
EUID: Effective group ID of the subject.
GID: GID of the subject.
EGUID: Effective GID of the subject.
program: The pathname of the binary file used to create the process initiating the request.
bowner: The owner of the binary file.

rowner: The owner of the target file.
size: Size of the target file.
datetime: The datetime when the access request is sent.
day: The day in a week when the access request is sent.
hour: The hour when the access request is sent.

UID, GID, EUID, EGID, program and bowner are attributes of the process, or subject. The rowner and size are the attributes of the object. The three time related attributes are attributes of the system.

We do not support time attributes on redirection. Since redirection only happens at the beginning of accessing a file, redirection in the middle of accessing a file may have confused and inconsistent results.

5.4.2 Work Flow

The work flow graph in Figure 2 shows how the firewall model deals with an access request.

1: A process sends the access request to file system. The access request goes through the file system firewall first where the firewall checks the filter rules.
2: If there is a matched filter rule, and the action of the rule is to deny, the access request is denied.
3: If no rule matches, or if there is a matching rule with an allow action, then the request is forwarded unmodified to the file system. The traditional UNIX access control checks will be performed on the object of the request.
4: If a redirection rule is matched, the access request is sent to the target file. The original UNIX access control checking will be performed to the target file.
5: If the access is granted, the process will receive the appropriate response (either against the target or object).

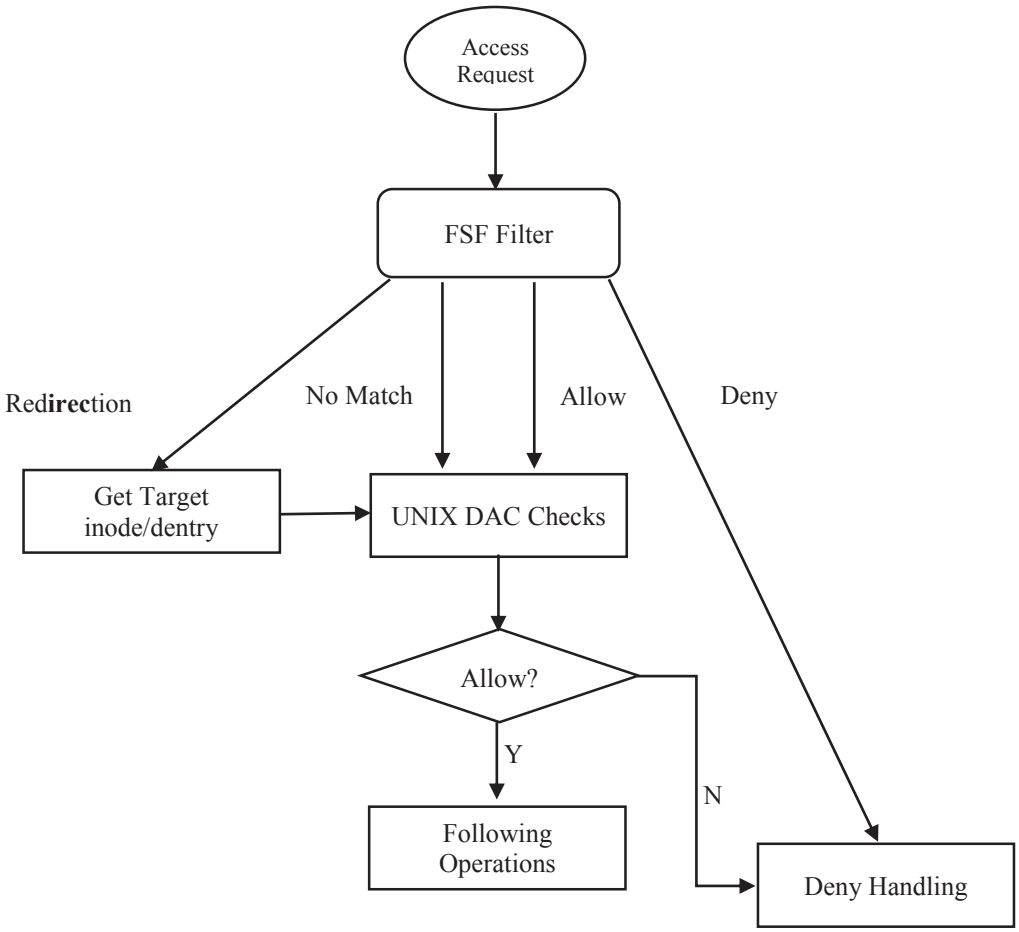

Figure 2: Work Flow of FSF

5.4.3 The Structure of the Prototype

The graph in Figure 3 shows how the components of the firewall interact with the kernel to perform access control. When there is an access request, control is transferred to the filter unit. The filter unit then collects the information about the access request and searches the filter rule database to see if there is a rule match on the target of the access request. If there is no match, the access request will pass through the firewall. If there is a match, depending on the action defined in the matched filter rule, the filter unit might deny the access request, let the access request pass through, or get the appropriate file if the access request needs to be redirected. Control then passes to the traditional access control mechanism within the operating system.

In the graph, when there is an access request, control is transferred to the filter unit. The filter unit then collects the information about the access request and checks with the filter rule database to see if there is a rule match on the object of the access request. Then depending on the action received from the filter rule database, the filter unit might deny the access request, let the access request pass through, or get the target file if the access request needs to be redirected. Then control is passed to the original access control processing in the operating system. The rule management module loads filter rules into the database or remove the filter rules from the filter rule database. This can be done at boot time or at runtime. Appendix A explains in details on the implementation of the filter unit in the kernel.

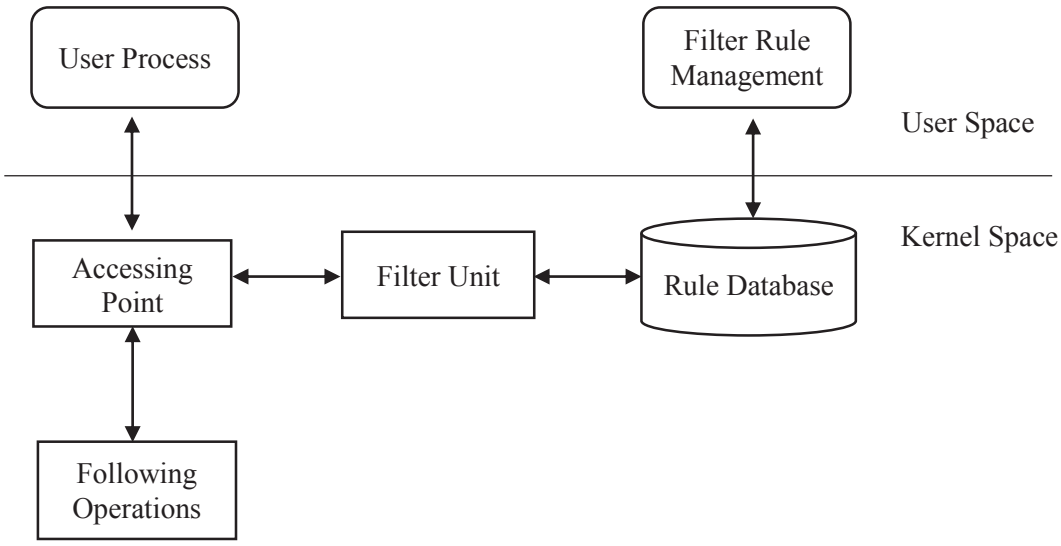

Figure 3: Components of FSF

5.4.4 Filter Rule Set Management

Since filter rules in network firewall system are complicated, expert systems have to be involved to analyze the filter rules. One concern about FSF is whether filter rules in FSF is as complicated as network filter rules. Even though FSF uses filter rules similar to network firewall, the filter rules in FSF are confined by the nature of access control in file system,

which makes the filter rules much less complicated. It is shown by discussing the possible mistakes and how to detect them.

The possible mistakes and errors can only happen in one of the following two types: in one filter rules, or in multiple filter rules defined on the same object. The discussion here is beyond syntax errors, which could be prevented.

The first type of mistakes is the logic errors between predicates in the same filter rule. For example, hour >19 and hour < 9 are in one filter rule. This type of errors can be detected by checking common set of appropriate attributes.

The second type of mistakes might occur in multiple filter rules defined on the same object. For the same object, only_allow rules and deny rules cannot coexist for the same target. Redirection rules can co-exist with either only_allow rule or deny rule. However, redirection rules are only checked at the start of accessing a file. So the rule management tool only needs to check the same type of rules defined on the same object. Only_allow or deny rules specify all situations the access requests should be granted, or specify all situations access requests should be denied. Filter rule management tools need to find repetitions and contradictions. If a new rule defines a situation that is a subset of a precious rule, they this rule does not need to be added to the filter rule database. If a new rule defines a condition which is the complement set of the condition of precious rules, then either all these rules should be removed (only_allow rules) or be substituted with one filter rule (deny rules). Filter rule management tool needs to detect contradictions among multiple redirection rules to make sure there are no intersections of situations. If two redirection rules have predicates on the same set of attributes, and all predicates on this set of attributes in one rules make predicates on the same set in the other rule true, there is a contradiction. If two redirection rules do not share any attributes, there is a contradiction.

5.5 Performance Test

A primary goal of this implementation was to determine whether or not an approach with this high level of granularity can provide acceptable performance. Our implementation was within the Linux 2.6.24.1 kernel on a machine built from the SUSE Linux distribution. The hardware platform contained an Intel Core 2 Duo processor running at 2.13GHz, 1GB RAM and a 160GB 7200 RPM hard disk. While the impact is significant, we conclude that the performance is acceptable. Our performance tests are described below.

5.5.1 Filter Rule Loading

The goal of the filter rule load test is to measure the time to load a large number of filter rules into the kernel. The loadRules program reads a filter rule file and calls the rule management module in the kernel to load the filter rules. Its function includes looking up both the source file and target file, translating the files to (inode, partition) pairs, checking to make sure the predicates are legal and adding this rule to the rule database. Files are identified by an (inode #, partition#) pair, where partition identifies the partition on which the file resides and inode identifies the inode number. Rules in the filter configuration file specify a file name, and as a rule is loaded the name is converted to an (inode, partition) pair.

We did two sets of tests. For each test, every file in the /usr/lib, /sbin, /lib directories were directed to a corresponding file in another directory. Each filter rule had 3 predicates. In test #1, each file has only 1 filter rule specified. While in test #2, each file has 10 filter rules associated with it.

Table 2 Test Results of Loading Filter Rule

File Count	Filter Rule Count	Source	Time(second)	Time per Filter Rule(second)
18375	18375	/usr/lib	21.5670794	0.0012
18375	183750	/usr/lib	146.809765	0.0008
7879	7879	/sbin, /lib	8.6629696	0.0011
7879	78790	/sbin, /lib	69.1612018	0.0009

The result is listed in table 2. The second column is the file count (the number of files in this test). The third column is the filter count. The source column indicates where the source files were from. The first row is the result of test #1. The second row shows the result of test #2. Loading multiple rules for the same file saves the time of allocating new node and inserting new node into the filter rule database. Thus, the average time needed loading one rule in the test #2 is less than in test #1. The average result is listed in the third row. There are 288794 filter rules in this test. The total time used is 246.2 seconds. The average time to load a filter rule in this test was 0.00085 seconds.

5.5.2 Rule Set in Memory Test

Filter rules reside in the kernel after they are loaded. Hence, they occupy some amount of system memory. The goal of this test is to quantify the impact on system performance. We loaded total 88977 filter rules on 18066 files into the kernel, each with 3 predicates. We then compiled the Linux 2.6.37.1 kernel. None of the files needed by the compilation was redirected.

The results are listed table 3. The first column lists the time used to compile the kernel without loading any rules. The second column shows the result with filter rules loaded. We ran the test 4 times. Each test was performed immediately after system reboot. The last line lists the average results. The results show that the overhead caused by 88,977 rules in the memory is *0.26%*.

Table 3 Overhead of Rule Set in Kernel

Test	Time(second) No Rule Loaded	Time(second) Filter Rule Loaded
Average	2045.9713	2051.4333

5.5.3 Path Name Lookup

We implemented redirection in the path name translation portion of the kernel. When a process requests access to a certain file, it sends the path name of the file to the kernel via a system call. The kernel performs a lookup to get the inode object of the file. After the kernel gets the inode object of the source file, FSF checks to see if redirection of the file is indicated.

This test measured the overhead the redirection causes in path name lookup. We issued a user level command to test whether a file exists in order to force the path name lookup. We compare the lookup time consumed with ordinary files, symbolic links (softlinks) and redirections. The lookup of an ordinary file requires one path name lookup. The lookup of a softlink involves one path name lookup of the original file and an extra lookup of the target file. Redirection demands one path name lookup and extra operations for redirection.

Tests were performed on 3108 files. Each file had 13 path components and 1 filename. In the test on soft links, 10 path components were soft links. In the test for redirection, 10 path components were redirected. Each filter rule had 1 predicate. The result is an average of 5 runs. Before the FSF redirection, we need to load the filter rules, which would cause lookup of the files, inodes and dentries. Before each run, we clear caches, inodes and dentries using Linux *sysctl*. The results of the test are given in table 4. The FSF kernel without redirection shows no significant difference from the plain kernel without redirection.

