SPF: SECURITY PERFORMANCE FLEXIBILITY FRAMEWORK FOR TRUSTED OPERATING SYSTEMS

BY

ALBERT LIMING CHU

B.S., University of Illinois at Urbana-Champaign, 2001

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois Urbana-Champaign, 2002

Urbana, Illinois
ABSTRACT

The rapid growth of networking, data sharing, and the Internet has made computer security an important part of computer research and development. A number of highly secure operating systems have been developed to handle the increasing need for security. These operating systems, typically called Trusted Operating Systems, offer a number of security mechanisms that can help protect information, make a system difficult to break into, and confine attacks far better than traditional operating systems. However, this security will come at a cost, since it can degrade the performance of an operating system. This performance loss is one of the reasons why Trusted Operating Systems have not become popular.

While Trusted Operating Systems offer an incredible amount of security, observations about computing workloads suggest that only some parts of the operating system security are actually necessary. Web servers are the best example. For many web servers, the majority of the information on the server is publicly readable and available on the Internet. Therefore, if a Trusted Operating System is used on a web server, any security used to secure the confidentiality of the server’s information is not necessary. Any security used to protect the confidentiality of web server data can be considered a waste of computational resources. The security needed in web servers is the security to protect the integrity of data, not the confidentiality of data. Other workloads such as multimedia or database workloads may also only need parts of the operating system security.

Based on this observation, this thesis proposes the Security Performance Flexibility (SPF) framework for Trusted Operating Systems. SPF recognizes that not all computing workloads require all the security in Trusted Operating Systems. SPF allows system administrators to selectively disable parts of the security in Trusted Operating Systems. By disabling parts of the Trusted Operating System security, performance of the system can potentially be increased. The SPF framework allows system administrators to balance the security and performance needs in their particular computing environment.
To my parents.
More than anyone else, I would like to thank Professor Klara Nahrstedt for her guidance and motivation. I especially want to thank her for taking the time to discuss research and graduate school with me when I was an itsy bitsy teeny tiny insignificant little undergrad. Many Professors do not take the time to deal with graduate students let alone undergrads. She’s quite the opposite of Professor Smith ☺.

I’d also like to thank my friends and family for their support through my many years of college. I’d also like to thank the research assistants in the MONET research group for their help as I worked on my thesis. I’d like to thank Argus Systems Group for introducing me to Trusted Operating Systems during my internship with them in 1999. Without this internship, I would have never thought of doing a thesis on this topic.

Last, and certainly not least, I’d like to thank Siebel Systems for awarding me one of their five Siebel Scholar Fellowships during the 2001-2002 school year. Without it, I would not have been able to spend the Spring and Summer of 2002 working on my thesis.
TABLE OF CONTENTS

LIST OF FIGURES ................................................................................................................... ix
LIST OF TABLES ..................................................................................................................... x
LIST OF ABBREVIATIONS .................................................................................................... xi

CHAPTER 1 Introduction ........................................................................................................... 1
  1.1 Background .................................................................................................................... 1
  1.2 Trusted Operating System Security .............................................................................. 3
    1.2.1 Trusted Evaluation Criteria ......................................................................................... 3
    1.2.2 Discretionary Access Control (DAC) ........................................................................ 4
    1.2.3 Mandatory Access Control (MAC) .......................................................................... 5
    1.2.4 Least Privilege ........................................................................................................... 8
    1.2.5 Auditing .................................................................................................................... 10
    1.2.6 Architecture ............................................................................................................. 11
  1.3 Problem Description ..................................................................................................... 12
  1.4 Problem Solution ......................................................................................................... 13
  1.5 Solution Evaluation .................................................................................................... 16
  1.6 Thesis Contributions .................................................................................................. 19
  1.7 Thesis Organization .................................................................................................... 19

CHAPTER 2 Related Work ...................................................................................................... 21
  2.1 Operating System Security ........................................................................................... 21
    2.1.1 Role-Based Access Control (RBAC) ......................................................................... 22
    2.1.2 Domain and Type Enforcement (DTE) ..................................................................... 25
    2.1.3 Support for Flexible Security Policies ...................................................................... 26
    2.1.4 Operating Systems Security and Security Performance Flexibility ......................... 28
  2.2 Operating System Flexibility ....................................................................................... 29
    2.2.1 Microkernels and Exokernels ................................................................................... 29
    2.2.2 Extensible Kernels .................................................................................................... 31
    2.2.3 Operating System Flexibility and Security Performance Flexibility ....................... 33
  2.3 Quality of Service (QoS) and Quality of Security .......................................................... 33
    2.3.1 Quality of Service (QoS) ........................................................................................... 33
    2.3.2 Quality of Protection (QoP) and Quality of Security Service (QoSS) .......................... 34
    2.3.3 Quality of Service, Quality of Security and Security Performance Flexibility ........... 35
  2.4 Summary of Related Work ........................................................................................... 36

CHAPTER 3 Design .................................................................................................................. 37
  3.1 Trusted Operating System Secure Operations ............................................................... 37
  3.2 Security Performance Flexibility Framework Overview .................................................. 39
    3.2.1 Operations Supported by SPF .................................................................................... 39
    3.2.2 Security Performance Flexibility Design Assumptions ............................................ 41
# Table of Contents

3.3 System Wide Security Performance Flexibility (System-SPF) ......................................................... 42

3.3.1 Operations Supported by System-SPF ............................................................ 43
3.3.2 System-SPF Architecture and Design ............................................................ 43
3.3.3 System-SPF Design Advantages and Disadvantages ........................................ 45

3.4 Process Security Performance Flexibility (Process-SPF) .................................................................... 45
3.4.1 Operations Supported by Process-SPF ............................................................ 45
3.4.2 Process-SPF Architecture and Design ............................................................ 46
3.4.3 Process-SPF Design Advantages and Disadvantages ........................................ 50

3.5 Object Security Performance Flexibility (Object-SPF) ........................................................................ 50
3.5.1 Operations Supported by Object-SPF ............................................................... 51
3.5.2 Object-SPF Architecture and Design ............................................................... 51
3.5.3 Object-SPF Design Advantages and Disadvantages ........................................ 53

4.1 Development Environment ................................................................................................................. 55
4.2 SELinux File System Security Design and Implementation ................................................................. 56
4.3 System Wide Security Performance Flexibility (System-SPF) ............................................................ 58
4.3.1 System-SPF Implementation ............................................................................. 58
4.3.2 System-SPF Implementation Advantages and Disadvantages ............................... 63
4.3.3 System-SPF System Administration .................................................................... 63
4.3.4 System-SPF Security Issues .............................................................................. 63

4.4 Process Security Performance Flexibility (Process-SPF) .................................................................... 64
4.4.1 Process-SPF Implementation ............................................................................. 64
4.4.2 Process-SPF Implementation Advantages and Disadvantages ............................... 72
4.4.3 Process-SPF System Administration .................................................................... 72
4.4.4 Process-SPF Security Issues .............................................................................. 72

4.5 File Security Performance Flexibility (File-SPF) .................................................................................. 73
4.5.1 File-SPF Implementation ................................................................................. 73
4.5.2 File-SPF Implementation Advantages and Disadvantages .................................... 83
4.5.3 File-SPF System Administration .......................................................................... 84
4.5.4 File-SPF Security Issues .................................................................................... 85

4.6 Summary of Implementation ............................................................................................................... 86

CHAPTER 5 Performance Evaluation of Security Performance Flexibility ....................................................... 88

5.1 Benchmarks ........................................................................................................................................ 88
5.1.1 Microbenchmarks .............................................................................................................. 89
5.1.1.1 AIM9 ................................................................................................. 89
5.1.1.2 UnixBench ......................................................................................... 89
5.1.2 Standard Application Benchmarks .................................................................................. 89
5.1.2.1 AIM7 ................................................................................................. 89
5.1.2.2 WebStone ........................................................................................... 90
5.1.3 Original Application Benchmarks ..................................................................................... 90
5.1.3.1 MPEG Frame Rate ............................................................................. 90
5.1.3.2 Video on Demand Frame Rate ...................................................................... 90

5.2 Experimental Setup and Methodology ................................................................................................ 91
# LIST OF FIGURES

Figure 1.1: Illustration of Security Labeling................................................................. 6  
Figure 1.2: MAC Security Label Models........................................................................ 7  
Figure 1.3: Security Architecture................................................................................... 11  
Figure 1.4: Security Performance Flexibility Framework Architecture......................... 14  

Figure 2.1: RBAC Model............................................................................................... 23  
Figure 2.2: Flexible Security Policy Architecture....................................................... 27  
Figure 2.3: Microkernel Architecture.......................................................................... 30  
Figure 2.4: Extensible Kernel Architecture................................................................. 32  

Figure 3.1: Security Performance Flexibility Framework Architecture......................... 42  
Figure 3.2: System-SPF Architecture........................................................................... 44  
Figure 3.3: Process-SPF Architecture......................................................................... 48  
Figure 3.4: Propagation of Process-SPF Configuration............................................... 49  
Figure 3.5: Object-SPF Architecture.......................................................................... 54  

Figure 4.1: SELinux modification of the read() system call........................................... 57  
Figure 4.2: System-SPF bit masks............................................................................... 60  
Figure 4.3: System-SPF modification of the read() system call.................................... 62  
Figure 4.4: Process-SPF bit masks.............................................................................. 65  
Figure 4.5: Loading Process-SPF Configuration.......................................................... 67  
Figure 4.6: Process-SPF modification of the read() system call.................................... 68  
Figure 4.7: Illustration of Process-SPF commands...................................................... 70  
Figure 4.8: System call interface for Process-SPF....................................................... 71  
Figure 4.9: File-SPF bit masks.................................................................................... 74  
Figure 4.10: File-SPF ext2 file system inode flags....................................................... 77  
Figure 4.11: File-SPF modification of ext2_read_inode()............................................. 78  
Figure 4.12: File-SPF modification of ext2_write_inode()............................................ 78  
Figure 4.13: System call interface for File-SPF............................................................ 80  
Figure 4.14: Loading File-SPF Configuration.............................................................. 81  
Figure 4.15: File-SPF modification of the read() system call........................................ 82  

Figure 5.1: Performance of SELinux relative to Redhat, Expectations.......................... 96  
Figure 5.2: Performance of SELinux relative to Redhat, Large Workload Range........... 112  
Figure 5.3: Performance of SELinux relative to Redhat, Small Workload Range........... 112  

Figure A.1: Thesis WebStone File Distribution............................................................... 120  
Figure A.2: Standard WebStone File Distribution........................................................ 120
## LIST OF TABLES

Table 3.1: Security relevant directory operations .............................................................. 38
Table 3.2: Security relevant process operations ................................................................. 38
Table 3.3: Security relevant file operations ....................................................................... 38
Table 3.4: Security relevant network operations ............................................................... 38
Table 3.5: Security relevant IPC operations ...................................................................... 38
Table 3.6: Operations supported by SPF ........................................................................... 39
Table 3.7: Operations supported by System-SPF ............................................................... 43
Table 3.8: Operations supported by Process-SPF .............................................................. 46
Table 3.9: Operations supported by File-SPF ................................................................. 51
Table 3.10: Operations supported by Network-SPF .......................................................... 51
Table 3.11: Operations supported by IPC-SPF ............................................................... 51

Table 4.1: Security relevant directory operations and relevant system calls ..................... 56
Table 4.2: Security relevant file operations and relevant system calls ............................. 56
Table 4.3: Operations supported by System-SPF and relevant system calls .................... 58
Table 4.4: Functions modified to implement System-SPF ............................................... 61
Table 4.5: Operations supported by Process-SPF and relevant system calls ................... 64
Table 4.6: Functions modified to implement Process-SPF ............................................... 68
Table 4.7: Functions modified to support Process-SPF ................................................... 72
Table 4.8: Operations supported by File-SPF and relevant system calls ........................... 74
Table 4.9: Functions modified to implement File-SPF .................................................... 82
Table 4.10: Functions modified to support File-SPF ....................................................... 82

Table 5.1: AIM9 Benchmark Results, Security ChecksExecuted ......................................... 99
Table 5.2: AIM9 Benchmark Results, Security Checks Skipped ....................................... 99
Table 5.3: AIM9 Benchmark Improvement, Security Checks Skipped .................... 99
Table 5.4: UnixBench Benchmark Results, Security Checks Executed ............................ 100
Table 5.5: UnixBench Benchmark Results, Security Checks Skipped ............................ 100
Table 5.6: UnixBench Benchmark Improvement, Security Checks Skipped ................... 100
Table 5.7: AIM7 Database Throughput Results, read() and seek() security disabled .... 102
Table 5.8: AIM7 Database Throughput Improvement, read() and seek() security disabled 102
Table 5.9: WebStone Throughput Results, read(), stat(), and open() security disabled ... 104
Table 5.10: WebStone Throughput Improvement, read(), stat(), and open() security disabled 104
Table 5.11: MPEG Frame Rate, 30 Frames/Second, read() security disabled ............... 107
Table 5.12: MPEG Frame Rate Improvement, 30 Frames/Second, read() security disabled ... 107
Table 5.13: MPEG Frame Rate, 60 Frames/Second, read() security disabled ............... 107
Table 5.14: MPEG Frame Rate Improvement, 60 Frames/Second, read() security disabled ... 107
Table 5.15: VOD Frame Rate, 30 Frames/Second, read() security disabled .................. 110
Table 5.16: VOD Frame Rate Improvement, 30 Frames/Second, read() security disabled ... 110
Table 5.17: VOD Frame Rate, 60 Frames/Second, read() security disabled .................. 110
Table 5.18: VOD Frame Rate Improvement, 60 Frames/Second, read() security disabled ... 111
**LIST OF ABBREVIATIONS**

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACL(s)</td>
<td>Access Control List(s)</td>
</tr>
<tr>
<td>API</td>
<td>Application Program Interface</td>
</tr>
<tr>
<td>BSD</td>
<td>Berkeley Software Distribution</td>
</tr>
<tr>
<td>CGI</td>
<td>Common Gateway Interface</td>
</tr>
<tr>
<td>Config</td>
<td>Configuration</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DAC</td>
<td>Discretionary Access Control</td>
</tr>
<tr>
<td>DDT</td>
<td>Domain Definition Table</td>
</tr>
<tr>
<td>DIT</td>
<td>Domain Interaction Table</td>
</tr>
<tr>
<td>DoS</td>
<td>Denial of Service</td>
</tr>
<tr>
<td>DTE</td>
<td>Domain and Type Enforcement</td>
</tr>
<tr>
<td>DTEL</td>
<td>Domain and Type Enforcement Language</td>
</tr>
<tr>
<td>DTMACH</td>
<td>Distributed Trusted Mach</td>
</tr>
<tr>
<td>DTOS</td>
<td>Distributed Trusted Operating System</td>
</tr>
<tr>
<td>File-SPF</td>
<td>File Security Performance Flexibility</td>
</tr>
<tr>
<td>FPS</td>
<td>Frames Per Second</td>
</tr>
<tr>
<td>GSS</td>
<td>Generic Security Service</td>
</tr>
<tr>
<td>SGSS</td>
<td>Seraphim GSS</td>
</tr>
<tr>
<td>HP</td>
<td>Hewlett-Packard</td>
</tr>
<tr>
<td>HTML</td>
<td>Hyper Text Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hyper Text Transfer Protocol</td>
</tr>
<tr>
<td>HTTPD</td>
<td>Hyper Text Transfer Protocol Daemon</td>
</tr>
<tr>
<td>ID(s)</td>
<td>Identifier(s)</td>
</tr>
<tr>
<td>I/O</td>
<td>Input and Output</td>
</tr>
<tr>
<td>IPC</td>
<td>Interprocess Communication</td>
</tr>
<tr>
<td>IPC-SPF</td>
<td>Interprocess Communication Security Performance Flexibility</td>
</tr>
<tr>
<td>ISSO</td>
<td>Information Systems Security Officer</td>
</tr>
<tr>
<td>ITSEC</td>
<td>Information Technology Security Evaluation and Certification</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>LIDS</td>
<td>Linux Intrusion Detection System</td>
</tr>
<tr>
<td>MAC</td>
<td>Mandatory Access Control</td>
</tr>
<tr>
<td>MHZ</td>
<td>Megahertz</td>
</tr>
<tr>
<td>MLS</td>
<td>Multi-Level Security</td>
</tr>
<tr>
<td>MPEG</td>
<td>Moving Picture Expert Group</td>
</tr>
<tr>
<td>Network-SPF</td>
<td>Network Security Performance Flexibility</td>
</tr>
<tr>
<td>NFS</td>
<td>Network File System</td>
</tr>
<tr>
<td>Object-SPF</td>
<td>Object Security Performance Flexibility</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>OSDB</td>
<td>Open Source Database Benchmark</td>
</tr>
<tr>
<td>PID(s)</td>
<td>Process Identifier(s)</td>
</tr>
<tr>
<td>Process-SPF</td>
<td>Process Security Performance Flexibility</td>
</tr>
<tr>
<td>QDDA</td>
<td>Quick and Dirty Development Application Benchmark</td>
</tr>
<tr>
<td>QoP</td>
<td>Quality of Protection</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>QoSS</td>
<td>Quality of Security Service</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RBAC</td>
<td>Role-Based Access Control</td>
</tr>
<tr>
<td>RSBAC</td>
<td>Rule Set Based Access Control</td>
</tr>
<tr>
<td>SA</td>
<td>Security Administrator</td>
</tr>
<tr>
<td>SELinux</td>
<td>Security-Enhanced Linux</td>
</tr>
<tr>
<td>SO</td>
<td>Security Officer</td>
</tr>
<tr>
<td>SPF</td>
<td>Security Performance Flexibility</td>
</tr>
<tr>
<td>System-SPF</td>
<td>System Wide Security Performance Flexibility</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TCSEC</td>
<td>Trusted Computer System Evaluation Criteria</td>
</tr>
<tr>
<td>TOS</td>
<td>Trusted Operating System</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>VOD</td>
<td>Video On Demand</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

1.1 Background

In the last ten years, the rapid growth of networking, data sharing, and the Internet have made computer security an important part of computer research and development. It is an important issue for both general computer users and computer scientists. Although a large amount of research has gone into the development of security technology, computing systems as a whole are still inadequate to provide high security in today’s systems. Many of today’s computer systems still lack the basic ability to protect important information and stop malicious attacks. A quick glance at the Defacement Archives web site shows how often computer attacks can occur in today’s systems [DEFACE].

In particular, security developed and placed in the application layer of a system is the most inadequate to provide high security. Unfortunately, this is where a large majority of security has been researched and developed. This software includes (but is not limited to) cryptography, authentication, firewalls, and application layer access control [LOSCOCCO98]. Cryptography and authentication techniques are probably the most popular form of security implemented and researched today. While it is extremely difficult to break, it has been and always will be subject to brute force or man in the middle attacks [COLOURIS01]. Firewalls and application layer access control can also help towards preventing malicious attacks. However, they do not provide the granularity necessary to keep a system secure, they cannot protect against internal security problems, and they can often be averted [LOSCOCCO98].

The largest vulnerabilities, posed to application layer security, are Trojan horses and other malicious code such as viruses. Once malicious code is inside a user’s application space, application layer security can do almost nothing to stop it. A malicious program running under a
user’s account can access all data and information accessible under the account. All the data can be tampered with or transferred to other individuals.

Paul Clark perhaps stated it best, stating that most of today’s systems are, “crunchy on the outside, yet soft and chewy on the inside” [CLARK00]. The quote points out that most of today’s computer security is placed in the application layer of an operating system and at network entry points. However, there is little to no security placed inside an operating system kernel. All of the security problems that exist at the application layer suggest that operating system security mechanisms might protect a system far better.