Table 4 Path Lookup On 3108 files

	Time (second)	Time per File (μs)
SUSE, ordinary file	0.05329	17.1
SUSE, softlink	0.06093	19.6
FSF, ordinary file	0.05389	17.3
FSF, softlink	0.06849	22
FSF kernel, redirection	0.07366	23.7

5.5.4 Reading and Writing Tests

Reading and writing tests are performed on 1147 files. Each run includes 91226 times of reading or writing of 100 characters. Two scenarios are involved: FSF kernel without any filter rule loaded, and FSF kernel with filter rules on 1147 files loaded. Table 5 lists the average total time of reading and writing. The numbers are average values of 1o runs. Before each run, the dentry cache and inode cache are cleared using *sysctl* command.

Table 5 Reading and Writing

	Time for reading(second)	Time for writing(second)
FSF, no filter rules	0.597336	0.145498
FSF, filter rules loaded	0.629063	0.146934

Chapter 6

Redirection

We discuss one of the filter function of FSF, redirection in this chapter. We explore capabilities of redirection by itself in both access control and administration aspects. We explain the rationale behind redirection, redirection on different file types and possible applications on security and administration.

Redirection [139] is part of file system firewall model. It can be used not only in file system security, but also file system administration. Many file systems provide the ability to redirect requests for a particular file to some other point in the file system hierarchy. This is commonly accomplished through creation of a special file that contains the information necessary to redirect the request to the appropriate location. FSF redirection is a VFS modification that supports transparent conditional redirection between two nodes in the file system hierarchy. The redirection in FSF does not require an on-disk link file. Whether requests are redirected is determined by a set of filter rules, similar to a firewall filter-rule set, that are loaded into the kernel. The redirection could happen without the awareness of users. On the other hand, the conditions on which to redirect can be based on virtually any element of the system state. With different conditions, the same pathname can point to different files. Transparent conditional redirection creates custom views of the file hierarchy. Redirection provides flexible solutions to many practical problems, and can be used as a component in the implementation of an application. Several example applications are given including redirecting UNIX domain sockets, access control, intrusion detection systems and honeypots, and manipulating file hierarchy.

6.1 Introduction

For many years, file systems have provided users with the ability to redirect an access request against one file to another point in the directory hierarchy. This is accomplished through creation of a special file (UNIX symbolic links, Microsoft Windows shortcuts, junctions or symbolic links, and MAC OS aliases). Requests against this file are then redirected to another file, the target of the redirection. FSF allows conditional redirection of file access requests without modification of on-disk data structures. Redirection is implemented within the Linux Virtual File System (VFS) [84, 85, 86] and is initiated by the kernel when a set of redirection rules are loaded, similar to the loading of network firewall rules.

Leaving the on-disk structures unmodified provides a degree of transparency not achieved by existing approaches. The redirection target is not apparent from a static view of the file system directory hierarchy or its contents. For example, suppose a user wishes to redirect requests against the file named source.txt to the file named target.txt. In current file systems, this would be accomplished by creating a file, e.g. a UNIX symbolic link source.txt that indicates the target target.txt of the redirection. A listing of file source.txt reveals that it is being redirected. In FSF redirection, listing source.txt does not reveal that requests are being redirected. It is sometimes possible to dynamically determine the fact that re-quests are being diverted. However, in this case, finding the target of the redirection typically requires a scan over a significant portion of the file system. On the other hand, redirections can be kept track of, so the related information is available when needed.

The transparency of redirection helps prevent exposing implementation details of certain tasks. For example, suppose the /var partition is almost full and there is huge free space on another partition, with FSF di-rectory /var can be redirected to the other partition without showing ordinary users the real location of the directory. With transparent redirection, certain files and directories can be hidden from untrusted users. This can serve as a form of access control to prevent leaking of information. For example, critical directories can be

hidden to a guest account or a demo account by redirecting the access requests to an empty directory or staged directories. In this case, the untrusted accounts cannot even list the original directories, while a lot of directories and files might be set as world readable to all users in the organization for convenience.

Another feature of the proposed approach is redirection with conditions. It allows the decision to redirect a request to be based on virtually any element of the system state, e.g., user ID (UID), effective user ID (EUID), requesting process, time of day, DTE domain, the size or the owner of the target file etc. That is, the redirection can be set to only happen on certain conditions. The condition could be that accessing time is between 9AM and 5PM. That the UID of the process is neither root nor Alice is a possible condition. That the owner of the target is Bob is another possible one. Being trans-parent and being conditional are the features tradition-al link files do not have. Suppose the redirection conditions are specified on the requesting process, then the specified process would access the target file instead of the source file. Other processes would still access the original source file. On different conditions FSF redirection creates different views of the file system. This can be used in file sharing and access control. Different redirection rules associated with different attributes of the system can be used to solve different kinds of problems.

FSF redirection can be used by system administrators to solve certain problems or to serve as a component in applications by creating customized views of the file system hierarchy under predefined conditions. Normal users may use redirection in a limited way. They can only set up redirection in their home directory when they are the owner of both source and target files in the filter rules. FSF redirection supports redirection on normal files, directories, soft links, named pipes and sockets.

6.2 Related Work

Dynamic or variant symbolic links [87, 88] and context dependent symbolic links [89, 90] are the closest work comparable with FSF redirection. These types of symbolic links point

to variant target files or directories. A variable or variant symbolic link is a symbolic link that has a variable name in the name of the target [88]. The variable is part of the target name. By specifying the value of the variable, the symbolic link is redirecting to appropriate target file or directory. For context de-pendent symbolic links, the variable is the context. Operating systems that support variant symbolic links include NetBSD, DragonFly BSD and Domain/OS. HP/Tru64 uses the cluster member number as the context in the context dependent symbolic link [88]. Com-pared with variant symbolic links and context dependent symbolic links, FSF redirection does not need to create special link files to redirect. The conditions to redirect in FSF redirection are defined as one or more predicates in the redirection rules, which is much more powerful and flexible. Another difference is that besides normal files and directories, FSF redirection supports redirection on symbolic links, named pipes and sockets as well.

Redirecting File System (RedirFS) [49] looks very similar to our work, but they are inherently different. The RedirFS is a framework without any functions itself. It resides between the VFS layer and the different file systems. RedirFS is designed to be used by Linux kernel modules (LSM), called filters in RedirFS. Filters can modify VFS file system calls through this framework to add functions to manipulate files and their metadata. Users can develop a filter to perform certain operations, such as compression and encryption, and then use RedirFS to interact with kernel. Both RedirFS and FSD redirection have the keywords of redirect and filter. The implementations are both on Linux kernel 2.6. However, RedirFS is a frame work for additional functionality, while FSF redirection redirects access requests of one file to another file.

6.3 FSF Redirection

FSF redirection provides transparent conditional redirection from one point of the file system hierarchy to another point without extra link files added to the file system. The same file can be accessed through multiple pathnames. Adding conditions to FSF redirection makes the redirection much more flexible. Under certain conditions, access

requests for one path-name could be redirected to different files as specified in the matched redirection rule.

6.4 Filter Rule

The redirection in file systems resembles the forward operation for network packages in network firewall. Through network filter rules, certain packages are for-warded to another node in the network. Through FSF redirection filter rules, certain access requests are redirected to another file for file systems. A redirection filter rule has three parts: the source file, the target file and a set of predicates on system attributes. The source file is the original file to be accessed. The target file is the file being redirected to. The set of predicates serve as the conditions of the redirection. The access requests against the source file will be redirected to the target file only if the set of predicates all are evaluated to be true. A filter rule is defined as follows:

filter rule: (source file, target file, filters: {p1^p2^p3 ...^ pn}).

The p1, p2,…, pn are the predicates on system attributes. The relations between the predicates are logic and. System attributes could be about the binary process, the target of the access request and the system itself. Example attributes are user ID(UID), group ID(GID), effective user ID(EUID), effective group ID(EGID), name of binary process, owner ID of the target file(OUID), size of the target file(TSIZE), host name, time of day, etc. Example predicate could be (UID != root) and (time-of-day > 9am) and (time-of-day < 5pm).

6.5 Redirection for Supported File Types

MaskFS works across partitions and file systems. It supports transparent redirection of normal files, directories, symbolic links, named pipes and sockets. It does not support redirection of block device or character device.

If a normal file is redirected, the access requests of the file will be against the target file specified in the matched filter rule.

File system hierarchy can be visualized as an upside down tree. Redirection of a directory file is redirecting the sub-tree of the source directory to the sub-tree of target directory. From the view of the tree, the sub-tree of source directory looks like being substituted by the target sub-tree.

Figure 4 below depicts the redirection from dir1/dir21 to dira/dirba. Tree (A) is the original directory tree of dir1. It is what the directory looks like without any redirection. Tree (B) is the directory tree of dira. Tree (C) shows what dir1 looks like with redirection, where dir21 sub-tree is substituted by dirba sub-tree from dira directory tree.

A symbolic link file is handled similar to regular file in redirection. The link file stops linking to the original target file. Access requests to the link file is redirected to the target file specified in the matching filter rule.

Named pipes, or FIFOs (first in, first out), are used as one type of inter-process communication. Named pipes have on disk file names and allow two separate processes to communicate with each other by the name of the pipe. Processes can open them by opening the pathnames to read from the pipes or write to them. MaskFS supports the redirection of named pipes. A named pipe can be redirected to another named pipe. Then the data sent to source pipe will be redirected to the target pipe. A process would read from or write to the target pipe specified by the matched filter rule other than the pipe with the name it requested. This changed the destination of the data sent to the pipe.

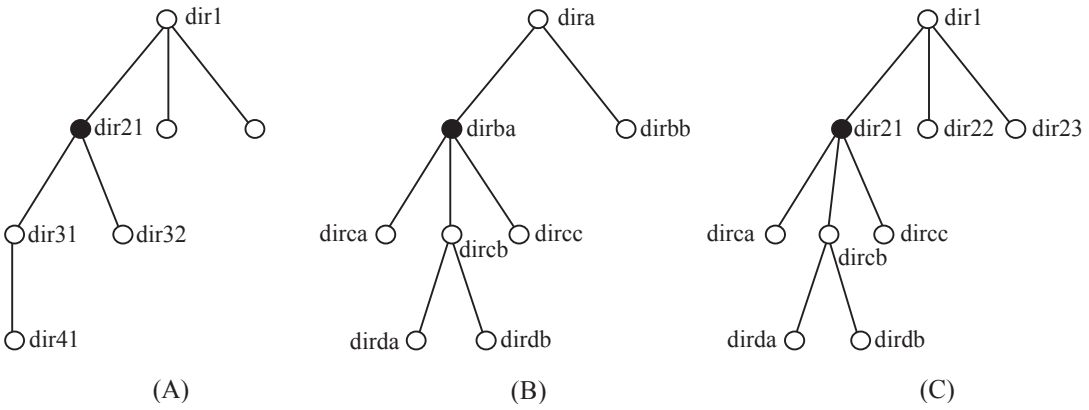

Figure 4: Redirection of a directory.

The sockets we discuss are UNIX domain sockets, that is, inter-process sockets. Sockets work in client-server mode and they are duplex-capable. They have pathnames on disk as well. In MaskFS when a socket is redirected to another socket, the client process associated with the source socket would communicate with a different process through the target socket. Since sockets are duplex-capable, it should be dealt with extra care to apply redirection to sockets.

Character and block device files are interfaces for device drivers with which the applications can communicate with hardware devices. Even though device files can be opened in the same way as normal files, further operation on the opened files are specific to the different hardware devices. Redirections of device files are not meaningful on most cases, so we do not support redirection of device files.

6.6 Applications

Redirection is a fundamental and very convenient tool. It manipulates the file system namespace and provides virtual views of the file system hierarchy for user ac-counts or processes. We will discuss several applications from the aspects of both an administrator and an ordinary user. To apply redirection, it takes two steps. One is to set up the filter rules; and the other is to load the rules into kernel by calling application program load-rule.

6.6.1 Used by Administrators

6.6.1.1. Redirecting Inter-process Communication (IPC) Sockets

IPC sockets [91], or UNIX domain sockets, are used to exchange data between processes on the same host. When redirection is applied to a socket, it changes the destination of the specified socket. One experiment we did is to manage printing jobs between two printers by redirecting a printer's socket to the other one's socket. These two printers are installed on the same machine. Printer1 uses socket cups1.sock to get the print data send by users. Printer2 uses socket cups2.sock.

Redirect: /var/run/cups/cups1.sock
/var/run/cups/cups2.sock
filter 3: gid = csgrad hour > 10 hour < 15

This filter rule defines that during peak period 10am – 3pm, the print jobs for Printer1 will be redirected to Printer2 for the group of 'csgrad'. This is a much simple case. But the flexibility provided by filters makes it easy to deal with much complex and practical problems. For example, we can write a wrapper script for the print command such that when the files to print are more than certain size, the printing jobs will be redirected to a printer that only prints big files.