In fact, a number of operating systems have been developed to include a wide range of security mechanisms inside the operating system kernel. These operating systems provide a much more fine-grained degree of security. They can prevent many of the security problems mentioned above. Surprisingly, most of the security technologies developed for operating systems are not new. Much of it was originally developed during the 1970s and 1980s. However, due to system administration complexity, cost, and performance, these highly secure operating systems did not achieve commercial success [CLARK00]. The government and military are the only organizations that heavily use these highly secure operating systems [FERRAILO92].

Due to recent rising security concerns in the commercial sector, many of these operating system security technologies are being looked at and implemented again. Vendors such as Argus-Systems Group, Hewlett-Packard, and Sun Microsystems are developing highly secure operating systems with these security technologies [ARGUS] [TRUSTEDSOLARIS] [VIRTUALVAULT]. Several open source operating systems are also being developed with high security in the kernel. The NSA has released a secure version of Linux called Security-Enhanced Linux (SELinux) [SELINUX]. Linux distributions such as Rule Set Based Access Control Linux (RSBAC) and Linux Intrusion Detection System (LIDS) are also available [LIDS] [RSBAC]. In addition, a secure version of BSD is being developed [TRUSTEDBSD].

These highly secure operating systems are typically called Trusted Operating Systems (TOS), Trusted Systems, or Secure Systems. The definitions of these terms are interpreted differently
between organizations. Therefore, no single term can effectively describe all of these highly secure operating systems. This thesis will use the term Trusted Operating System whenever these systems as a whole are referred to.

1.2 Trusted Operating System Security

Before delving into the problem this thesis will cover, an introduction to the traditional principles of Trusted Operating Systems will be discussed. As was mentioned earlier, the term Trusted Operating System is defined differently throughout literature. However, there seems to be four key features in all Trusted Operating Systems. They are Discretionary Access Control (DAC), Mandatory Access Control (MAC), Least Privilege, and auditing. This section will first discuss a bit of history behind Trusted Operating Systems. Then we will discuss these four key features. Finally, the architecture of Trusted Operating Systems will be discussed. More recent research in Trusted Operating Systems will be discussed in Chapter 2.

1.2.1 Trusted Evaluation Criteria

Research on operating system security mechanisms has been going on since the 1970s. However, operating system security requirements were not solidified until the Department of Defense published the Trusted Computer System Evaluation Criteria (TCSEC) in 1983 [CLARK00] [DOD85]. The TCSEC became the basis for several later security evaluations [ABRAMS95]. Common Criteria, a joint effort by European and North American nations, is the most prominent of these evaluations [COMMON]. Although Common Criteria is the most modern and up to date standard published, the TCSEC is still highly recognized.

While the TCSEC solidified many security ideas into one concept, its greatest contribution was the rating system it developed. The ratings system established criteria to rate the strength of security mechanisms in operating systems. The strength of the security is classified into one of four classes. The classes are indicated as class A, B, C, or D, where class A systems have the highest amount of security and class D systems have the least [ABRAMS95]. There are
sub-classes within each of these classes, but the details will not be described here. For additional details see [ABRAMS95] or for a complete description see [DOD85].

Class D systems have minimal or no security built into the operating system. It is the class used to describe any operating system that cannot meet at least a C rating. Although it may be surprising, MS-DOS and Windows 95/98/Me are considered class D operating systems [TANENBAUM01].

Class C systems provide Discretionary Security Protection or Discretionary Access Control (DAC). They also provide auditing features. Both of these features will be described more thoroughly in Sections 1.2.2 and 1.2.5 respectively. Many modern day operating systems, such as UNIX, Linux, and Windows NT, are considered class C operating systems.

In addition to DAC and auditing, Class B systems provide Mandatory Security Protection or Mandatory Access Control (MAC). MAC will be described more thoroughly in Section 1.2.3. In very general terms, class B systems are Trusted Operating Systems. Argus System’s Pitbull, Hewlett-Packard’s Virtual Vault, and Sun Microsystems Trusted Solaris are all class B systems. Due to the strict criteria of the TCSEC, the NSA’s Security-Enhanced Linux is not considered a class B system. This is despite the fact that it offers most of the requirements to be considered a class B system.

Class A systems have auditing, DAC, MAC, and the ability to prove the system provides the security indicated. Generally speaking, no commercial vendors develop or distribute class A systems. They primarily exist in military environments.

1.2.2 Discretionary Access Control (DAC)

Discretionary Access Control (DAC) security mechanisms let users determine who is authorized to access objects and how they can be accessed [ABRAMS95]. For example, DAC mechanisms may allow the owner of a file to give all other users in the system the ability to read a file. However, the same owner may not allow others to write to the file. The best example of DAC
mechanisms in practice can be seen in UNIX and Linux operating systems. In these operating systems, every file contains a set of bits that indicate the read, write, and execute permissions for the file. Every user is allowed to modify the permissions for the files they own. They may grant read, write, or execute access for themselves, anyone in their group, or to all users in the system [TANENBAUM01]. It is important to note that DAC allows users to modify security permissions for themselves and not just for other users in the system. This feature of DAC can help prevent human error [ABRAMS95]. For example, DAC can stop a user from accidentally deleting or modifying a file they do not wish to.

DAC is the most widely used security mechanism placed in operating systems today. Unfortunately, DAC is not capable of providing high security in an operating system and is one of the major security weaknesses in today’s operating systems. DAC lacks the ability to protect both the confidentiality and integrity of information. Trojan horses are the best example of how DAC is ineffective. Malicious code can take advantage of a user’s DAC and modify the security permissions of all the files they own. Malicious code can make files publicly readable or writeable, thus destroying the confidentiality and integrity of the files [LOSCOCCO98]. Similarly, DAC cannot always protect against malicious insiders or even mistakes by authorized users.

1.2.3 Mandatory Access Control (MAC)

Mandatory access control (MAC) can be best described as security that is not at the discretion of users [ARGUS01A]. In other words, security permissions are not dependent on how a user sets permissions on objects; the security is under complete control of the operating system and the policy set by a system administrator [LOSCOCCO98]. MAC can control the flow of information in a system much more tightly than DAC can. How MAC provides higher security than DAC is best explained through an explanation of their implementation and modeling.

Typically, MAC is implemented through the assignment of a security label (also known as a sensitivity label or integrity label) to every object in an operating system [JOSHI01]. The objects include processes, files, sockets, network packets, semaphores, message queues, and
most other objects or structures in an operating system [LOSCOCC01B]. Figure 1.1 illustrates how a simple security label might be implemented into a UNIX or Linux file system [DALTON01]. As can be seen in the figure, an additional label field has been added to the *inode* structure of each file in the operating system. Although the example indicates that an integer is used for security label, it can be a string or even another structure. By assigning a label to every object in a system, a wide variety of MAC security policies can be implemented inside the operating system. This allows MAC to control access to files and resources much more strongly than DAC [JOSHI01].

```c
struct inode {
    uid_t    i_uid;
    gid_t    i_gid;
    mode_t   i_mode;
    /* etc. all other fields in an inode structure */
#ifdef TRUSTED_OS
    int      label;
#endif
}
```

Figure 1.1: Illustration of Security Labeling.

For example, Hewlett-Packard has developed a secure version of Linux with trusted security mechanisms built into the kernel [DALTON01]. In their system, security labels are assigned to every object in the system. The labels partition the system into distinct compartments. Throughout the kernel, security labels are compared between processes and objects the process wishes to access. If the security labels are not equal, access is denied. These security checks ensure that processes and objects in one compartment cannot read, write, or interfere with any process or object in another compartment. This type of policy is illustrated by Figure 1.2a. The circle represents all objects and resources in the system. As can be seen, all objects in the system have been partitioned into four different areas, labeled compartments A through D. Each compartment does not have to be equal in size nor do the resources in each partition have to be equally distributed.

By partitioning the system into compartments, higher security can be achieved than with just DAC. As an example, let us consider a Trojan horse that makes a user’s confidential data publicly readable. MAC can limit the malicious effect of the Trojan horse. Only users and
processes inside the compartment will be able to read the data made public by the Trojan horse. Users and processes from different compartments will be unable to read the data. This shows that the potential consequences of security breaches can be greatly reduced with MAC.

![Diagram of MAC Security Label Models]

Figure 1.2: MAC Security Label Models.

(a) HP Secure Linux. (b) Bell-LaPadula MLS model. (c) Argus Pitbull.

While the compartmentalization of a system is perhaps the easiest way to see how MAC works, it is not the most common modeling of security labels. The most commonly used modeling of security labels is the Bell-LaPadula model. It is primarily used in military systems to protect the confidentiality of information [DALTON01] [LANDWEHR81]. The security labels are used to implement a hierarchical multi-level security (MLS) system [JOSHI01] [TANENBAUM01]. In this type of system, the security policy does not allow users at a certain security level to view any object at security level higher than itself [ABRAMS95] [TANENBAUM01]. Throughout the kernel, security checks are done to ensure that security labels of a process dominate the security label of an object or resource it wishes to access. This type of policy is illustrated in Figure 1.2b, where Top Secret is the highest security level and Unclassified is the lowest security level. This type of system ensures that confidential data is protected against disclosure to unauthorized individuals. This is despite any type of DAC security that may exist. It ensures that Trojan Horses, malicious insiders, or even mistakes by individuals, will not allow confidential data to leak outside the security level in which it has been placed. Other hierarchical MLS security models have also been developed. For example, the Biba model is very similar to the Bell-
LaPadula model, but it is more concerned with protecting the integrity of data rather than its confidentiality [ABRAMS95]. While all hierarchical systems differ in some fashion, they will all be referred to as MLS systems. As the years have passed, MAC and MLS have almost become one and the same concept [SELINUX]. The reader should keep in mind that MLS is only one way MAC can be modeled.

As one last example of how labels can be used to implement different types of MAC, Figure 1.2c shows how security labels are modeled in Argus System's Pitbull. Pitbull’s implementation of security labels can be considered a mixture of the two previously discussed models. Each security label is actually a combination of two different components, a hierarchical component and a compartmental component. These labels allow a multilevel security policy to be enforced within different compartments of the operating system [ARGUS01A].

MAC is the cornerstone behind Trusted Operating Systems security. Using security labels to implement different MAC policies can confine and limit many of the security issues that were mentioned earlier. It is important to note that MAC does not replace DAC. Typically, both are implemented in a system, allowing some security discretion for users. MAC typically limits how users can modify security permissions for the files they own [ABRAMS95]. In addition, system administrators are also subject to MAC security checks. This is unlike DAC security checks, which root processes are typically allowed to bypass [ARGUS01A].

1.2.4 Least Privilege

The principle of least privilege states that users and processes should only have the privileges necessary to complete their tasks, nothing more and nothing less [ABRAMS95]. It can be very difficult to administer the minimum amount of privileges for each user and process. However, giving any user or process more authority than it needs can open the door for security breaches.

There are various ways least privilege can be implemented. MAC can be used to implement least privilege if security labels are set up properly [LOSCOCO98]. However, the most
popular way to implement least privilege is through mechanisms similar to capabilities. The permissions in capabilities indicate what a process is allowed to access and what types of actions a process is allowed to take. Every process in the system holds a list of capabilities. These capabilities list the actions a process is allowed to perform during its lifetime. With the previously mentioned security mechanisms, least privilege can help make a system far more secure.

Let us consider a network daemon that runs in the background of a network server. Capabilities can indicate what specific actions the daemon is allowed to perform and not perform. For example, capabilities can indicate that the daemon is allowed to transfer network packets but is not allowed to access files. These permissions limit the ability of an attacker to exploit a bug or backdoor in the network daemon. If a bug or backdoor is discovered, the capabilities ensure that he attacker cannot use the security hole to access files.

Separating the power of a system administrator is the most powerful use of least privilege mechanisms [BISHOP89]. Most systems today have an all-powerful account, often called the root or superuser account. System administrators handle a variety of administrative duties, such as setting up file systems, accounts, and applications, with this account. Usually, the root user is not subject to any security checks, thus making it easier to handle administrative duties.

However, having an all-powerful user in your system can be dangerous. While it makes system administration easier, an attacker will have free reign to do whatever they wish, if the root account (or any process running under the root account) can be compromised [ABRAMS95] [ARGUS01A] [BISHOP89] [WALKER96].

In order to limit the power of the root user, many trusted systems separate the power of the root user into a number of smaller powers, usually called privileges [ARGUS01A]. The privileges can be collected into sets of privileges, each of which can be assigned to a sub-administrator [ABRAMS95] [ARGUS01A]. A set of sub-administrators, each with a subset of the original

\[1\] Capabilities are interpreted and used differently throughout computer literature. For example, [BOVET01] defines capabilities more broadly than [SILBERSCHATZ98] and [TANENBAUM01] do. For the purposes of this thesis, capabilities can be considered any type of security permissions that are “carried” along with a process.
root user’s privileges, is collectively responsible for the system administration duties of the system. The root user account is eliminated. If an attacker or piece of malicious code gains power as one of these sub-administrators, the division of privileges will limit the amount of damage the attacker can perform. The attacker can only perform actions limited to the sub-administrators abilities. For example, in Argus System’s Pitbull, the original superuser account has been divided into three sub-administrators called the Security Administrator (SA), Security Operator (SO), and Information Systems Security Officer (ISSO). Each of these administrators has a subset of the privileges of the original root user. Each sub-administrator is responsible for a subset of the system administrative duties. Some of the duties of the original root user can only be done through cooperation between two sub-administrators [ARGUS01A]. This division of power can severely limit the effect of an attack through an administrative account or process. Along with DAC and MAC, least privilege helps make a system highly secure.

1.2.5 Auditing

As mentioned earlier in Section 1.2.1, auditing is an important part of Trusted Operating Systems security. Auditing is the recording or logging of all security relevant operations and transactions in a system [ABRAMS95]. Recording information is the key to identifying the source of an attack or foreseeing future attacks [PICCIOTTO87]. Most systems today are configured to provide some basic logging of information, however most are not equipped to handle the fine grained auditing necessary for highly secure systems. In particular, most systems cannot audit user accesses to files or system resources. This information is vital towards discovering security holes or foreseeing future security problems [PICCIOTTO87].

On the other hand, Trusted Operating Systems are equipped to handle fine-grained auditing and have the ability to stop any user or process from modifying the logged information [DOD85]. The protection of the log is also a key component to Trusted Operating Systems. Without it, attackers or malicious users can hide their activity and actions.
Determining what actions to audit and the analysis of logged information is a much more complicated topic that reaches into the area of intrusion detection. Information on that topic will be deferred to [LUNT88] and [LUNT93]. For the purposes of this thesis, auditing can be considered the ability to log fine-grained information in a system and the ability to protect against destruction or modification of the log.

1.2.6 Architecture

The basic architecture of Trusted Operating Systems is illustrated below in Figure 1.3. As a reminder to the reader, the architecture of a system is not the same as the implementation. A variety of implementations for trusted security mechanisms can be done.

![Security Architecture Diagram](image)

Figure 1.3: Security Architecture.

(a) Standard Operating Systems. (b) Trusted Operating Systems.

Traditional operating systems follow the architecture shown in Figure 1.3a [SILBERSCHATZ98]. All applications and middleware interface with the operating system through the system call interface. In most operating systems, there is little to no security in the operating system. If there is security in the operating system kernel, it is usually a small amount
that only implements DAC. This is illustrated in Figure 1.3a with a relatively thin layer of kernel security checks.

In Trusted Operating Systems, there is a much larger amount of security placed into the operating system [ALDRICH94]. Figure 1.3b illustrates this security with a much thicker layer of kernel security checks. What is inside the kernel security check layer of a Trusted Operating System depends on the implementation. The kernel security layer may include DAC, MAC, Least Privilege, auditing, or any number of additional security features. The key point is that the kernel security checks are much larger than traditional operating systems. This large layer of security causes Trusted Operating Systems to suffer performance degradation. All system calls to the kernel must go through this layer of security checks before they can do any useful work.

1.3 Problem Description

As has been mentioned earlier, most Trusted Operating Systems are used in the military. A number of efforts have been made to make trusted systems technology more suitable for commercial organizations. In particular, recent research has concentrated on developing security models and flexible security policies that are more suitable for commercial organizations. This will be discussed in more detail in Chapter 2.

One of the areas that has not been heavily researched is the performance of Trusted Operating Systems and its effects on users. As the architecture in Figure 1.3b shows, the additional security checks in the kernel will cause Trusted Operating Systems to be slower than traditional operating systems [DALTON01]. This is one reason why Trusted Operating Systems have not become popular in the commercial sector. As we will show in Chapter 2, the performance slow down of Trusted Operating Systems may actually grow larger with the security mechanisms now being researched.

In some systems (in particular non-military systems), this high demand on security may not be completely necessary. The best example may be seen in web servers. As [NAHUM99] indicates, the typical web transaction goes through a sequence of fourteen operations. Of these
fourteen steps, nine go through additional security checks for reading data from disk or sending data on a network socket. However, the majority of web servers deal with pure public information. With the majority of data on a web server being public, the need for security checks during reads from disk seems like a waste of CPU cycles. This loss occurs for data that is already known to be readable by every user on the Internet. The real security need for web servers seems to be the security of write accesses, not read accesses. In other words, the integrity of the information is what is important, not the confidentiality.

Trusted Operating System security may also affect multimedia and video streaming services. Typically, when video is played, each frame is read off of disk one by one and displayed on a screen. In the case of video streaming, the frame is sent out through a socket onto the network. Every single read from disk and every write out to a network socket is now slowed down by repeated security checks. This may have a significant effect on the quality and frame rate of video, especially if the system is heavily loaded. Much like the example with web servers, this video quality degradation may not be entirely necessary. For example, a Video On Demand server may care more about the quality of the video stream than the security of read accesses to the server.

1.4 Problem Solution

As was stated in the previous section, there are several types of system workloads that repeatedly do security checks in a Trusted Operating System. For these workloads, the security checks may be undesired or completely unnecessary. The information in these workloads could be public, integrity of the system may be the primary concern, or quality of the workload is more desirable than the security of certain operations. This suggests that performance in Trusted Operating Systems can be increased if security can be disabled in some parts of the operating system. Performance can be increased if system administrators are given the ability balance their security and performance needs.

In order to provide better performance for specific system workloads, this thesis proposes the security performance flexibility (SPF) framework for Trusted Operating Systems. The SPF
framework allows system administrators to disable certain security checks in a Trusted Operating System. This gives system administrators the ability to balance their security needs with their performance needs.

Figure 1.4: Security Performance Flexibility Framework Architecture.

The architectural idea behind the SPF framework is illustrated in Figure 1.4. The SPF framework gives system administrators the option of disabling security checks for specific system calls. By skipping security checks in the kernel, performance for a system can be increased. For example, a system administrator can use SPF to turn off all read security checks in a web server. By turning off the read security checks of a web server, it is possible the web server’s throughput can be increased.
There are different levels at which the SPF framework can be implemented in a Trusted Operating System. Three different levels have been identified for evaluation in this thesis:

1. System Wide Security Performance Flexibility (System-SPF).

System-SPF provides the ability to disable all trusted security checks in particular operations of the system. For example, all read security checks can be disabled in the system. By disabling the read security checks in the entire system, performance of the system as a whole can improve. This may be particularly useful for dedicated web servers, because web servers generally have pure public information. Disabling read security checks for the entire web server may help increase web server throughput.