6.6.1.2. Access Control on Extra Conditions

On current Unix/Linux, access control systems include traditional UNIX discretionary access control, access control list, Capabilities and security extensions such as SELinux [32] which supports Type Enforcement [92] and Role Based Access Control (RBAC) [93]. On most of the systems, the security policies are defined on the subjects, either account credentials or process domains. The advantage with redirections is security policies can be specified on more elements besides subjects in the security policies, such as the time of day, size of the target resource, last modified time of the target, etc. With redirection, denied access requests will be redirected to an empty directory or file.

The following filter rules are used to set up security policies such that the critical data file /billing/account/data1 can only be accessed during workdays in rule1, by the program /usr/sbin/techServ in rule2, and when the size of the file data1 is smaller than 100MB in rule3.

rule 1:
Redirect: /billing/account/data1
/empty
filter 5: day != Monday day ! = Tuesday day ! = Wednesday day ! = Thursday day != Friday

rule 2:
Redirect: /billing/account/data1
/empty
filter 1: program !=/usr/sbin/techServ

rule 3:
Redirect: /billing/account/data1
/empty
filter 1: size > 100MB

These access control policies could be implemented in certain access control systems. However, redirection provides straight-forward, light-weight and flexible al-ternative solutions.

6.6.1.3. Host Intrusion Detection System / Honey-pot

Redirection can be used to implement host intrusion detection systems [94] or honeypots [95]. Both Intrusion detection systems and honey pots need to monitor the running systems especially untrusted processes. With redirection, access requests for system resources made by these untrusted processes can be redirected to staged resources. The original data was

untouched by the untrusted processes. This makes it possible to lure and keep the attackers, and at the same time, keep the intrusion detection system or the honey pot available and running.

Redirect: /etc/passwd
/staged/etc/passwd
filter 1: 1=1

This filter rule specifies that once this filter rule is loaded, all access requests for file /etc/passwd will go to the staged file /staged/etc/passwd .

6.6.2 Used by Ordinary Users

Ordinary users may use redirection in a limited way. We require that the user that specifies the filter rules is the owner of both the source file and the target file. These two files have to be in the user's home directory.

6.6.2.1. Create Custom Views of the File Hierarchy

Dropbox is a cloud service that provides remote storage so we can access our data from different places using different devices.
Most of the time, the files in the Dropbox are copies of files on original file system on disk. They are copied from different directories scattered in the file system, or even from different devices. We need to keep track of where the files are originally from in the local file hierarchy. At the same time, among the copies in Dropbox and on local disk, we need to decide which one is the most recently modified copy.
Redirection can be used to access the file in Dropbox through the local file hierarchy. This answers both of the questions: where the file is from and which version is the newest. Suppose /home/Alice/work/projectA is the directory that has a copy in the Dropbox.

Redirect: /home/Alice/work/projectA
/home/Alice/Dropbox/projectA

filter1: uid = Alice

The above filter rule sets up the transparent redirection from /home/Alice/work/projectA to /home/Alice/Dropbox. After loading this filter rule, Alice can work in the exactly same local directory. There are no extra links and no modifications on the projectA directory. However, she actually accesses the projectA directory in the Dropbox, and she always accesses the newest version of the files.

Chapter 7

Comparisons of Access Control Systems

In this Chapter, we will discuss the theoretical comparisons of access control models. The importance of access control models on information systems has been recognized decades ago. Since then many access control models have been designed and implemented, and many access control systems have been deployed on various systems. The research community of information security has been interested in the comparisons of access control models. The comparisons of these access control models serve as references for system designers and end-users when they need to choose access control models to meet their protection requirements.

Two methods are presented in this chapter in terms of comparisons of access control models. One is to compare the express power of different access control models through simulation. This is the dominant method of comparing access control models in research community. The other is a straight-forward comparison of access control properties based on defined quality criteria.

Before we further our discussion of comparisons, first we need to make a clarification between access control models and access control systems. An access control model is more on the theoretical design side, while an access control system is more on the implementation side. An access control system includes specifications of access control policies, chosen access control model(s), implementation of the access control model(s) and deployment of the implementation on the information system [122]. One access control model may have multiple different implementations with different data structures, end-user

interfaces, efficiency and performance on different access control systems. In this chapter, we will discuss the comparison of access control models, rather than access control systems.

7.1 Theory on Compare Expressive Power of Access Control Models

Comparing the expressive power of access control models is a fundamental problem in computer security. The comparison is usually achieved through simulation. That is, if an access control model ***A*** can be simulated by another access control model ***B***, then access control model ***B*** is said to be at least as expressive as access control model ***A***.

The previous research on comparing expressive powers of access control models [114, 115, 116, 117, 118, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131] can mainly be grouped into three categories based on comparing approaches. The first is to enforce the access control policies of another access control model [114, 116,126]. The second is to implement one access control model by another through state mappings and/or state-transition mappings [120,121,124,125,127,128]. The third is to add queries of certain rights at certain states on top of the second category [123,129,130].

Most of these previous simulations use the implementation method, that is, to implement the access control policies of another model or to simulate the state-transitions of another access control model. While others put an emphasis on formal definitions of simulations and comparisons of expressive powers of access control models that will preserve security properties [121,124,127,128,129,130] during the process of simulation. The security properties include availability, mutual exclusion and bounded safety [130]. We will refer to these two methods as implementation method and formal state-transition method respectively in this chapter. First we will visit the formal state-transition method.

7.1.1 State-transition Simulation

In an access control model, a protection state is the collection of the values of all the configurations and entities in the system that involved in access control. In the simulation of access control models, actions or operations that trigger one protection state to transit to the next state are recognized as relevant actions and operations. The following two simulations both preserve the security properties.

Ammann, Sandhu and Lipton [121] use directed graphs to represent the access control scheme. Each access control scheme consists of a set of states and state-transition rules. An access control model is a set of access control schemes. They define a scheme ***A*** simulates scheme ***B*** if and only if both of the following are true.

1. For every state ***a*** reachable by scheme ***A***, there exists some correspondent state ***b*** reachable by scheme ***B***.
2. For every state ***b*** reachable by scheme ***B***, either the state ***a*** that corresponds to ***b*** is reachable by scheme ***A***, or there exists some successor state ***b**** of ***b*** such that the state ***a**** that corresponds to ***b**** is reachable by scheme ***A***.

Comparisons of expressive power are defined below:

Model ***Y*** is less expressive than model ***X*** if and only if there exists at least one scheme ***A*** in model ***X*** that cannot be simulated by any scheme ***B*** in model ***Y***.

Model ***Y*** is as expressive as model ***X*** if and only if for every scheme ***A*** in model ***X***, there exists a scheme ***B*** in model ***Y*** such that scheme ***B*** can simulate scheme ***A***.

Model ***X*** is equivalent to model ***Y*** if and only of model ***X*** is as expressive as model ***Y*** and model ***Y*** is as expressive as model ***X***.

Tripunitara and Li [129,130] define an access control scheme to be the state-transition system <S, Q, r, T>, in which S is a set of states, Q is a set of access control queries, r is defined as relation S × Q → {true, false}, and the T is a set of the state-transition rules.

They define reduction as:

Given two access scheme A= < S_A, Q_A, r_A, T_A> and B < S_B, Q_B, r_B, T_B >, for every a ∈ S_A and t_A∈ T_A, <b, t_B > = ***f*** (<a, t_A >), the mapping ***f*** is defined as a reduction if it has the following two properties:

1. For every state a_1 that is reachable from ***a***, and every query q_A in scheme ***A***, there exists a reachable equivalent state b_1 that is reachable from ***b***, that gives that same answer for query f(q_A).

2. For every state b_1 that is reachable from ***b*** and every query q_A in scheme ***A***, there exists a reachable equivalent state a_1 that is reachable from ***a***, that gives the same answer for query f(q_A).

The method of comparing the expressive power of access control models is defined below: Given two access control models ***A*** and ***B***, ***B*** is at least as expressive as ***A*** if for every scheme in ***A*** there exists a reduction from it to a scheme in ***B***. In addition, if for every scheme in ***B***, there exists a reduction from it to a scheme in ***A***, then we say that ***A*** and ***B*** are equivalent in expressive power.

7.1.2 Complexity of Formal Simulations of FSF Model

We try to apply the state-transition simulation theory to FSF model, but it is hard to find an access control model that has matching state with FSF. In FSF multiple attributes are involved in the decisions on whether to grant or deny access requests. These attributes can be subject-related, object-related or system-related.

The state-transitions in most access control models happen on actions or commands from the subjects. However, in FSF, the state-transition is more complex. Some changes may or may not be triggered by the same action. For example, conditions on the size of object could be changed through modification command. However, not every modification command changes the answer of the related query. For instance, if a rule specifies that a user cannot write to a file when it is larger than 100M. The query that whether the user is

allowed to write to the file answers true before the file grows to 100M. But after that it will answer false. There are attributes that change on their own, such as time. A query returned false the last second might return true this second even though there are no actions or commands involved. For these two kinds of state-transitions in FSF, there are no matching state-transitions in most of the access control models.

Because no matching state-transitions exist for certain transitions in FSF, we cannot apply the formal state-transition simulation theory to FSF and another access control model. If we want to apply the formal simulation theory to construct a reduction from FSF to another access control model, the property #1 cannot be satisfied. If we want to construct a reduction from another model to FSF, then the property #2 cannot be satisfied.

7.2 Simulation by Implementation

Some access control models have a structural definition on how to specify access control policies. For instance, DTE defines domains and types first, then the access control policies are expressed on different domains and types. RBAC defines roles first, then specifies access control policies by associating permissions with roles, and associating users with roles. While other access control models define the access control policies directly. Such as ACL, Capabilities and FSF. FSF includes multiple decision-making elements than most access control models. Most access control models base the decisions of allowing or denying access requests on one of the user identities. DTE bases the decisions on subject domains. RBAC bases the decisions on roles. While in FSF, the decisions are on multiple elements or attributes. The attributes can be user identity, domain, role and or any subset of attributes. Theoretically FSF is able to accommodate many other access control models.

Below we will present how to simulate TE with FSF using the approach of implementation.

7.2.1 Type Enforcement (TE)

Type Enforcement (TE) groups subjects into domains and objects into types. It defines the allowed operations a domain has over types, and other domains. Same with FSF, it is a rule-based policy model, that is, TE and FSF both specify in their policies what access requests should be allowed and what access requests should be denied. In contrast the other type is property-based access control policy, which specifies the security properties to enforce, such as in Bell–LaPadula model.

7.2.1.1 Original Type Enforcement

The original type enforcement policies are specified in two tables. Domain definition table (DDT) [3] defines the allowed accesses a domain may have over a type, such as read, write and execute. Another table domain transition table (DTT) [3] specifies whether a subject in a domain can transit to another domain, which is accomplished by allowed process execution. The transition table is defined as true or false table. If a process in one domain is allowed to transit to another domain, then the corresponding item in the table is labeled as “true”. Otherwise it is “false”.

7.2.1.2 TE in SELinux

SELinux uses policy language instead of DDT and DTT tables to define access control policies. TE in SELinux is refined from its original definition. The policy rules of domains over types are the same as the original TE. TE in SELinux defines more options during domain transitions. Three criteria have to been satisfied in a domain transition in SELinux TE.

1. The process in origin domain must have execute right on the application program
2. The application program is an entry point for the target domain
3. The origin domain must be allowed to transition to the target domain

Three possible rights are defined on domain transition, NULL, auto, and exec.

Type 1: NULL. When a process in domain A executes one of the entry point programs of domain B, and the access right domain A has over domain B is NULL, the child process will continue to run in domain A. If the child process is not an entry point program, the child process remains in the parent's domain. Most of the time in SELinux, the child process remains in the domain of parent process.

Type 2: auto. When a process in domain A executes one of the entry point programs of domain B, and the access right domain A has over domain B is auto, the process will continue to run in domain B.

Type 3: exec. When a process in domain A executes the entry point program of domain B, the access right domain A has over domain B is exec, and the process explicitly requests to run in domain B, the process will run in domain B. This type of transitions happen very rarely.

The domain of a process is decided by three element: the program itself, the domain of the parent process and the domain transition rules. The same program can be executed and run in different domains.

7.2.2 FSF Implementation of TE

A protection state of TE is defined as <S, O, D, T, R, dtr, ddr>.
S is a set of subjects. More specifically, S is a set of processes. Each subject is associated with a domain type.
O is a set of objects.
D is a set of defined domain types.
T is a set of object types.
R is a set of rights.
dtr is a mapping on $(D \times T) \rightarrow R$ that defines rights domains have over types.

ddr is a mapping on (D×D)→{auto, NULL} that defines allowed domain transitions. For simplicity, we do not include the exec type of transition which will require extra operations on the involved process.

A protection state of FSF is defined as <S, O, R, A, f>.
S is a set of subjects, O is a set of objects, R is a set of rights, A is a set of all attributes, and f is a mapping on (O×A)→R that defines FSF filter rules.

This implementation is constructed on the same set of objects, subjects and rights. O in TE maps to O in FSF; R in TE maps to R in FSF; Domain in TE is mapped to domain attribute in FSF.

7.2.2.1 Access Control Related TE Commands

We identify ten TE commands and operations that will modify the protection states.
1. Create an object.
2. Remove an object.
3. Assign a type to an object.
4. Modify the type of an object.
5. Define a domain transition.
6. Remove a domain transition.
7. Define a domain (assign rights a domain has over a type).
8. Modify a domain (revoke rights a domain has over a type).
9. Execute a program.
10. Exit of a process.