However, System-SPF may not offer enough fine-grained ability for system administrators to manage their system. For example, a system administrator may want some applications to skip read security checks, but not all applications. Process-SPF provides the ability to disable security checks in specific applications or processes. For example, a system administrator may disable read security for a MPEG video player. Therefore, every time a video frame is read from disk, the security check of this read operation will be skipped. By skipping the read security checks, we may be able to improve the quality of the MPEG video being played. In this example, Process-SPF only disables the read security on the MPEG video player. Different security checks can be disabled in other applications or processes.

Object-SPF provides system administrators with a different fine-grained technique to manage their system. Object-SPF provides system administrators the ability to disable security checks on individual objects in the system. Objects can refer to files, network connections, interprocess communication (IPC) objects, etc. Let us consider the MPEG video player example from before. In order to improve MPEG video quality, we want to disable read security checks when a video is being played. However, some systems may have both publicly readable video and private videos that are not public. The system administrator may want to disable read security checks
only on the public videos and not the private videos. Object-SPF allows system administrators to designate individual files that should have security checks skipped. Similarly, Object-SPF allows system administrators to disable security checks on specific network connections, such as sockets, or IPC objects, such as message queues.

Since Object-SPF refers to individual objects in the operating system, Object-SPF can be discussed in terms of the different types of objects that can be implemented within an Object-SPF framework. We will only consider files, network connections, and IPC objects in this thesis. The design and implementation of SPF at each of these object levels will be referred to as:

1. File Security Performance Flexibility (File-SPF)
2. Network Security Performance Flexibility (Network-SPF)
3. Interprocess Communication Security Performance Flexibility (IPC-SPF)

It should be noted that the SPF framework is not a proposal to modify security policy in a Trusted Operating System. It actually makes the security mechanisms flexible in a Trusted Operating System. It determines if a security check for a system call should be skipped or not. If the security policy is modified instead of the security mechanism, the performance increase cannot be obtained. As an example, consider the files on a web server. The security policy can be modified to indicate all web server files as publicly readable. However, performance cannot be gained from this policy configuration, because a security check will still be done to ensure the file is still publicly readable. This security check will occur every time a web server file is accessed. Only by modifying the security mechanism, to skip certain security checks, can performance improvements be gained.

1.5 Solution Evaluation

In order to evaluate the potential of the SPF framework, it will be programmed into the NSA's Security-Enhanced Linux (SELinux) for Linux kernel 2.4.18 [SELINUX]. SPF’s performance
and implementation will be analyzed to see what potential the SPF framework can have compared to both SELinux and Redhat version 7.2.

Of the Trusted Operating Systems available, SELinux was selected for several reasons. First, Linux is becoming one of the most popular operating systems worldwide. Therefore, using Linux to study the SPF framework will be ideal, since its performance is representative of the performance required for today’s systems. Second, SELinux's standard security configuration is a mixture of three types of security mechanisms, type enforcement, Role-Based Access Control (RBAC), and Multi-Level Security (MLS) [SMALLEY01]. Its security is highly representative of the traditional and more recent research efforts in Trusted Operating Systems.

The most important reason for selecting SELinux is the fact that Linux is open source and the code for SELinux is available for modification. The source code for commercial or military Trusted Operating System cannot be obtained. Besides the handful of other secure Linux distributions, the only other operating system that can be considered is Trusted BSD [TRUSTEDBSD]. SELinux was selected over Trusted BSD due to Linux's current popularity.

While SELinux’s open source code is its primary benefit, it is also its primary downside. Generally, Trusted Operating System source code is not released to the public, since security holes can be discovered in the code. SELinux is primarily being developed as a research and instructional project for the Linux community. SELinux is constantly changing, as different ideas and features are discovered and implemented into the kernel. The developers of SELinux admit that security holes may exist in SELinux since they have not been looked for extensively [SELINUX]. Despite this problem, SELinux is still a good operating system to evaluate the SPF framework with.

For this thesis, the SPF framework will only be implemented into the file system of SELinux. When this research began, network security had not been completed in SELinux. Therefore, enabling SPF in the networking components of SELinux was not considered.\(^2\) We did not consider SPF with IPC because we do not have an application benchmark that uses IPC heavily.

\(^2\) As of October 2002, the majority of network security has been integrated into SELinux.
Since SPF will only be evaluated within the context of file systems, it is not necessary to implement all of Object-SPF. In other words, there is no need to implement Network-SPF or IPC-SPF. Therefore, this thesis will only implement System-SPF, Process-SPF, and File-SPF. System-SPF, Process-SPF, and File-SPF will be implemented in three different kernels so the advantages and disadvantages of each can be studied. In addition, performance measurements will be done to determine if any of System-SPF, Process-SPF, or File-SPF has a performance advantage over the others. In the future, all three of these may be placed into one system together.

In order to evaluate the potential for SPF, the following criteria will be considered:

1) Performance Enhancement

The primary goal of the SPF framework is to see how much performance improvement we can gain, if any at all. It is possible the SPF overhead may dominate the SELinux security overhead, thus performance cannot be increased. A variety of microbenchmarks and application benchmarks will be used to evaluate the performance of SPF.

2) System Administration Complexity

Trusted Operating Systems are more difficult for system administrators to manage than traditional operating systems. Trusted Operating Systems require additional time to setup the security policy, monitor systems activities, and manage changes to the security policy. It is essential that the system administration of SPF is simple enough that it does not burden system administrators further. System administrative complexity will be evaluated based on the commands created for our SPF implementation in SELinux.
3) Security Side Effects

It is possible that the SPF framework can open security holes. It is quite possible that SPF cannot be implemented without opening security holes and negating the security of Trusted Operating Systems. Therefore, the security of the SPF framework must be analyzed and its effect on SELinux security will be evaluated.

1.6 Thesis Contributions

The key contributions of this thesis are:

- The novel idea that not all security in a Trusted Operating System is necessary.

- The novel idea that skipping non-essential security checks in a Trusted Operating System may increase system performance.

- The novel SPF framework for Trusted Operating Systems that gives system administrators the ability to balance security and performance needs of a system.

- The novel recognition that the SPF framework can be implemented at different levels within a Trusted Operating System.

- The novel development of system administrative commands that allow system administrators to configure a Trusted Operating System dynamically as the system executes.

- Extending Quality of Security to include Trusted Operating System security mechanisms.

1.7 Thesis Organization

The outline of the thesis is as follows: Chapter 2 discusses various types of research that SPF is based off of and how it is similar or different to SPF. Chapter 3 will go over the design of the
SPF framework. Chapter 4 will detail the implementation of SPF in SELinux. Chapter 5 will go over the experiments and benchmarks used to evaluate the performance of SPF. Finally, Chapter 6 will offer conclusions and future work.
CHAPTER 2

Related Work

There is a minimal amount of research that can really be considered similar to our SPF framework. However, the SPF framework is similar to ideas from three different areas of research. The first of these areas is security. Related work in this area will be discussed in Section 2.1. The other two areas of research are operating system flexibility and Quality of Service (QoS). Operating system flexibility has been researched for quite some time, through the development of microkernels, exokernels, and extensible kernels. These topics will be discussed in Section 2.2. A light overview of Quality of Service (QoS) will be discussed in Section 2.3 and then its relation to security will be discussed. A summary of these related works and their relation to SPF will be covered in Section 2.4.

2.1 Operating System Security

Section 1.2 gave a general outline of traditional Trusted Operating Systems concepts and implementation. This section will cover several of the recent research efforts in operating systems security. Section 2.1.1 will discuss Role-Based Access Control (RBAC). Then a discussion of Domain and Type Enforcement (DTE) will be presented in Section 2.1.2. Security policy flexibility will be discussed in Section 2.1.3. Finally, a summary of the recent security efforts as a whole and their relation to SPF will be discussed in Section 2.1.4.

There are two prominent themes in operating system security research that should be pointed out. The themes are:
1) Making system administration easier.

As has been mentioned earlier, system administration of Trusted Operating Systems is far more complex than traditional operating systems. Recent research efforts have looked at ways to make system administration easier in Trusted Operating Systems.

2) Adding features to allow more security policies to be enforced.

In traditional Trusted Operating Systems, security policies are limited to those that can fit the security model implemented in the kernel. Thus, recent research efforts have looked at security mechanisms that can enforce a wider variety of security policies.

The reader should take note that the security mechanisms discussed in this section still follow the same architecture described in Section 1.2.6. These systems are implemented far differently than traditional Trusted Operating Systems. However, the security mechanisms are still operating systems based and inside the kernel.

2.1.1 Role-Based Access Control (RBAC)

As discussed in Chapter 1, a variety of MAC models have been implemented, each of which is unique and more suitable in different environments. For example, some models may be more suited more for military systems while others may be more suitable for commercial organizations. Overall though, traditional Trusted Operating System models do not apply well to commercial organizations. Commercial organizations are structured in ways that make security requirements difficult to map into one of the traditional MAC models.

Role-Based Access Control (RBAC) is a security model that associates every user in the system with a role. Based on this role, the operating system controls the user’s access to objects and resources [FERRAILOLO92]. Compared to traditional MAC models, RBAC is considered a far more suitable security mechanism for commercial organizations. RBAC’s potential for commercial use has made it a hot research topic in recent years. Note that RBAC is a MAC
mechanism. It controls access permissions in the operating system and is not subject to a user’s discretion [FERRAIOLO92].

As an example of how RBAC may be more suitable in a commercial organization, let us consider the roles several employees at a hospital might have [FERRAIOLO92]. A doctor is allowed to read a patient’s medical history, append medical examinations to a patient’s medical history, and place prescriptions. However, a doctor cannot modify a patient’s medical history and dispense medication. A pharmacist is allowed to read a prescription order and dispense medication, but pharmacists are not allowed to read a medical history or modify the prescription. A nurse should be able to read a patient’s medical history and prescription, but should not be able to modify either.

![Figure 2.1: RBAC Model.](image)

The previously mentioned MAC models cannot support a security policy for this example hospital environment. The access permissions and access control needed in this environment cannot map to any of the MAC models described in Section 1.2.3. The security requirements for each of these roles cannot be placed properly into its own compartment or hierarchical level. The RBAC model is much more suitable for the security needs of this example hospital. The security requirements of each type of hospital employee can map directly to a user role. Figure 2.1 illustrates how RBAC might differ from the traditional MAC models. In some respects, RBAC compartmentalizes a system, but allows overlap to occur between compartments.
Even though RBAC models security needs differently than the traditional methods, it does not have to eliminate DAC or MAC. For example, RBAC can define DAC policies for how each role is allowed to modify security permissions for the files they own [SANDHU98]. The traditional MAC models, shown in Figure 1.2, can also be implemented within a RBAC environment. [OSBORN00] and [NYANCHAMA96] show how MAC models can be mapped into the roles of a RBAC system. The ability for RBAC to support both DAC and traditional MAC security is one of the reasons why it has become a popular research topic. It is considered by some to be the future of security in operating systems [JOSHI01]. Just like traditional MAC, RBAC can effectively protect against many security problems and help confine security problems.

RBAC systems are also designed so that they are easier to manage than traditional Trusted Operating Systems. In both RBAC and traditional Trusted Operating Systems, system administration is much more complicated than traditional operating systems. Security permissions must be set on all files and objects before the security policy can be enforced. In traditional Trusted Operating Systems, the task of modifying these permissions can be very time consuming.

However, RBAC systems are designed to make system administration easier [GLIGOR96]. Let us consider a company in which an employee was moved from one division to another. When this employee is moved, the access permissions of this user must be modified. In traditional Trusted Operating Systems, access permissions for all of the user’s files and programs must be modified. However, in a RBAC system, only the user's role needs to be changed [JOSHI01]. RBAC system administration has been implemented at a level of abstraction that makes system administration easier [FERRAILO95]. Easier system administration also makes a system more secure. It lowers the possibility that policy or administrative mistakes can occur [GLIGOR96].

The principle of least privilege can also be supported in a RBAC system [FERRAILO92] [FERRAILO95]. RBAC can limit the actions that each user in the system is allowed to perform. The role of the root user can also be split into different roles, effectively dividing the root account into a number of sub-administrators.
Most implementations of RBAC use a combination of policy databases and ACLs to enforce the RBAC policy. The policy databases are used to map users to roles. The ACLs enforce the security policy within the system [BARKLEY98] [FRIBERG97] [RBAC]. Any detailed explanation of RBAC implementation and configuration will be deferred to the example implementation provided at [RBAC].

2.1.2 Domain and Type Enforcement (DTE)

Domain and Type Enforcement (DTE) is a relatively new topic of research. DTE strives to development a Trusted Operating System that is easier to administer and can enforce a broader range of security policies [SMITH96]. It offers enough policy flexibility that any of the traditional MAC models can be enabled within the system [SMITH96].

In DTE operating systems, every object is associated with a type and every user is associated with a domain [BADGER95]. DTE is implemented through the use of two global access matrices called the Domain Definition Table (DDT) and the Domain Interaction Table (DIT). The DDT specifies the security permissions allowed between all domains and objects. Every time a user tries to access an object or perform an operation, the DDT is accessed to check if permission is allowed. The DIT is similar to the DDT except that it defines the security policy used to control domain-to-domain interaction [BADGER95].

The key feature of DTE is its Domain and Type Enforcement Language (DTL). DTL is a high level language used to describe the security policy configuration in a DTE system. It enables the DDT and DIT to be programmed quickly and efficiently. The DTL allows system administrators to quickly implement a wide variety of security policies. Therefore, DTL can help make DTE system administration easier.

DTE may seem very similar to the MAC mechanisms described earlier. In particular, DTE types seem to be similar to security labels and DTE domains seem to be similar to compartments. Therefore, DTE may not seem to offer much more than MAC. However, DTE offers a much
more fine-grained amount of security control than traditional MAC. In traditional MAC, all objects and processes are confined to a compartment or hierarchical security level. Security mechanisms, in the operating system kernel, compare security labels to see if access is permitted. However, DTE can let system administrators enforce far stricter restrictions [SMITH96]. For example, processes in domain X can be limited to reading files of type Y and writing files of types Z. This cannot be enabled in the traditional MAC models.

DTE can also enforce least privilege. Processes can be placed into specific domains. Each of these domains can limit the type of operations processes can perform [WALKER96]. Therefore, if a security breach occurs through one of these processes, the attacker is confined to the operations allowed in the domain.

DTE can be considered similar to RBAC. The DDT and DIT can be set to implement a model of security similar to Figure 2.1. Unlike HP’s Secure Linux, there is no need to have strict partitioning of objects and resources. The DIT can specify interactions between different domains. Thus, the DTE essentially allows overlap between domains.

2.1.3 Support for Flexible Security Policies

While the recent research efforts discussed earlier enforce a variety of security policies, they are still quite limited. The security policy that can be enforced is limited to the security mechanisms implemented in the kernel. This suggests that flexible security policies are necessary to meet the needs of all system administrators. In order to offer a much broader range of security policies, an operating system architecture must be developed to support flexible security policies. RBAC, DTE, and any specific security implementation will not suffice, because specific security implementations limit the types of security policy than can be enforced. The Distributed Trusted Operating System (DTOS) and Flask operating system are two Trusted Operating Systems developed with an architecture that can support virtually any security policy. DTOS and Flask are implemented in microkernels (Mach and Fluke respectively), however the architecture developed can be extended to many other operating systems [SPENCER99]. For example, the NSA’s Security-Enhanced Linux architecture is based off of the Flask architecture.
The basic architecture of DTOS and Flask is shown in Figure 2.2 [MINEAR95] [SPENCER99].

Unlike the traditional approaches of adding security layers in the kernel, there are two additional subsystems in this architecture. The object manager's responsibility is to call the security server every time a user attempts to access an object. The security server checks the security policy configuration and informs the object manager if access is permitted or denied. Note that the security server is not part of the kernel. It is a separate module that can be called by the kernel. It can also be modified or replaced. In DTOS and Flask, the security server is a process in the system [OLAWSKY96]. This is because DTOS and Flask are built upon microkernels. In Security-Enhanced Linux, the security server can be compiled into the kernel or linked as a loadable kernel module.

![Diagram of Flexible Security Policy Architecture](image)

Figure 2.2: Flexible Security Policy Architecture.

By having the security server in a separate module, almost any security policies and mechanisms can be implemented. Security need not be fixed or statically placed within the operating system kernel [OLAWSKY96] [SPENCER99]. Modifying, updating, or replacing the security server, even while the system is executing, will change the security policies and mechanisms of the system. In addition, notice that the object manager always contacts the security server for security checks. This feature allows security policies to be modified dynamically as the system executes.

---

3 In some respects, Figure 2.2 is not correct because DTOS and Flask are built on microkernels. However, to maintain consistency with the previous diagrams, the diagram is not significantly altered.
executes. Once the policy inside the security server is modified, every security check thereafter will be based on the new policy.

In order to increase performance in DTOS and Flask, security policy caching was suggested as a means to increase performance. The most recently referenced security checks are stored in a software-implemented cache, located in the object manager [OLAWSKY96] [SPENCER99]. This can help increase performance. It might be particularly useful for applications such as a Video On Demand server. If video frames from the same file are repeatedly read from disk, the same security checks are done over and over again. By caching recent security policies in the object manager, some parts of the security check can be skipped. When security policies are changed, the operating system is responsible for flushing recently accessed security policies out of the object manager’s cache.

2.1.4 Operating Systems Security and Security Performance Flexibility

As stated throughout this section, it seems the future direction of security in operating systems is centered on two themes. These two themes are the development of Trusted Operating Systems with easier system administration and mechanisms to enforce a wider range of security policies. [MAZIERES97] suggests that future development of security must be done on flexible operating systems, such as microkernels or exokernels. These operating systems allow us to simplify security code and discover security holes far more easily. Flexible security policies are a feature that may be useful for a number of organizations. However, this thesis will show that security policy flexibility is not the only flexibility that may be useful. Performance of security features should also be considered a flexible parameter.

In addition, the development of features to support easier system administration and wider ranges of security policy enforcement may result in additional performance loss. RBAC security checks must go through policy databases and ACLs every time a transaction occurs. DTE security checks must go through table lookups in the DDT and DIT. DTOS and Flask must go through two security modules for each security check. Security features that cover a wider range of goals will suffer greater performance losses than traditional security methods because the code cannot
be optimized. This suggests that development of SPF in operating systems may become more important in future Trusted Operating Systems. It will become important because recent research efforts indicate the performance gap between normal operating systems and Trusted Operating Systems will grow.

### 2.2 Operating System Flexibility

Implementing flexibility into operating systems is not a new concept. It has been studied for quite some time. It began with research of microkernels, then later exokernels and extensible kernels. Research on such kernels began because most operating systems are monolithic, difficult to upgrade, and difficult to debug [TANENBAUM01]. Placing flexibility in an operating system gives it the ability to be upgraded and debugged far more easily. This section will give an overview of the research done to make operating systems more flexible. Section 2.2.1 will discuss microkernels and exokernels. Section 2.2.2 will discuss extensible kernels. Finally, operating system flexibility and its relation to security and SPF is discussed in Section 2.2.3.

#### 2.2.1 Microkernels and Exokernels

Microkernels have been researched for quite some time, even since the mid 1980s. A variety of microkernel systems have been developed, perhaps the most famous being Mach. [LIEDK95] gives a good overview of many of these microkernel systems. Microkernel research began because most operating systems are monolithic pieces of code. Their monolithic nature makes these operating systems unreliable, full of bugs, difficult to port, and hard to modify [TANENBAUM01].