7.2.2.2 FSF Implementation of TE Commands

Now we will implement each of the TE commands.
1. Create an object.

This operation adds a new object to the state. However, this operation does not change any permissions. So no corresponding extra operations other than the creation of the object are needed in FSF.

2. Remove an object.

On destroying an object $\boldsymbol{O}$ in TE, FSF needs to remove the filter rules on this object. It means to call the operation removeObjectFilterRules($\boldsymbol{O}$) in FSF. The operation removeObjectFilterRules() will remove all the filter rules defined on object $\boldsymbol{O}$.

3. Assign a type to an object.

In TE, assigning a type $\boldsymbol{T}$ to an object $\boldsymbol{O}$ means granting the rights domains have over this type $\boldsymbol{T}$ to this object. In FSF, for each domain $\boldsymbol{D_i}$ that has right set $\boldsymbol{R_i}$ over type $\boldsymbol{T}$, it will generate a filter rule with $\boldsymbol{O}$ as the object, $\boldsymbol{R_i}$ as the right set, and {domain = $\boldsymbol{D_i}$} as attribute predicate by calling genFilterRule ($\boldsymbol{O}$, $\boldsymbol{R_i}$, {domain = $\boldsymbol{D_i}$}).

4. Modify the type of an object.

In TE, to modify the type of an object from $\boldsymbol{T1_i}$ to $\boldsymbol{T2_i}$ means to grant the rights domains have over the type $\boldsymbol{T2_i}$ instead of $\boldsymbol{T1_i}$ to this object. In FSF, removeObjectFilterRules($\boldsymbol{O}$) operation is called to remove all filter rules defined on the same object $\boldsymbol{O}$. Then FSF needs to call genFilterRule ($\boldsymbol{O}$, $\boldsymbol{R_i}$, {domain = $\boldsymbol{D_i}$}) for each domain that have rights defined over $\boldsymbol{T2_i}$.

5. Define a domain transition.

TE defines a domain transition by specifying the entry-point program, parent domain and the child domain. Accordingly, FSF needs to save this transition rule by calling saveTransitionRule(program, (domain_p, domain_c)).

6. Remove a domain transition.

TE removes a domain transition by disallowing the domain transition from domain_p to domain_c and unset the entry-point program. FSF calls removeTranistioinRule(program, (domain_p, domain_c)) to remove this rule.

7. Execute a program.

In TE, executing the entry program of a domain may cause domain transition. TE needs to check whether current process has the right to enter the new domain by executing the entry-point program. Correspondingly, FSF needs to get the domain value of current process:

domain_p = getAttribute (domain, parent_process), then checks the program to execute to see if it is an entry point program with parent domain equals the domain_p. If this is the case, FSF returns the new domain and set the domain of the executing program domain_c by calling setAttribute (new_domain, child_process).

8. Exit of a process.

The exit of a process means a process is finished. However, nothing related to domains and types is modified. So in FSF, no extra operations are needed.

9. Assign rights set ***R*** a domain ***D*** has over a type ***T***.

For each of the object O_i with the type ***T***, FSF needs to execute genFilterRule (O_i, ***R***, {domain = ***D***}).

10. Revoke rights set ***R*** of a domain ***D*** has over a type ***T***.

In FSF, for each of the object with the type, execute, removeFilterRules(O_i, ***R***, {domain = ***D***}).

The above implementation shows that FSF can simulate TE. Including attributes in the FSF makes it a very flexible access control model. The attributes can include a domain, a role, or the owner of the object. TE is just one example of the access control models FSF can accommodate.

7.3 The Comparisons on Specified Quality Criteria

Another approach is to compare different access control models against specified quality criteria. To compare two or more specific access control models, we need to choose the criteria that are both relevant and meaningful. In this section, we will explain the criteria [1, 96, 122] we choose and the results of comparisons. We select four access control models: DAC, DTE, RBAC and our FSF. Table 6 summarizes the results of comparisons.

7.3.1 Comparing Criteria

7.3.1.1 C1: Administrative Policies

Administrative policies define who manages the access control policies and how. That is, they define who has the permissions to add, modify, and remove the access control policies in the access control model. The most well-known type is centralized administration. In centralized administration, the access control policies are managed by the administrator(s). With ownership, the owner of the resources has the right to grant or revoke the permissions of the resources.

In DAC, the access permissions are decided based on the identities of subjects. The owner of a resource has the capability to grant access to other subjects. So it has the type of ownership in terms of administrative policies. In DTE, administrators define domains and types and set up the DTE policies for domains, so it is a centralized. In FSF, administrators have the capabilities to specify filter rules, load them and remove them from the filter database. It is centralized policy administration.

In RBAC administrators define roles and groups and the appropriate rights for each role and group. The users are assigned to one or more roles and get the rights associated to the roles. So it supports centralized policy administration. On the other hand, the administration rights can be associated with owner of resources, so it can support ownership as well.

Model	DAC	DTE	RBAC	FSF
Administrative Policies	Ownership	Centralized	Centralized / Ownership	Centralized

7.3.1.2 C2: Separation of Duties

Separation of Duties is a fundamental principle in computer security. It means multiple people are required to complete a critical task or a security process, since multiple people are less likely to be compromised at the same time.

In RBAC, constraints can be used to achieve separation of duties. For example, two roles can be defined as mutually exclusive, so a user can only be in one of these mutually roles. The other three models do not support separation of duties directly.

Models	DAC	DTE	RBAC	FSF
Separate of duties	Not support	Not support	Support	Not support

7.3.1.3 C3: Definition of Least Privilege

Least privilege means that a subject should be given only the privileges that are needed to complete a task. Besides DAC, All the other three access control models can be set up to support least privileges.

7.3.1.4 C4: Delegation

Delegation means that a non-administrator user has the capability of granting permissions to other users. DAC supports this. DTE and FSF are MAC access control models. Only administrators can grant permissions to users. In RBAC, the rights of granting permissions can be associate with certain roles.

Models	DAC	DTE	RBAC	FSF
Delegation	support	Not support	Support	Not support

7.3.1.5 C5: Implementation Mechanisms

DAC can be implemented with access control lists, access control matrix, or capabilities. DTE defines domain types, resource types, and permissions domains have over types and other domains in policies files. RBAC's implementation mechanisms include roles, role hierarchy, constraints and administration of assignments of user-role and role-permission. FSF uses policy files with filter rules on conditions.

7.3.1.6 C6: Conditions

Conditions can be system-dependent conditions, content-dependent conditions, or history-dependent conditions. In DAC, permissions are defined based on user identity. In DTE,

permissions are defined based on the type of the process and type of the resource. RBAC associates permissions with roles. No conditions are involved in these three models.

Models	DAC	DTE	RBAC	FSF
Conditions	Not support	Not support	Not support	Support

7.3.1.7 Open (Negative) and Close (Positive) Authentication

Open authentication means that the access requests are granted by default. An access request is denied only when it has a matching rule in authentication stating otherwise. On the contrary, close authentication means that the access requests are denied by default. An access request is allowed only when it has a matching rule in authentication stating otherwise.

Models	DAC	DTE	RBAC	FSF
Open and Close Authentication	close	close	close	open

Table 6 Comparisons on Specified Criteria

	DAC	DTE	RBAC	FSF
C1:Administrative policies	Ownership	Centralized	Centralized/ Ownership	Centralized
C2:Separation of duty	Not support	Not support	Support	Not support
C3:Least privileges	Not support	Support	Support	Support
C4:Delegation	Support	Not support	Support	Not support
C5:Implementation mechanisms	ACLs, ACM, or Capabilities	Domain and type enforcement	hierarchy, constraints, user-role and role-permission assignments	Filter rules on attributes

C6: Conditions	Not support	Not support	Not support	Support
C7:Close (Positive) and Open(Negative) Authentication	Close	Close	Close	Open

7.4 Conclusions

In this chapter, we present two different approaches of comparisons of access control models. One is through simulation to compare the expressive powers. If an access control model can simulate another access control model, we say the former access control model is at least as expressive as the later one. We simulated TE model with FSF and showed that FSF is more expressive than TE, since there are state-transitions in FSF that cannot be simulated by TE. Theoretically, with attributes, FSF can accommodate many other access control models. The other approach is quality comparisons based on related quality criteria. It provides clear results in terms of different access control properties. We compared DAC, DTE, RBAC and our FSF on seven criteria.

Chapter 8

User Study of Access Control Systems

In this chapter, we explain the design, executing and the results of a user study [138] on three selected access control systems[1]. This chapter includes the work that is published in SIN13 (for copyright information see Appendix: Reuse License).

Access control system is one of the key components in the security settings of computer systems. It protects system resources from unauthorized access attempts. Many access control systems have been designed and made available to administrators. However, there are access control systems that are rarely used, such as access control list [97]. There are other access control systems users would like to turn off, such as SELinux. Access control systems only take affect when they are used. The work we show in this chapter is an effort to understand and improve the usability of access control systems. We conduct a user study to observe the process users use three access control systems: UNIX discretionary access control (DAC), SELinux, and file system firewall we designed and implemented. We present the results of the study, analyze the results and propose our recommendations that could improve user experience with access control systems.

8.1 Introduction

An access control system on operating systems defines the operations that active entities, such as processes or users, may perform over resources or other active entities. Over the years, many access control systems have been deployed on various operating systems.

[1] This work is published in the 6th International Conference on Security of Information and Networks (SIN2013).

These access control systems implement substantially different abstractions of the requests made by processes against system resources.

Access control is a last line of defense for protecting computer system resources from a compromised process. It is the traditional center of gravity of computer security [98]. However, some powerful access control systems are rarely used. For example, access control lists (ACLs) are not used very often [97]. Other access control systems, such as SELinux, may be turned off1 on many systems even though the users need the protection SELinux could provide. Why would administrators choose not to use an access control system? It may be challenging to understand the existing security policies. Setting up a new set of access control policies may require a comprehensive understanding of the access control system. The process of setting up security policies may easily lead to errors. Learning to use the access control system may be intimidating. Clearly, usability issues can have serious security consequences.

In this chapter, we explain the design and the conduct of a user study on three different access control systems on Linux. UNIX DAC system, SELinux system and file system firewall designed and implemented by us. The goal is to study how well the potential users can finish specified tasks in each access control systems, and to look for suggestions to improve the usability of access control systems. The subjects are undergraduate students and graduate students in computer science major. They have very limited experience on using access control systems. The results show that with some training and sufficient information, they are capable of finishing most basic tasks. They have difficulties on more complex tasks which requires more knowledge on both Linux systems and access control systems. Based on the results and our findings from the study, we propose recommendations that might improve the usability of access control systems.

8.2 Background and Related Work

A great deal of effort has been devoted to bridging the gap between security and usability in information systems over the last decade [99]. The Symposium on Usable Privacy and Security (SOUPS) has been held since 2005. Additional workshops on usable security have been held more recently. The International Organization for Standardization (ISO) 9241-11 standard definition of usability is "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use." One branch of this research is human computer interface and security (HCISec) [100]. Much previous work emphasizes the interfaces [101] of security systems, such as password systems [102, 103]. Another branch focuses on misconfigurations of access control policies [104, 105, 106].

There have been a number of other access control user studies. Felt et al. [107] study the permission warnings during application installation on android system. Smetters and Good [108] examine the real-world use of access control features in standard document sharing systems and propose a set of design guidelines on access control system on these systems. Motiee et al. [109] investigate the use of low-privileged user accounts and user account control on Windows. Cao and Iverson [110] have conducted a user study to compare intentional access control management system wizard and ACL editor on distributed file system. Mazurek et al. [111] give guidelines for designing access control system for home computers and other digital devices based on feedback from user interviews.

Cliffe et al. [112] have conducted a user study on Linux to compare FBAC-LSM with SELinux and AppArmor. This is a study on Linux system involving three access control systems, which is similar to our study. The tasks for participants are to confine two applications, with one of them containing a Trojan horse. They concluded that functionality-based schemes such as FBAC_LSM empower non-expert end users to confine their applications and protect themselves from a variety of prevalent security threats [112]. Comparison of access control systems was not a goal in our study. We chose

these three access control systems, because they can provide different angles to observe the participants. Our work was an effort to improve usability of access control systems.

8.3 Design of User Study

This user study had two goals. One was to examine how well the participants could use these three access control systems to accomplish selected common tasks. The other was to study how users perceive and use different access control systems, in order to identify how the usability of access control systems might be improved.

The selected access control systems are inherently different (as are all access control systems). Though they have common goals, these systems have different abstractions, settings, and features. The ability of users to complete standard tasks, the processes employed by users to complete the tasks, and the perceptions of participants using these three systems will generate much more meaningful information than a single system.

8.3.1 Participants

Usable security is centered on the people who perform security operations. Usability is always within a certain context. A system usable to experts might not be usable to ordinary end users. The participants in this study are senior undergraduate and graduate computer science majors who have a limited knowledge of only the UNIX DAC system. They have a high degree of computing literacy, but they are essentially all beginners in the special domain of access control systems.

8.3.2 Three Access Control Systems

In this section, we give an overview of the access control systems selected for the study.

UNIX DAC

The UNIX discretionary access control system has widespread use. In UNIX DAC, the access permissions of a file include read, write and execute. Permissions are specified for each of the file owner, the file group, and everyone else (world). The owner of the resource has the discretion to delegate the privileges other users have over the resource he owns.

User accounts have a distinct user identifier (UID). Groups contain a collection of users. Each group has a distinct group identifier (GID). A user is in one primary group and might be in several secondary groups.

Each running process has an effective UID and GID, which is typically the UID of the user that created the process, and the GID of the group in which the user was operating at the time the process was created. This effective UID and GID are used to determine the access allowed to a resource. A UID of zero (root) allows virtually unlimited access to all system resources.

SELinux

SELinux is a mandatory access control (MAC) security mechanism implemented in the Linux kernel starting with Linux 2.6 kernel. It provides flexible, fine-grained and powerful access control. It supports type enforcement (TE) model, role-based access control (RBAC) and multiple-level security (MLS). Our study focuses on TE. In TE [3], subjects (processes) are grouped into different domains and objects (files, directories, device files, etc.) are grouped into types. Access control policies specify permissions domains have to types and other domains. In SELinux, each subject and each object are associated with a security context in the following format:

user:role:type

The first item in the triple is SELinux user (which is different from the UNIX user). The last item in the context triple is the TE type. TE security policies are defined between

subject types and object types and between subject types. The SELinux user is associated with the types. Since SELinux is mandatory, only the administrator can setup and modify the access control policies. Ordinary users cannot setup their own access control policies. During runtime, SELinux will check the security context triples to decide whether an access request should be granted or denied. Most of the SELinux systems run in the "target" policy mode. That means only the targeted subjects or objects are protected. Other entities are labeled with unconfined types and are not protected by SELinux.

File System Firewall

File system firewall (FSF) system is a novel access control system we implemented in the Linux kernel, specifically, on openSUSE. No participant had used the FSF prior to the study. Access control policies are specified in a policy file with filter rules. Each filter rule is an access control restriction for one particular file. The filter rule defines on what condition an access request against the file or directory should be allowed or denied. Three permissions are defined: read, write and execute. FSF acts before traditional UNIX access control. By default, if no filter rules are specified on a file or directory, an access request for the file will go to the traditional UNIX DAC system. Three types of filter rules are defined: deny rule, only allow rule and redirection rule. The filter rules are defined against resources with file structures on disk so far, such as files, directories, links, pipes and sockets. The system we used in this user study is a prototype access control system.

In FSF, the administrator can load these access control policies into the kernel or remove them from the kernel over any files and directories on the system.

8.3.3 Design of the Tasks

From the aspect of users, these three access control systems have different key components. The UNIX DAC system includes the following components.

- the root account

- commands and GUI tools to modify the permissions
- user accounts and groups
- setUID or setGID programs
- sudo, chroot jails and other mechanisms

TE itself is an access control model fairly straightforward to understand. Users must understand the basics of SELinux before they can use SELinux, which includes at least the following.

- more refined permissions defined by SELinux besides read, write and execute
- related domain types and resource types on the systems
- related SELinux macros
- SELinux commands tools to check, set and modify SELinux settings, Boolean variables and policies
- SELinux policy modules and policy files structures

The basic components of FSF system are listed below.

- commands to load, check and remove the rules
- the form of a configuration file with predicates
- three types of filter rules: deny, only allow and redirection
- the relations between predicates and multiple filter rules on the same object the relations between predicates and multiple filter rules on the same object

8.3.4 Tasks Used in Study

Each of these access control systems implements a different abstraction and hence has a different set of features, settings, etc. Each has its strengths and weaknesses. The goal of this study was not a technical comparison of the systems, but to use these different systems to gain insight into access control system usability.

Since the systems are so different, it did not make sense to use the same set of tasks for all three systems. Each task set had two parts. Part 1 included the basic tasks for each system. We selected these basic tasks according to the key components of each system. Tasks in part 2 were selected so that the same basic function was performed using the unique features of each system. The common goal was to confine a process or a daemon. FSF identified the process or daemon to confine by its absolute path name. The tasks are listed in table 7, 8 and 9.

Table 7 UNIX DAC Tasks

Task		Task Description	Object
Part 1	1	Prevent other accounts from accessing a user's files	Basic permission setting
	2	Multiple accounts sharing directory and files	Selective accounts sharing
	3	Add directories to sharing in task #2	Modify current policy
	4	Protect a public shared directory	Sticky bit and umask
	5	Assign a ordinary user a privileged task	Confine root with sudo
Part 2	6	Examine current access control set up for a particular process	Understand access control scheme on a process set up by others
	7	Set up access control scheme for a daemon	Design access control scheme on a process for certain requirements

Table 8 SELinux Tasks

Task		Task description	Object
Part 1	1	Change permissions using SELinux Boolean variable	Modify Boolean variable
	2	Modify object SELinux contexts to different permissions	Modify SELinux contests
	3	Setup a simple SELinux policies	Create a minimal SELinux policy module
	4	Debug an existing SELinux policy	Audit2allow
Part 2	5	Read an existing policy setup on a process	Read and understand existing policy
	6	Set up policy to confine a daemon	Design and create SELinux policies

Table 9 File System Firewall Tasks

Task		Description	Object
Part 1	1	Directories and files can only be accessed during a period of time	Time attributes
	2	Prevent other accounts (including root) from accessing certain files	Confine root
	3	Redirect access request to another object	Redirection rule
Part 2	4	Read filter rules set for a process	Read current filter rules
	5	Create filter rules for a daemon	Create filter rules

8.4 Measurement

ISO 9241-11 states that there are three key components in usability measurements: effectiveness, efficiency and satisfaction. Kainda et al. [100] include three extra possible measurement metrics: accuracy, memorability, and knowledge/skill to evaluate the

usability of a system. In the evaluation of access control system, accuracy is measured as how accurate participants could complete their tasks. Memorability is about data that need to be memorized, such as passwords in authentication systems. There are no this type of data in the use of access control system, so this is not a metric for our study. The knowledge/skill is essential for administrators to make sound decisions on setting up access control policies, so we include this in our measurements.

The measurements for each single task are defined in Table 10. Measurement for the whole system is defined in Table 11. The task perception section in Table 10 is an effort to understand what kinds of tasks appear more difficult and what could be done to mitigate the situation. The ease of learnability in Table 11 is to measure the difficulty to learn to use the specific access control system.

Table 10 Measurements for a Task

Goal	Measurements	How to measure	Who
Effectiveness	Task complete	Yes/No	By result evaluator
	Solution accuracy	Percentile	By result evaluator
Efficiency	Time	Quantity	By participants
Task perception	Task difficulty level	Scale level	By participants
	Task knowledge requirements	Scale level	By participants

Table 11 Measurements for an Access Control System

Goal	Measurements	How to measure	Who

Satisfaction	Satisfaction on the commands of each access control system	Scale level	By participants
Learnability	Ease of learnability of each access control system	Scale level	By participants

8.5 Study Methodologies

We conducted our user study on two Linux distributions, specifically, Fedora core 18 and openSUSE 11.4. Traditional UNIX access control and SELinux were set up on Fedora system. While the file system firewall system was setup on openSUSE. Under Linux, the active components are the users and processes. Access control systems under Linux define which user or process can access specified system resources. In our study, the resources are files and directories only; we did not set up tasks on other resources such as signals or devices.

8.5.1 Participants

There were twelve (all male) participants in our study. We did two pilot studies on two more experienced participants. One was an undergraduate student who had worked in campus IT department. The other was a graduate student doing advanced study in access control. The pilot studies helped clear certain confusions and make modifications to the tasks, reading materials and environment settings. The remaining ten took part in the final study. Of the twelve people in total, three were college students in CS, eight were CS graduate students and one was a faculty. The questionnaires showed that 100% percent of participants had used UNIX DAC system, from seldom to daily. While 70% of the participants had never used SELinux, the other 30% rarely used SELinux. Since FSF is new, no one had used it before. The user study was done in a five-day series of sessions. Participants chose time slots that worked for them. It took 1.5-2 hours for most participants

to complete the study. One devoted participant worked on the study for about 3 hours. Two participants did not finish the entire study; they left after they finished tasks on UNIX DAC. All the other participants finished the study completely.

8.5.2 Procedure

Before the study, the participants were given reading materials and papers describing the tasks to be completed. We then gave them a short explanation of the user study and a review of each of the access control systems. Next, we explained the reading materials and answered the questions they had. Then they used computers to finish the tasks using handouts together with related reading materials. Participants chose the order of completion of the tasks. We used recordMyDesktop software to record the screen of each participant during the process. Participants were allowed to ask questions on the access control systems, tasks and reading materials, as long as the questions were not about how to finish the tasks. They were allowed to take breaks, or leave when they do not want to continue the study. The questionnaire was completed last.

8.5.3 Data Collection

We collected data in three ways. One was to save the task results on the computer. Before the user study, we set up the environment and cleared the related settings and files. After each participant finished the study, we saved the task results and answers into files from the computer they used. The second one was to save the video records. We used recordMyDesktop to record the screen during the time that participants were working on the computers. After each user study, we saved the video files. The questionnaire was the third way of collecting data from the participants.

8.6 Results

8.6.1 Results on Each Access Control System

Table 12 shows the results of UNIX access control tasks. The "Complete Success" column lists percentage of participants that finished all the tasks correctly. The "Accuracy" column shows the mean of accuracy scores from all participants.

Tasks of part 1 are more basic operations involving understanding of groups, permission bits and sudoers configurations. The accuracy of these operations is over 70%. Tasks in part 2 are more comprehensive tasks, which are to read and write access control settings to confine a process. The accuracy is below 50%.

Table 12 UNIX Access Control Tasks Results

Task		Success (out of 12)	Partial Success (out of 12)	Failure (out of 12)	Average Score (%)	Average Time (min)
Part 1	1	12 (100%)	0	0	100	2.4
	2	6 (50%)	6 (50%)	0	88	7
	3	7 (58%)	3 (25%)	2 (17%)	73	1.4
	4	7 (58%)	5 (42%)	0	79	5
	5	8 (67%)	4 (33%)	0	87	7.6
Part 2	6	5 (42%)	1 (8%)	6 (50%)	48	6.4
	7	3 (25%)	5 (42%)	4 (33%)	46	12.1

Table 13 SELinux Tasks Results

Task		Compete Success (out of 10)	Partial Success (out of 10)	Complete Fail (out of 10)	Average Score (%)	Average Time (min)
Part1	1	1(10%)	0	0	100	2.5
	2	3(30%)	7(70%)	0	74.5	12.3
	3	7(70%)	2(20%)	1(10%)	79	11.2
	4	8(80%)	0	2(20%)	80	19.7
Part2	5	5(50%)	0	5(50%)	50	11
	6	1(10%)	4(40%)	5(50%)	42.5	20

Table 13 gives the results of SELinux tasks. Similar to the UNIX DAC access control tasks, participants performed better on the part 1 tasks, which were single tasks, than on the more comprehensive tasks in part2. All participants spent more time on each of the SELinux tasks than UNIX DAC. Apparently reading a security policy takes almost half of the time required to set up a policy for both the UNIX DAC system and SELinux system.

Table 14 File System Firewall Tasks Results

Task		Compete Success (out of 10)	Partial Success (out of 10)	Complete Fail (out of 10)	Average Score (%)	Average Time (min)
Part1	1	10	0	0	100	6.4
	2	6	4	0	80	3.4
	3	9	1	0	96	4.6
Part2	4	10	0	0	100	5.4
	5	9	1	0	96	4.4

Table 14 gives the results on the FSF tasks. Different from the other two access control systems, there is no significant difference between the accuracies for different tasks, even the ones in part 2. The first task was the first use of this system for each participant. The average time for the first task is longer. Specifications of access control policies are similar among different tasks, so the scores are not as different as in the other two access control systems. Reading the policies (task 4) took more time than writing the policies (task 5).

8.7 Overall Explanation of the Results

Table 15 Average Scores of Three Access Control Systems (%)

Task	Part1					Part2	
	1	2	3	4	5	1	2
UNIX DAC	100	88	73	79	87	48	46
SELinux	100	74.5	79	80	N/A	50	42.5
FSF	100	80	96	N/A	N/A	100	96

Table 15 is a summary of Table 12, 13, and 14 on accuracy. Participants can accomplish part1 tasks (basic operations) with an accuracy higher than 70. The last two tasks for all three systems were reading an existing security setup for a process, and setting up security schemes for a process. To set up a confined environment in UNIX DAC, two popular approaches are using chroot jail, and manipulating UID, GID and permission bits. Due to time constraints, we did not put the chroot jail approach on the task list. The approach using UID, GID and permission bits does not create a complete, self-contained statement of the solution of the task. The settings are on related files and in the system. On the other hand, the complete security policy specification can be found in the policy files on a SELinux system. This explains why participants score more reading existing SELinux policies than UNIX DAC security setups.