On the other hand, microkernels attempt to minimize the size of the operating system. Microkernels only have the bare minimum mechanisms necessary to control hardware resources. Microkernels separate the operating system mechanisms that control hardware from the operating system features that make an operating system unique [RASHID89]. For example, microkernels offer features such as mapping of logical addresses to physical addresses,
interprocess communication (IPC), and I/O port access. However, they do not offer services that are typically implemented in traditional operating systems. For example, they do not offer virtual memory management, device drivers, file systems or sockets [GOLUB90] [LIEDK95]. Instead, these services are programmed as user level processes.

The basic architecture of microkernels is shown in Figure 2.3 [GOLUB90] [TANENBAUM01]. As can be seen, some traditional operating system services have been migrated into user land. Unlike the architecture shown in Figure 1.3 and Figure 2.2, there is no middleware. Application programs, that require traditional operating system services, must use the kernel’s IPC mechanisms to communicate with the appropriate user land service [GOLUB90]. As Figure 2.3 illustrates, the user application utilizes the microkernel’s IPC mechanism to communicate with the file system manager, so that it can access a file. The file system manager then communicates with the disk device driver.

![Figure 2.3: Microkernel Architecture.](image)

By placing the majority of services in user land, microkernels become far more modular than traditional operating systems. The modularity of the system makes it far more flexible and modifiable. The modularity can benefit upgrading, debugging, portability, system extensions, and support for legacy applications [GOLUB90] [LIEDTKE95] [RASHID89]. By allowing an operating system to be far more modular, the system can be modified and tuned for optimal performance. In addition, the majority of bugs can be confined to the service process.
This can make a system far more stable since bugs will no longer crash the entire system, it will only crash a user process.

Exokernels are similar to microkernels except they minimize the operating system kernel further. Exokernels offer low-level support for the multiplexing of resources and nothing else [ENGLER95A]. Unlike microkernels, exokernels do not support IPC mechanisms. Almost everything in an exokernel is migrated into user land. This allows application processes and libraries to have even more control over hardware resources. The goal of exokernels is to eliminate as many operating system abstractions as possible, because operating system abstractions often cause poor performance, reliability, adaptability and flexibility [ENGLER95B].

### 2.2.2 Extensible Kernels

The previous section’s discussion, of microkernels and exokernels, suggest that the best way to offer flexibility in operating systems is to migrate operating system services to user land. While this may be true, the microkernel and exokernel approach may not offer the best solution for operating system flexibility [SELTZER95]. In many situations, the flexibility needed will be small or minor. Let us consider the scheduling algorithms and page replacement algorithms that are implemented in an operating system. Workloads will perform differently based on the scheduling algorithm and page replacement algorithms implemented in an operating system [SELTZER95]. Applications might want to select the best scheduling algorithms and page replacement algorithms so that their performance can be maximized. These are only small parts of the operating system. It may not be necessary to move scheduling services into user land in order to make these algorithms flexible.

Extensible kernels are a mix between microkernels and traditional monolithic kernels. They place flexible options or extensions inside an operating system kernel. This allows applications to modify kernel behavior for their needs. Extensible kernels give applications an interface to modify a small set of system parameters. In contrast, microkernels and exokernels require a user land program to be modified if a small parameter needs to be changed.
Figure 2.4 illustrates the architectural idea in extensible kernels [SELTZER95]. After an application makes a system call to the kernel, the kernel will execute the appropriate extension that is selected. Unlike microkernels, the flexibility of extensible kernels can be placed within the kernel of a traditional monolithic operating system [SELTZER95]. This implementation of operating system flexibility may be particularly useful for legacy applications and legacy operating systems [GHORMLEY98].

There has been a variety of research done on extensible kernels. [SELTZER97] gives an overview on the extensible kernels that have been developed. There are a variety of issues involved with extensibility of operating systems. [BERSHAD95] gives a good overview of these issues.
2.2.3 Operating System Flexibility and Security Performance Flexibility

As has been mentioned in this section, flexibility in operating systems can provide a number of great benefits to the applications running in the operating system. In particular, flexibility allows the operating system to be modified easily. Thus, performance enhancements in an operating system can be integrated more easily. Flexibility in traditional operating systems can also apply to Trusted Operating systems. Flexibility can benefit Trusted Operating Systems by allowing security checks to be skipped if they are considered unnecessary. By skipping security checks, performance increases in the operating system can be gained.

Notice that the SPF framework is quite similar to the architecture of extensible kernels. Notice that Figure 2.4 is quite similar to Figure 1.4. Instead of the five extensions from Figure 2.4, there are only two extensions in Figure 1.4. The extensions in Figure 1.4 are either the execution of the kernel security check or skipping the kernel security check.

2.3 Quality of Service (QoS) and Quality of Security

The SPF framework is also similar to the concept of Quality of Service (QoS). QoS is one of the most researched topics in Computer Science today. The discussion of QoS and its mechanisms can go quite in depth. Therefore, only the basic ideas behind it will be discussed in Section 2.3.1. For more in depth information, we will defer to [BRUNO98], [BRUSTOLONI99], [CHU97], [RAJKUMAR98] and [STEINMETZ95]. After the overview of QoS, its relation to security will be discussed in Section 2.3.2. The topics of Quality of Protection (QoP) and Quality of Security Service (QoSS) will be discussed in Section 2.3.2. Section 2.3.3 will summarize the research topics in this section and discuss how they relate to SPF.

2.3.1 Quality of Service (QoS)

Quality of Service (QoS) is the specification of service needs for a resource [STEINMETZ95]. A service may require anything from CPU cycles to network bandwidth. QoS is necessary for services to gather and retain the resources it needs. Without QoS, a service cannot accomplish a
task with guarantees. For example, a service may wish to guarantee performance at a certain rate or completion in a certain amount of time.

The most common use of QoS is with multimedia. As an example, consider a MPEG movie. Video playback requires that frames be played at certain intervals apart from each other. For example, if a video has a frame rate of 30 frames/second, each frame should be played 33.3 milliseconds apart. In order for the video to play smoothly, there must be guarantees from the underlying system that every frame can be played within 33.3 milliseconds after the previous frame. In most operating systems, there is no underlying mechanism that can guarantee this. For example, several processes may be introduced that have higher scheduling priorities than the MPEG video player. These processes may delay processing of the next video frame and cause the frame to play late. When frames are played late, the video will subsequently look choppy. Although QoS is typically associated with multimedia, it can also be associated with generic applications.

2.3.2 Quality of Protection (QoP) and Quality of Security Service (QoSS)

The Generic Security Service Application Program Interface (GSS-API) is an API proposed by RFC 1508 (which was later obsoleted by RFC 2078 and RFC 2743) [LINN93]. The GSS-API is the specification of an API that can be used with security aware network applications. The API was intended to become a standard, so that underlying protocols can be modified and applications can be ported more easily.

The GSS-API is one of the first pieces of literature to acknowledge that security incurs overhead cost. Since this is a network API, the overhead cost in question is the encryption or decryption of data. The overhead cost amount depends on the cryptographic algorithm used and the length of the cryptographic key. RFC 1508 specifies that the GSS-API should have a Quality of Protection (QoP) parameter to indicate what type of security is needed for every message sent out on a network. The QoP parameter is necessary because certain types of network data may not need cryptography or may not want to suffer the overhead cost associated with each message. The QoP parameter can be used to avoid the security overhead entirely or it can be used to select
a less computationally intensive cryptographic algorithm than the default. [LIU01] implements QoP into an extension of the GSS-API called the Seraphim GSS-API (SGSS-API). The SGSS-API implements nine different QoP protection levels that can be selected by application developers. The protection levels offer differing amounts of cryptographic security based on algorithm complexity and key length.

Quality of Security Service (QoSS) is a proposed extension of QoP to cover a broad range of security services, such as authentication or intrusion detection [IRVINE00]. It may be difficult to see how authentication or intrusion detection can have different levels of quality. As an example, authentication can have security ranging from low-level authentication schemes, such as passwords, to medium authentication schemes, such as smart cards, and to high-level authentication schemes, such as biometrics. [TALWAR01] implements QoSS into a user identification and user mobility architecture for ubiquitous environments. The architecture can negotiate different cryptographic key lengths based on the QoSS requirements a user requests.

In addition to the security services, QoSS also allows ranges of security to be specified. The range of security indicates a minimal and maximal amount of security. For example, the range of security can indicate low and high cryptographic key lengths. With this range of security, multimedia-streaming services can select a cryptographic algorithm that stays within the range of desired security but streams the best video quality possibility. Specifying a range of security gives services the opportunity to dynamically adjust to fit the needs of their services, while staying within the desired range of security. This is different from QoP, in which a certain type of security must be selected.

2.3.3 Quality of Service, Quality of Security and Security Performance Flexibility

Section 2.3 illustrates how applications and services may require resource guarantees so that they can complete a desired task with certain guarantees. From this viewpoint, SPF is very similar to QoS. Trusted Operating Systems security mechanisms may hurt the performance requirements a system administrator requires. The SPF framework will give system administrators the ability to disable security checks so their performance requirements can be met.
The ideas behind QoP and QoSS are also similar to SPF. QoP, QoSS and SPF all recognize that security is not a fixed parameter and is something that can be adjustable. QoP, QoSS and SPF all recognize that different qualities and performance can be gained by adjusting security. While QoP, and QoSS are similar to SPF in these respects, there are some differences. For example, ranges of quality can be specified in QoS and QoSS. For example, a movie player may want to guarantee that a movie will be played between 30 and 60 frames per second. These types of ranges cannot be specified in the SPF framework. The SPF framework specifies the set of system calls that should skip security checks in the Trusted Operating System. Technically speaking, the set of system calls selected to skip security checks can represent a range of SPF qualities, however these ranges are quite different from the ranges specified by QoS and QoSS.