The questionnaires showed that 100% percent of participants had used UNIX DAC system, from seldom to daily. While 70% of the participants had never used SELinux, the other 30% rarely used SELinux. We had expected the participants to achieve higher accuracy on UNIX DAC than SELinux. Participants did spend more time on SELinux tasks than UNIX DAC tasks. However, in Table 9, the accuracy differences between the two were not dramatic. One reason is that we provided participants with sufficient explanation and examples on related Boolean variables, SELinux types that should be set, and policy macros used in the SELinux policy files. On the other hand, we specified clearly in the task descriptions what permissions that the confined process should have over different resources.

On a production system, in order to setup SELinux policies for a process, the administrator at least needs to know:

1a. What resources should the current process have access to?

1b. What SELinux types do these resources belong to?

2a. What access permissions should the current process have over each of these resources?

2b. What are the appropriate permissions defined in SELinux? (SELinux provides fine-grain access control, one real world permission could be mapped to several SELinux permissions. SELinux has certain macros defined. Understanding these macros would make this part easier.)

3a. What relations does the current process have with other existing processes?

3b. What interfaces should be used or defined?

Among these questions, 1a, 2a and 3a are about understanding the system and the current process. 1b, 2b and 3b are about learning to use SELinux. For standard system daemons, experienced administrators would easily come up with answers to these questions.

However, for a new daemon or a new application, this would require testing and debugging related SELinux modules. The point is that we provided participants with sufficient information to answer 1a, 2a and 3a, so they only needed to focus on learning to use SELinux. Even through the scores of UNIX DAC and SELinux in the table are not significantly different, this does NOT imply that SELinux is as easy to use as UNIX DAC.

The FSF is an access control system that uses filter rules to specify access control policies over resources. A full discussion of features of the firewall system is beyond the scope of this paper. We will focus on the user experience. To specify a policy, users need to specify the target and the predicates on the current filter rule. Then they need to figure out the relations of this new policy with existing policies over the same resources. It is the same with all tasks. That is why even the last two tasks, reading and setting up policies for a specified process did not show much difference from scores of other tasks.

Table 16 Average Overall Rating from Participants

Feature and scale	**UNIX**	**SELinux**	**Firewall**
Mean rating for **easy to use** (1: strongly disagree; 2: disagree; 3: neutral; 4: agree; 5: strongly agree)	4	2.1	4.6
Mean rating for **fast to learn** (1: strongly disagree; 2: disagree; 3: neutral; 4: agree; 5: strongly agree)	3.9	2.9	4.4
Mean **tasks difficulty level** (1: very easy; 2: easy; 3: moderate; 4: difficult; 5: very difficult)	2	3	1.5
Mean **tasks knowledge/skill** (1: not much; 2: some; 3: moderate; 4: a lot; 5: a great amount)	2.4	3.2	2

Table 16 shows the ratings in the questionnaires from the participants. Participants rated the firewall system much easier to use and faster to learn than the other two access control systems. The difficulty level and the knowledge/skill level of FSF are both rated lowest. The simple logic on predicates and multiple filter rules could be one reason. The other reason might be that the filter rules are more readable to participants since they are closer to natural language. The concepts of SELinux, new commands, arguments of the commands, policy file structures and macros do show more knowledge/skill requirements. The knowledge/skill level is rated highest for SELinux.

8.8 Discussions

Lesson Learned from UNIX DAC

One interesting result was that UNIX DAC did not show much better results on reading and setting up access control policies for a process than SELinux. Participants actually got a higher score for reading policies of SELinux than UNIX DAC, even though participants rated the task difficulty and knowledge/skill level of UNIX DAC lower than SELinux. One reason might be that access control policies for confining a process or a daemon on SELinux and the FSF, are much more readable than on UNIX DAC. SELinux and the FSF use files to save access control policies. So the access allowed by a process is listed in the files. This makes creating a policy, reading the policy, and making changes to the policy more convenient. While on UNIX DAC, to confine a process without using the chroot jail, the setup will be scattered on the system. We observed that some participants look confused while working with UNIX DAC. They keep changing directories and listing files during the task to read a policy.

Lessons Learned from SELinux System

One SELinux task was to modify the current httpd SELinux policy to allow the denials of reading a file. The task could be done using a permissive domain. More precisely, one could set the daemon domain permissive, run the command which issues the reading

request to collect denial messages, then install all the extra allowing rules of httpd policy. This would only require installing the policy once. Another way is to keep the daemon domain enforcing the current policy, run the command, check for denial messages and setup additional SELinux policy for the denials. This step is then repeated, until the reading of the file is allowed. We guided participants to use this second method since this is more practical on a running system. For this particular task, the participants had to repeat the above process three times to generate three additional SELinux policy modules (for getattr, read and open permissions) to finally allow the reading. This is somewhat similar to debugging code. 20% participants went on with the third modules to finally get a successful reading. 20% gave up after the second module. 40% gave up after the first module. Most of them showed frustration after the second module or even the first module.

This task is relatively trivial. The frustration may be caused by their lack of experience. They did not know what permissions were required and how many were they. Experienced administrator would not have problems with this task.

However, suppose a new application with complex operations is to be installed, administrators would face the same problem when setting up SELinux policies? They will not know exactly what permissions the application needs. If each daemon or application could have a description of resources and appropriate permissions it requires during execution, the work to set up SELinux policies would definitely become easier. Especially, this would greatly encourage new SELinux users.Another task is permanent labeling a directory. It requires two SELinux commands. 20% Participants missed the second command for the last directory. This reminds us whenever possible, it is better to make the necessary steps to accomplish a task as few as possible in access control systems.

Another task was permanently labeling a directory. It required two SELinux commands. Twenty percent of participants missed the second command for the last directory. This

reminds us that whenever possible, it is better to minimize the number of steps necessary to accomplish a task within access control systems.

Lessons Learned from FSF System

On the FSF, most errors occurred because the participants were not clear on the relations of predicates in the definition, and the logic relations of multiple filter rules. For instance, some define filter rules with both predicates "uid = Bob" and "uid = Andy". Since a user account cannot be both Bob and Andy at the same time, this is a not a legitimate rule. These mistakes in defining filter rules call for additional filter rule checking and evaluations in the loadRules program.

Another reason for errors is the lack of motivation to test the security policy setup. This was common to all three systems. Some participants would not test the results if the testing required more effort, such as switching accounts. We assume part of the reason is that it is a study. Real access control users would need to test out their security setups. Yet, this does suggest we could develop certain auto-test software to mitigate the burden of administrators.

In the user study, we provided users with a file of example filter rules. We observed that all the participants opened this file while they edited their own filter rules. We were impressed by the accuracy of their filter rules. We also observed that most users read the same man-pages multiple times. Some users tried to look for related problems and solutions online. This suggests that examples and explanations would be of great help to users.

8.9 Summary and Suggestions

We conducted a study if 12 CS major students using three access control systems. All participants had used UNIX DAC more or less, 30% participants knew a little about

SELinux, but had rarely used it. No one had ever used FSF system before. We studied the process that participants used different access control systems and the results of tasks. They were able to use the FSF system to solve different tasks with an average score of 94.4. They learned to use UNIX DAC and SELinux commands to finish basic operations (part1 tasks) with average scores 85.4 and 83.4, respectively. They could read and set up UNIX DAC and SELinux policies for a specified process with a score of 47 and 46.3, respectively.

For UNIX DAC and SELinux, with about 30 minutes' training and reading materials, participants could accomplish most typical tasks, but they had difficulty finishing more complex tasks which required certain administrators' expertise. This administrators' expertise included knowing the SELinux settings of the current system, and knowing how each confined process was supposed to behave over system resources. Adding a module that evaluates the relations of predicates on the same filter rule or multiple filter rules on the same target would improve the usability of the firewall system.

Below are some general suggestions we have learned from this user study.

- Access control policies on a process or a daemon should be presented so that users could easily see the entire picture, what the process or daemon can do or cannot do. Scattered settings cause confusion.
- Users are not motivated to test the permissions settings, at least in this user study. Setting up a testing environment can be challenging and time-consuming. Users of access control systems would benefit greatly from a well-designed auto-test system.
- Working with SELinux policies is similar to programming in certain ways. Policy modules include macros and interfaces defined elsewhere. It would be helpful to be able to read the related macros and interfaces within a policy development environment.
- All participants checked the manual pages. Two participants went to online resources for examples. The example filter rules file we gave in file system firewall

helped a lot. Appropriate documentations with examples are needed for access control systems.

8.10 Limitations

The tasks in our study did not cover all the aspects of the three access control systems. For UNIX DAC, we did not include use of a chroot jail in our study. For SELinux, we only used the command-line tools. No GUI tools were involved. The user study was done on Fedora 18. So the results of SELinux should be evaluated in this context. Our participants were not administrators or experts, so the results and evaluations are from beginner users of access control systems.

Chapter 9

Conclusion

Current access control systems on operating systems are specified on one single entity, such as the account ID or DTE domain. This dissertation proposes an access control system file system firewall (FSF) that specifies the security policy over a programmable condition which is defined on various attributes of the subject, object and the system. The features of FSF decide its adoption and use is relatively easy, which is confirmed in the user study we conducted.

9.1 Contribution

We propose a new type of access control system on operating systems in this dissertation. The security policies can be defined on various and multiple attributes on the subject, object and the system of the access requests. We implemented the prototype of FSF on openSUSE. The performance test of the prototype shows the overhead is acceptable. The redirection in FSF system can serve not only a security feature, but also a tool that administrators can use to solve certain problems. This is discussed in Chapter 6. We did a theoretical comparison of FSF with the well-known TE in SELinux. We implemented TE with FSF in Chapter 7. Usability is one of the most important feature of access control systems. Only when access control systems are used, can they take effect and protest sensitive data. We showed the user study and results in Chapter 8. The ease of use is one of the most beneficial features firewall model has.

9.2 Future Work

One of our future work is to explore extra attributes that are meaningful in system and user applications. The prototype of FSF provides a platform to include more attributes.

Redirection, as a part of FSF, is very useful tools on its own. We want to find appropriate applications to expand its usage.

The integrity part is another part of future work. We discussed it in the early stage of FSF. The design of integrity is left to study and research.

References

[1] Matt Bishop. *Introduction to Computer Security*. Addison-Wesley, 2002.

[2] Andrew S. Tanenbaum. *Modern Operating Systems*. Pearson Prentice Hall, Pearson Education, Inc. 2008.

[3] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, S. A. Haghighat. *Practical Domain and Type Enforcement for UNIX*. 1995 IEEE Symposium on Security and Privacy, page: 66, Oakland CA, May 1995.

[4] Serge E. Hallyn and Phil Kearns. *Domain and Type Enforcement for Linux*. 4th Annual Linux Showcase & Conference, Atlanta, Octorber, 2000.

[5] Karen A. Oostendorp, Lee Badger, Christopher D. Vance, Wayne G. Morrison, Michael J. Petkac, David L. Sherman and Daniel F. Sterne. *Domain and Type Enforcement Firewalls*. The Annual Computer Security applications Conference (ACSAC) December, 1997.

[6] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Sherman and Karen A. Oostendorp. *Confining Root Programs with Domain and Type Enforcement (DTE)*. In proceedings of the Sixth USENIX UNIX Security Symposium, San Jose, California, July 1996.

[7] Lee Badger, Daniel F. Sterne,David L. Sherman,Kenneth M. Walker and Sheila A. Haghighat. *A Domain and*
Type Enforcement UNIX Prototype. In proceedings of the Fifth USENIX UNIX Security Symposium, Salt Lake
City, Utah, June 1995.

[8] Lee Badger, Daniel F. Sterne,David L. Sherman,Kenneth M. Walker and Sheila A. Haghighat. *Practical Domain and Type Enforcement for UNIX*. In Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA,
May 1995.

[9] Jan Kiszka and Bernardo Wagner. *Domain and Type Enforcement for Real-Time Operating systems*. Emerging
Technologies and Factory Automation, 2003. Proceedings. ETFA ’03. IEEE Conference, Volume: 2, pages: 439-446, Sept. 2003.

[10] Robert Grimm. *Domain and Type Enforcement for Legacy File systems*. Available at: *http://cs.nyu.edu/rgrimm/papers/dte-legacy-fs.pdf* (accessed May 23, 2014).

[11] Jonathon Tidswell and John Potter. *An Approach to Dynamic Domain and Type Enforcement*. In proceedings of the Second Australasian Conference on Information Security and Privacy, pages: 26-37, 1997.

[12] David F. Ferraiolo, Janet A. Cugini. D.Richard Kuhn. *Role-based Access Control (RBAC): Features and Motivations*. 11th Annual Computer Security Applications Proceedings 1995, pages 241-48, New Orleans, LA, December 11- 15 1995.

[13] David F. Ferraiolo and D.Richard Kuhn. *Role-Based Access Controls*. 15th National Computer Security conference (1992) Baltimore, pages 554-563, Oct 13-16, 1992.

[14] Ramaswamy Chandramouli. *A Framework for Multiple Authorrzation Types in a Healthcare Application System.* In proceedings of the 17th Annual Computer Security Application Conference (ACSAC2001), pages 137-148, IEEE, 2001.

[15] Michael Fox, John Giordano, Lori Stotler, Arun Thoms. *SELinux and Grsecurity: A Case Study Comparing Linux Security Kernel Enhancement.* Available at: http://www.cs.virginia.edu/~jcg8f/GrsecuritySELinuxCaseStudy.pdf (accessed May 23, 2014).

[16] William Enck, Sandra Rueda, Joshua Schiffman, Yogesh Srenivasan. *Protecting Users from"Themselves".* Proceedings of the 2007 ACM workshop on Computer security architecture, pages: 29-36, Fairfax, Virginia, USA, 2007.

[17] John H. Howard , Michael L. Kazar , Sherri G. Menees , David A. Nichols , M. Satyanarayanan , Robert N. Sidebotham , Michael J. West. *Scale and Performance in a Distributed File System.* ACM Transactions on Computer Systems (TOCS), v.6 n.1, pages 51-81, Feb. 1988.

[18] *Trusted Computer System Evaluation Criteria.* United States Department of Defense. DoD Standard 5200.28-STD. Available at: *http://csrc.nist.gov/publications/history/dod85.pdf* (accessed May 23, 2014).

[19] *CERT Advisory CA-2000-16 Microsoft.* Available at: *http://www.cert.org/historical/advisories/CA-2000-16.cfm* (accessed May 23, 2014).

[20] Martin Abadi, Cedric Fournet. *Access Control Based on Execution History*. In Proceedings of the Network and Distributed System Security Symp, 2003.

[21] Micheal M. Swift, Anne Hopkins. *Improving the Granularity of Access Control for Windows 2000.* ACM Transactions on Information and System Security (TISSEC) Volume 5, Issue 4, pages: 398-437, November 2002.

[22] Jeffery Choi Robinson, W. Scott Harrison, Nadine Hanebutte. *Implementing Middleware for Content Filtering and Information Flow Control.* Proceedings of the 2007 ACM workshop on Computer security architecture, pages: 47-53, Fairfax, Virginia, USA, 2007.

[23] Marie Rose Low, Bruce Christianson. *Fine Grained Object Protection in UNIX.* ACM SIGOPS Operating Systems Review, Volume 27, Issue 1, pages: 33-50, January 1993.

[24] Jon A. Solworth, Robert H. Sloan. *A Layered Design of Discretionary Access Controls with Decidable Safety Properties.* IEEE Symposium Security and Privacy, Oakland, 2004.

[25] Philip W.L.Fong. *Access Control by Tracking Shallow Execution History.* Security and Privacy, 2004 Proceedings. 2004 IEEE Symposium, pages: 43-55, May 2004.
[26] Trent Jaeger, Kevin Elphinstone, Jochen Liedtke. *Flexible Access Control Using IPC Redirection.* Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop, Rio Rico, AZ, USA, pages 191-196, March, 1999.
[27] *Summary about Posix.1e.* Available at: *http://wt.tuxomania.net/publications/posix.1e/* (accessed May 23, 2014).
[28] Chris Wright and Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. *Linux Security Module Framework.* In Proceedings of the Ottawa Linux Symposium, 2002.
[29] ChrisWright and Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. *Linux Security Modules: General Security Support for the Linux Kernel.* In proceedings of the 11th USENIX Security Symposium, 2002.
[30] Xiaolan Zhang, Antony Edwards, Trent Jaeger. *Using CQUAL for static Analysis of Authorization Hook.* In proceedings of the 11th USENIX Security Symposium, August 2002.
[31] Stephen Smalley, Timothy Fraser, and Chris Vance. *Linux Security Modules: General Security Hooks for Linux.* Available at: *http://www.hep.by/gnu/kernel/lsm/* (accessed May 23, 2014).
[32] Stephen Smalley, Chris Vance and Wayne Salamon. *Implementing SELinux as a Linux Security Module.* NAI Labs Report #01-043, December 2001.
[33] Stephen Smalley and Timothy Fraser. *A Security Policy Configuration for the Security-Enhanced Linux.* NAI Labs Technical Report, February 2001.
[34] Peter Loscocco and Stephen Smalley. *Integrating Flexible Support for Security Policies into the Linux Operating System.* In Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, June 2001.
[35] Peter Loscocco and Stephen Smalley. *Meeting Critical Security Objectives with Security-Enhanced Linux.* Proceedings of the 2001 Ottawa Linux Symposium, July 2001.
[36] Peter Loscocco and Stephen Smalley. *Integrating Flexible Support for Security Policies into the Linux Operating System.* NSA Technical Report, February 2001.
[37] Mick Bauer. *Paranoid Penguin: An Introduction to Novell AppArmor.* Linux Journal Volume 2006, Issue 148, page 13, August 2006.
[38] S. Hallyn and P.Kearns. *Domain and Type Enforcement for Linux.* In Proceedings of the 4th Annual Linux Showcase and Conference, October 2000.
[39] John McCormick. *Top 10 Linux, UNIX threats.* Avalible at: *http://www.techrepublic.com/article/sans-updates-its-list-of-the-top-10-linux-unix-threats/#.* (accessed May 23, 2014).
[40] Dean Turner. *Symantec Internet Security Threat Report Volume XIII.* Available at:

http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xiii_04-2008.en-us.pdf (accessed May 23, 2014).

[41] Dennis M. Ritchie. *On the Security of UNIX.* In Unix Programmer's Manual [Unix Programmer's Manual, 1979].

[42] John Shelby Heidemann. *Stackable Layers, an Architecture for File System Development.* Technical Report UCLA+CSD 910056, University of California, Los Angeles, CA, July 1991.

[43] Max Hailperin. *Operating System and Middleware.* Seiten 230257. Thomson Course Technology, Boston, 2007.

[44] Silberschatz, Galvin, Gagne. *Operating System Concepts.* Wiley, sixth edition.

[45] What is UNIX? Available at: *http://www.unix.org/what_is_unix.html*(accessed May 23, 2014).

[46] Christin Groba, Stephan Grob and Thomas Springer. *Context-Dependent Access Control for Contextual Information.* Availability, Reliability and Security, 2007. ARES 2007. The Second International Conference on2007.

[47] Christian Payne. *A Cryptographic Access Control Architecture Secure Against Privileged Attackers.* Conference on Computer and Communications Security, Proceedings of the 2007 ACM workshop on Computer security architecture, Fairfax, Virginia, pages: 70-76, 2007.

[48] *RedirFS and Dazuko Filter.* Available at: *http://osdir.com/ml/linux.dazuko.devel/2006-03/msg00000.html* (accessed May 23, 2014).

[49] Redirecting FileSystem. Available at: *http://www.redirfs.org/tiki-index.php* (accessed May 23, 2014).

[50] Niels Provos, Thorsten Holz. *Virtual Honeypots.* Addison-Wesley Professional, 1 edition, 2007

[51] John S. Heidemann. *File-System Development with Stackable Layers.* ACM Transactions on Computer Systems (TOCS), Volume 12, Issue 1, pages: 58-89, February 1994.

[52] Erez Zadok, Ion Badulescu, and Alex Shender. *Extending File System Using Stackable Templates.* In Proceedings of the USENIX Annual Technical Conference, Monterey, California, USA, page: 5, June 6-11, 1999.

[53] Erez Zadok. *Stackable File Systems as a Security Tool.* Technical Report CUCS-036-99, Computer Science Department, Columbia University, December 1999.

[54] Erez Zadok and Ion Badulescu. *A Stackable File System Interface for Linux.* In Proceedings of the 5th Annual Linux Expo, page 141=151, Raleigh, North Carolina, May 1999.

[55] J. Sipek, Y. Pericleous, and E. Zadok. *Kernel Support for Stackable File Systems.* In Proceeding of the 2007 Ottawa Linux Symposium, volume 2, pp. 223227, Ottawa, Canada, June 2007.

[56] Erez Zadok. *Writing Stackable Filesystems.* Linux Journal, *http://www.linuxjournal.com/article/6485* (accessed May 23, 2014)

[57] Erez Zadok. *FiST: A System for Stackable File-System Code Generation.* PH.d dissertation in the Graduate School of Arts and Sciences Columbia University, May, 2001

[58] Central Computing UNIX (CCU) System. Available at: *http://www.umanitoba.ca/computing/ist/systems/unix/index.html* (accessed May 23, 2014).

[59] Ravi S. Sandhu, Edward J. Conyne, Hal L. Feinstein and Charles E. Youman. *Role-based Access Control Models.* Computer, vol. 29, no. 2, pp. 38-47, Feb. 1996.

[60] Ravi Sandhu, David Ferraiolo and Richard Kuhn. *The NIST Model for Role-Based Access Control: Towards a Unified Standard.* Proceedings of the fifth ACM workshop on Role-based access control, Berlin, Germany, pages: 47-63, 2000.

[61] Serge E. Hallyn. *Role-based Access Control in SELinux.* Available at: *http://www.ibm.com/developerworks/linux/library/l-rbac-selinux/* (accessed May 23, 2014).

[62] D. Richard Kuhn. *Role Based Access Control on MLS Systems without Kernel Changes.* Proceedings of the third ACM workshop on Role-based access control, Fairfax, Virginia, United States, pages: 25-32, 1998.

[63] Thomas M Chalfant. *Role Based Access Control and Secure Shell - a Closer Look at Two* SolarisTM *Operating Environment Security Features.* Available at: *http://icc.mpei.ru/documents/00000805.pdf* (accessed May 23, 2014).

[64] D. Richard Kuhn.*Mutual Exclusion of Roles as a Means of Implementing Separation of Duty in Role-Based Access Control Systems.* ACM Workshop on Role Based Access Control, Proceedings of the second ACM workshop on Role-based access control, Fairfax, Virginia, United States, pages: 23-30, 1997.

[65] *Role Based Access Control - Frequently Asked Questions.* Available at: *http://csrc.nist.gov/groups/SNS/rbac/* (accessed May 23, 2014).

[66] Andreas Schaad, Jonathan Moffett and Jeremy Jacob. *The Role-Based Access Control System of a European Bank: A Case Study and Discussion.* ACM Workshop on Role Based Access Control, Proceedings of the sixth ACM symposium on Access control models and technologies, Chantilly, Virginia, United States, pages: 3-9, 2001.

[67] David F. Ferraiolo,D. Richard Kuhn and Ravi Sandhu. *RBAC Standard Rationale.* IEEE Security & Privacy (IEEE Press) Volume 5, Issue 6, pages: 51-53, 2007.

[68] Leonard J. Lapadula. *Rule-Set Modeling of a Trusted Computer System.* Available at: *http://www.acsac.org/secshelf/book001/09.pdf* (accessed May 23, 2014).

[69] Amon Ott, Simone Fischer-Hubner. *The 'Rule-Set Based Access Control' (RSBAC) Framework for Linux.* Available at: *http://www.rsbac.org/doc/media/rsbac-framework.pdf* (accessed May 23, 2014).

[70] Kenneth Ingham and Stephanie Forrest. *A History and Survey of Network Firewalls.* Technical. Report 2002-37, University of New Mexico Computer Science Department, 2002. Available at: *http://www.cs.unm.edu/~moore/tr/02-12/firewall.pdf* (accessed May 23, 2014).

[71] Chris Brenton and Cameron Hunt. *Network Security.* SYBEX, second edition, 2003.

[72] Willam Stallings and Lawrie Brown. *Computer Security.* PEARSON, Prentice Hall, 2008.
[73] John H. Howard, Michaell L. Kazar and Sherri G. Menees and etc. Scale and Performance in a Distributed File System. ACM Transactions on Computer Systems, Volume 6, No. 1, pages 51-81 February 1988.
[74] Stephen Smalley. *Configuring the SELinux Policy.* Available at: *http://www.nsa.gov/research/_files/publications/selinux_configuring_policy.pdf* (accessed May 23, 2014).
[75] Summers, C. Rita. *Computer Security: Threats and Safeguards.* McGraw Hill, 1997.
[76] D. D. Clark and D. R. Wilson. *A Comparison of Commercial and Military Computer Security Policies.* IEEE Symposium on Security and Privacy, pages 184-194, Oakland, April 1987.
[77] Lingyu Wang, Duminda Wijesekera and Sushil Jajodia. *A Logic-based Framework for Attribute Based Access Control* Proceedings of the 2004 ACM workshop on formal methods in security engineering, pages 45-55, Washington DC, 2004.
[78] Mohammad A. Al-Kahtani and Ravi Sandhu. *A Model for Attribute-Based User-Role Assignment* Proceedings of the 18th Annual Computer Security Applications Conference, page 353, 2002.
[79] Xinwen Zhang, Yingjiu Li and Divya Nalla. *An Attribute-Based Access Matrix Model* Proceedings of the 2005 ACM symposium on Applied Computing, pages: 359-363, Santa Fe, New Mexico, 2005.
[80] W.T. Polk.*Approximating Clark-Wilson access triples with basic UNIX controls.* UNIX Security Symposium IV (1993), pp. 145154.
[81] David Aspinall and Martin Hofmann. *Another Type System for In-Place Update.* Proceedings of the 11th European Symposium on Programming Languages and Systems, pages: 36 - 52, 2002.
[82] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. *Providing dynamic update in an operating system.* Proceedings of the annual conference on USENIX Annual Technical Conference, Pages: 32 -32, Anaheim, CA, 2005.
[83] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell and Iulian Neamtiu. *Mutatis mutandis: safe and predictable dynamic software updating.* Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages: 183-194, Long Beach, California 2005
[84] Daniel P. Bovet & Marco Cesati. *Understanding the Linux kernel*, *Third Edition*. Sebastopol, California: O'Reilly, 2005.
[85] Michael Beck, Harald Bohme, Mirko Dziadzka and Ulrich Kunitz. Linux kernel programming, Third Edition. King's Lynn, Norfolk, Great Brit-ain: Addison-Wesley Professional, 2002.
[86] Robert Love. *Linux kernel development, Third Edition*. Crawfordsville, Indiana: Addison-Wesley Professional, 2010.
[87] "Varaint Symlinks." Available at *http://leaf.dragonflybsd.org/cgi/web-man?command=ln&section=1* (accessed May 23, 2014).
[88] Uresh Vahalia. *Unix Internals: The New Fron-tiers*. Upper Saddle River, New Jersey: Prentice Hall, 1996.

[89] “Context dependent symbolic link.” Available from ubuntu.com. *http://manpages.ubuntu.com/manpages/dapper/ man8/ocfs2cdsl.8.html* (accessed May 23, 2014).

[90] Mike Ault, Madhu Tumma. *Oracle 10g Grid & Real Application Clusters: Oracle 10g Grid Computing with RAC*. Kittrell, North Carolina: Rampant Techpress, 2004.

[91] W. Richard Stevens. *UNIX Network Program-ming, Volume 2: Interprocess Communications, Second Edition*. Upper Saddle River, New Jersey: Prentice Hall, 2012.

[92] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker and Sheila A. Haghighat, “A Domain and Type Enforcement Unix Prototype.” In *Proceedings of the Fifth USENIX UNIX Security Symposium*, vol. 5, pages 12-12, June, 1995.

[93] David F. Ferraiolo, Janet A. Cugini. D.Richard Kuhn. “Role-based Access Control (RBAC): Features and Motivations.” In *11th Annual Computer Security Applications*, pages 241-48, December, 1995.

[94] David Wagner and Paolo Soto. “Mimicry attacks on host-based intrusion detection systems.” In *Proceedings of the 9th ACM conference on Computer and communications security.* Pages 255 - 264, 2002.

[95] Lance Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley, 2002.

[96] Hasani, Shabnam Mohammad, and Nasser Modiri. "Criteria Specifications for the Comparison and Evaluation of Access Control Models." *International Journal of Computer Network and Information Security (IJCNIS)* 5.5 (2013): 19.

[97] Konstantin Beznosov, Philip Inglesant, Jorge Lobo, Rob Reeder, and Mary Ellen Zurko. 2009. Usability Meets Access Control: Challenges and Research Opportunities, In *Proceeding of SACMAT '09 Proceedings of the 14th ACM symposium on Access control models and technologies* (Stresa, Italy, June 3-5, 2009). SACMAT’09. ACM, New York, NY, 73-74.

[98] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems, 2nd Edition*. Wiley. 2008.

[99] Alma Whitten. 2004. Making Security Usable, Doctoral thesis. CMU-CS-04-135. Carnegie Mellon University.

[100] Ronald Kainda, Ivan Flechais and A.W. Roscoe. 2010. Security and Usability: Analysis and Evaluation, In 2010 International Conference on Availability, Reliability and Security (Krakow, Poland, February 15-18, 2010). ARES’10. IEEE, 275–282.

[101] Alma Whitten, and J. D. Tygar. 1999. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In *Proceedings of the 8th conference on USENIX Security Symposium* (Washington, DC, August 23-26, 1999). SSYM'99. USENIX Association, Berkeley, CA, 14-14.

[102] Sonia Chiasson, P. C. van Oorschot, and Robert Biddle. 2006. A usability study and critique of two password managers. In *Proceedings of the 15th conference on USENIX Security Symposium* (Vancouver, B.C., Canada, July 31-August 04, 2006). USENIX-SS'06. USENIX Association Berkeley, CA. 1.

[103] Dinei Florêncio and Cormac Herley. 2010. Where do security policies come from? *In Proceedings of the Sixth Symposium on Usable Privacy and Security* (Redmond, Washington, July 14-16, 2010). SOUPS '10. ACM, New York, NY, Article No. 10.

[104] Lujo Bauer, Scott Garriss, and Michael K. Reiter. 2008. Detecting and resolving policy misconfigurations in access-control systems. In *Proceedings of the 13th ACM symposium on Access control models and technologies* (Estes Park, CO, June 11-13, 2008). SACMAT '08. ACM, New York, NY, 185-194.

[105] Lujo Bauer, Yuan Liang, Michael K. Reiter, Chad Spensky. 2012. Discovering Access-Control Misconfigurations: New Approaches and Evaluation Methodologies. In *Proceeding of the second ACM conference on Data and Application Security and Privacy* (San Antonio, TX, Feb 7-9, 2012). CODASPY '12. ACM New York, NY, 95-104.

[106] Hong Chen, Ninghui Li, Christopher S. Gates, and Ziqing Mao. 2010. Towards Analyzing Complex Operating System Access Control Configurations. In Proceedings of the 15th ACM symposium on Access control models and technologies (Pittsburgh, Pennsylvania, USA, June 9-11, 2010). SACMAT '10. ACM, New York, NY, 13-22.

[107] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin and David Wagner. 2012. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceeding of the Eighth Symposium on Usable Privacy and Security* (Washington, DC, July 11-13, 2012). SOUPS '12. ACM, New York, NY, Article No. 3

[108] D. K. Smetters and N. Good. 2009. How Users Use Access Control. In *Proceedings of the 5th Symposium on Usable Privacy and Security* (Mountain View, CA, July 15-17, 2009). SOUPS '09. ACM, New York, NY, 1–12.

[109] Sara Motiee, Kirstie Hawkey, and Konstantin Beznosov. 2010. Do Windows Users Follow the Principle of Least Privilege? Investigating User Account Control Practices. In *Proceedings of the Sixth Symposium on Usable Privacy and Security* (Redmond, WA, July 14-16, 2010). SOUPS '10. ACM, New York, NY, 1-13.

[110] Xiang Cao and Lee Iverson. 2006. Intentional Access Management: Making Access Control Usable for End-Users. In *Proceedings of the second Symposium on Usable Privacy and Security* (Pittsburgh, PA, July 12-14, 2006). SOUPS '06. ACM, New York, NY, 20–31.

[111] Michelle L. Mazurek, J.P. Arsenault, Joanna Bresee, Nitin Gupta, Iulia Ion, Christina Johns, Daniel Lee, Yuan Liang, Jenny Olsen, Brandon Salmon, Richard Shay, Kami Vaniea, Lujo Bauer, Lorrie Faith Cranor, Gregory R. Ganger, and Michael K. Reiter. 2010. Access Control for Home Data Sharing: Attitudes, Needs and Practices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, GA, April 10-15, 2010). CHI '10, ACM, New York, NY, 645-654.

[112] Z. CLIFFE SCHREUDERS, TANYA MCGILL, and CHRISTIAN PAYNE. 2011. Empowering End Users to Confine Their Own Applications: The Results of a Usability Study Comparing SELinux, AppArmor, and FBAC-LSM. *ACM Transactions on Information and System Security (TISSEC)*. v.14 n.2, 1-28, September 2011.

[113] Habib, Lionel, Mathieu Jaume, and Charles Morisset. "A formal comparison of the bell & lapadula and rbac models." *Information Assurance and Security, 2008. ISIAS'08. Fourth International Conference on*. IEEE, 2008.

[114] Sandhu, Ravi S. "A lattice interpretation of the Chinese Wall policy." *Proceedings of the 15th NIST-NCSC National Computer Security Conference*. 1992.

[115] Bertino, Elisa, et al. "A logical framework for reasoning about access control models." *ACM Transactions on Information and System Security (TISSEC)* 6.1 (2003): 71-127.

[116] Barkley, John. "Comparing simple role based access control models and access control lists." *Proceedings of the second ACM workshop on Role-based access control*. ACM, 1997.

[117] Thomsen, Daniel J., and J. T. Haigh. "A comparison of type enforcement and Unix setuid implementation of well-formed transactions." *Computer Security Applications Conference, 1990, Proceedings of the Sixth Annual*. IEEE, 1990.

[118] Chinaei, Amir H., Ken Barker, and Frank Wm Tompa. "Comparison of Access Control Administration Models." *Ubiquitous Communication and Computing Journal (UBICC)* 4.3 (2009).

[119] Wikipedia (2013), available at *http://en.wikipedia.org/wiki/Firewall_%28computing%29* (accessed May 23, 2014).

[120] Sandhu, Ravi, and Qamar Munawer. "How to do discretionary access control using roles." In *Proceedings of the third ACM workshop on Role-based access control*, pp. 47-54. ACM, 1998.

[121] Paul Ammann, Richard Lipton, and Ravi S. Sandhu, "The expressive power of multi-parent creation in monotonic access control models." *Journal of Computer Security*, 4(2-3):149–165, January 1996.

[122] Hu, Vincent C., and Karen Ann Kent. Guidelines for access control system evaluation metrics. US Department of commerce, *National Institute of Standards and Technology*, 2012.

[123] Ajay Chander, Drew Dean, and John C. Mitchell. A state-transition model of trust management and access control. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 27–43. IEEE Computer Society Press, June 2001.

[124] Srinivas Ganta. *Expressive Power of Access Control Models Based on Propagation of Rights*. PhD thesis, George Mason University, 1996.

[125] Qamar Munawer and Ravi S. Sandhu. Simulation of the augmented typed access matrix model (ATAM) using roles. In *Proceedings of INFOSECU99 International Conference on Information and Security*, 1999.

[126] Sylvia Osborn, Ravi S. Sandhu, and Qamar Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and System Security*, 3(2):85–106, May 2000.

[127] Ravi S. Sandhu. Expressive power of the schematic protection model. *Journal of Computer Security*, 1(1):59–98, 1992.

[128] Ravi S. Sandhu and Srinivas Ganta. On testing for absence of rights in access control models. In *Proceedings of the sixth Computer Security Foundations Workshop*, pages 109–118. IEEE Computer Society Press, June 1993.

[129] Mahesh V. Tripunitara and Ninghui Li. Comparing the expressive power of access control models. In *Proceedings of 11th ACM Conference on Computer and Communications Security (CCS-11)*, pages 62–71. ACM Press, October 2004.

[130] Tripunitara, Mahesh V., and Ninghui Li. "A theory for comparing the expressive power of access control models." *Journal of Computer Security* 15, no. 2 (2007): 231-272.

[131] Harrison, Michael A., Walter L. Ruzzo, and Jeffrey D. Ullman. "Protection in operating systems." *Communications of the ACM 19*, no. 8 (1976): 461-471.

[132] Sandhu, Ravinderpal Singh. "The schematic protection model: its definition and analysis for acyclic attenuating schemes." *Journal of the ACM (JACM)* 35, no. 2 (1988): 404-432.

[133] Snyder, Lawrence. "Theft and conspiracy in the Take-Grant protection model." *Journal of Computer and System Sciences*, 23(3):333–347, Dec. 1981.

[134] Sandhu, Ravi S. "The typed access matrix model." *Research in Security and Privacy, 1992. Proceedings, 1992 IEEE Computer Society Symposium on*. IEEE, 1992.

[135] Lipton, Richard J., and Lawrence Snyder. "A linear time algorithm for deciding subject security." *Journal of the ACM (JACM)* 24, no. 3 (1977): 455-464.

[136] Jones, Anita K., Richard J. Lipton, and Lawrence Snyder. "A linear time algorithm for deciding security." *In Foundations of Computer Science*, 1976, *17th Annual Symposium* on, pp. 33-41. IEEE, 1976.

[137] Bishop, Matt, and Lawrence Snyder. "The transfer of information and authority in a protection system." *In Proceedings of the seventh ACM symposium on Operating systems principles*, pp. 45-54. ACM, 1979.
[138]Lihui Hu, Jean Mayo and Charles Wallace. “An Empirical Study of Three Access Control Systems”. *In proceedings of the 6th International Conference on Security of Information and Networks*, pp. 287-291, ACM, 2013
[139]Lihui Hu and Jean Mayo. “MaskFS: Transparent Conditional Redirection in the File System”. Poster presentation at the 27th Large Installation System Administration Conference, 2013.

Appendix: Reuse License

Reuse license of the paper “An empirical study of three access control systems” for chapter 8 of this dissertation

This is a License Agreement between Lihui Hu ("You") and Association for Computing Machinery, Inc. ("Association for Computing Machinery, Inc.") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by Association for Computing Machinery, Inc., and the payment terms and conditions.

License Number 3367710028941

License date Apr 14, 2014

Licensed content publisher Association for Computing Machinery, Inc.

Licensed content publication Proceedings of the 6th International Conference on Security of Information and Networks

Licensed content title An empirical study of three access control systems

Licensed content author Lihui Hu, et al

Licensed content date Nov 26, 2013

Type of Use Thesis/Dissertation

Requestor type Author of this ACM article

Is reuse in the author's own new work? No

Format Print and electronic

Portion Full article

Will you be translating? No

Title of your thesis/dissertation A FIREWALL MODEL OF FILE SYSTEM SECURITY

Expected completion date May 2014

Estimated size (pages) 103