### 2.4 Summary of Related Work

This chapter has covered three different areas of research that can be considered similar to SPF. First, security research for operating systems concentrates on methods to make operating system security easier to manage and far more flexible than in the past. Operating system flexibility concentrates on different kernel designs that allow kernel features to be more modular and thus easily modifiable. These ideas are very similar to concepts in the SPF framework. The SPF framework adds flexibility to Trusted Operating Systems. Unlike recent security research, SPF involves methods to make security mechanisms flexible, not just the security policy. By implementing flexible security mechanisms to skip security checks, performance can be gained. The concepts of QoP and QoSS in Section 2.3 are similar to SPF in the respect that they realize security does not have to be a fixed parameter. QoP and QoSS realize that security can be adjusted to suit a particular application’s performance needs.
CHAPTER 3

Design

In this chapter, we will discuss the design of the SPF framework. As was mentioned in Section 1.4, SPF can be implemented into different levels of an operating system. SPF can be implemented system wide (System-SPF), within individual processes (Process-SPF), or within objects of the operating system (Object-SPF).

This chapter will first discuss the operations Trusted Operating Systems typically implement additional security with. This will be covered in Section 3.1. The design of System-SPF will be discussed in Section 3.2. The design of Process-SPF will be discussed in Section 3.3. The design of Object-SPF will be discussed in Section 3.4.

Even though this thesis will only implement and evaluate SPF in the context of file system performance, we will discuss the design of the SPF framework with networking and interprocess communication (IPC) as well. However, the design of the SPF framework will focus more on file systems. The reader should note that the design described in this chapter is not specific to SELinux or any Trusted Operating System. It can be applied and implemented into any Trusted Operating System.

3.1 Trusted Operating System Secure Operations

There are four major operating system subsystems that Trusted Operating Systems implement additional security with. These subsystems are processes, files, networking, and IPC. The security relevant operations of these subsystems are listed in Tables 3.1 through 3.5. Since directories are almost an entire subsystem different than normal files, security relevant directory operations are listed in a separate table.
<table>
<thead>
<tr>
<th>Make Directory</th>
<th>Remove Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Directory</td>
<td>List Directory Contents</td>
</tr>
</tbody>
</table>

Table 3.1: Security relevant directory operations.

<table>
<thead>
<tr>
<th>Execute Program</th>
<th>Kill Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fork Process</td>
<td>Set Process Parameters</td>
</tr>
</tbody>
</table>

Table 3.2: Security relevant process operations.

<table>
<thead>
<tr>
<th>Create File</th>
<th>Delete File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open File</td>
<td>Read File</td>
</tr>
<tr>
<td>Write File</td>
<td>Copy File</td>
</tr>
<tr>
<td>Seek File</td>
<td>Get File Attributes</td>
</tr>
<tr>
<td>Set File Attributes</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: Security relevant file operations.

<table>
<thead>
<tr>
<th>Setup Connection</th>
<th>Configure Network Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send Message</td>
<td>Receive Message</td>
</tr>
</tbody>
</table>

Table 3.4: Security relevant network operations.

<table>
<thead>
<tr>
<th>Create IPC Object</th>
<th>Destroy IPC Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Receive from IPC Object</td>
<td>Write/Send to IPC Object</td>
</tr>
</tbody>
</table>

Table 3.5: Security relevant IPC operations.
3.2 Security Performance Flexibility Framework Overview

3.2.1 Operations Supported by SPF

Only a handful of the operations listed in Section 3.1 may benefit from Security Performance Flexibility. SPF should benefit applications and workloads that do repeated operations over and over again. We can gain performance increases by eliminating security checks in these repeated operations. If a security relevant operation is executed rarely, then security checks will not make up a significant portion of the workload. Therefore, we may not be able to obtain performance increases, by skipping security checks, in those operations. We have identified the following operations as ones that should be supported in the SPF framework.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Directory</td>
<td>Open File</td>
<td>Setup Connection</td>
<td>Read/Receive from IPC Object</td>
</tr>
<tr>
<td>List Directory Contents</td>
<td>Read File</td>
<td>Configure Network Connection</td>
<td>Write/Send to IPC Object</td>
</tr>
<tr>
<td></td>
<td>Write File</td>
<td>Send Message</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Copy File</td>
<td>Receive Message</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Seek File</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Get File Attributes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.6: Operations supported by SPF.

For shared systems, such as a University computer shared by students, changing directories, listing directory contents, and copying files occur many times during a system’s daily execution. However, in some systems, these operations will never be malicious. By skipping the security checks for these operations, better performance can be gained for the system overall.

As was mentioned earlier, most files on a web server are publicly readable. Therefore, any security checks done to check read accesses for a web page file is a waste of CPU cycles. [NAHUM99] points out that three of the major file system operations done on a web server: opening a file, determining the size of the file, and reading the file from disk. Therefore, it may be beneficial if SPF supports opening files, reading files, and getting file attributes.
[NAHUM99] also lists major network operations done when a web page request is made. There are five major network operations done on each web page request: accepting the connection, configuring the connection, receiving a web request, sending a HTTP header, and sending the requested file. This means five additional network security checks will be done on every web page request. This may further decrease web server performance. Therefore, it may be beneficial if SPF supports the network operations listed in Table 3.6.

Adding SPF support for seeks can benefit database applications that may read random blocks on a database file. Skipping security checks during disk seeks can help reduce latency or search time for database queries.

SPF may be able to improve performance in applications with a large amount of synchronization between processes. IPC and synchronization can be a bottleneck in the performance of synchronizing processes. Therefore, skipping security checks for some IPC objects may help decrease the effect of this performance bottleneck.

The write operation on files is the only operation that may not have a well-known application or workload that can benefit from SPF on write operations. However, it may be potentially useful for some systems, which is why it has been added to Table 3.6. For example, computers set up as routers may benefit from SPF on write operations, because the file system might only be used to store temporary files.

The reader may be interested why none of the security relevant process operations in Table 3.2 are listed in Table 3.6. Operations related to processes are executed once or very few times. Therefore, SPF cannot improve performance significantly by skipping security checks in process operations. For example, only one security check is done when a program is first executed. The Trusted Operating System performs a security check to ensure that the user is allowed to execute the particular application. After the process begins executing, the process continues until it completes. Similarly, the security check done to kill a process will be done only once.
We should note that the operations listed in Table 3.6 are the operations that may benefit in the overall framework of SPF. However, the operations may not be applicable to all of the SPF designs discussed in this chapter.

### 3.2.2 Security Performance Flexibility Design Assumptions

Several assumptions will be necessary for our design of the Security Performance Flexibility framework. They are indicated below:

- Operating systems have a system call interface that applications use to interface with the operating system.

- Files, network, and IPC objects are the only objects in the operating system.

These assumptions are not required for the design in this chapter to be correct. The assumptions are used as a point a reference to describe the SPF framework and illustrate the architecture in diagrams. For example, microkernels may not have a system call interface. Applications access services through the microkernel’s IPC mechanisms. However, this interface is similar to a system call interface. Therefore, the design of the SPF framework in this chapter can still be used with microkernels.

### 3.2.3 Security Performance Flexibility Architecture and Design Overview

A high level architectural view of the SPF framework is shown below in Figure 3.1. It is the same as Figure 1.4. As was mentioned in Chapter 1, the goal of SPF is to skip the security checks of certain system calls.

There are a number of questions that are raised by the figure.

- How does the kernel know if it should skip the security check in system call X?
• How does a system administrator configure the system to skip the security check in system call X?
• Where will the SPF Configuration information be stored?

As was mentioned in Chapter 1, there are various levels in which SPF can be implemented. The answers to these questions will depend on the level in which the SPF framework is implemented. The following sections will describe the architecture and design of SPF at those levels and answer the above questions. The level at which SPF is designed will have implications about the operations, from Table 3.6, that can be supported and system administrative difficulty. Each of these issues will be considered in the design.

![Security Performance Flexibility Framework Architecture](image)

**Figure 3.1: Security Performance Flexibility Framework Architecture.**

### 3.3 System Wide Security Performance Flexibility (System-SPF)

As was mentioned in Chapter 1, System-SPF allows a system administrator to disable security checks in a set of system calls. The security checks in these system calls are skipped system wide, for every process and application. This may particularly benefit dedicated servers such as
web servers, Video On Demand servers, and database servers. System-SPF may be particularly useful in dedicated servers because dedicated servers tend to have specific workloads that are executed throughout the system.

### 3.3.1 Operations Supported by System-SPF

First, we must determine which operations in a Trusted Operating System should be supported in System-SPF. Table 3.7 indicates the operations we feel should be supported. Notice that this table is identical to Table 3.6. This is because System-SPF is a global to the kernel and the entire system. Therefore, any operation that can be supported in the SPF framework must be supportable by System-SPF.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Directory</td>
<td>Open File</td>
<td>Setup Connection</td>
<td>Read/Receive from IPC Object</td>
</tr>
<tr>
<td>List Directory Contents</td>
<td>Read File</td>
<td>Configure Network Connection</td>
<td>Write/Send to IPC Object</td>
</tr>
<tr>
<td></td>
<td>Write File</td>
<td>Send Message</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Copy File</td>
<td>Receive Message</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Seek File</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Get File Attributes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.7: Operations supported by System-SPF.

### 3.3.2 System-SPF Architecture and Design

The architectural design of System-SPF is illustrated in Figure 3.2. The System-SPF Configuration indicates which system calls, if any, should skip security checks. When an application makes a call to the operating system, through the system call interface, the kernel should consult the System-SPF Configuration. The System-SPF Configuration indicates if security checks should be skipped in system call X. If the System-SPF Configuration indicates that security should be checked, then the system call should continue its execution by doing the appropriate security checks in the kernel security layer. After the security check, assuming permission is granted, the system call can continue its execution in the rest of the kernel. If the
System-SPF Configuration indicates that security should be skipped, then the system call may bypass the kernel security layer and continue execution in the rest of the kernel. As can be seen, the System-SPF Configuration information is stored in a central location. The diagram suggests that this central location is in the Trusted Operating System kernel, but this is not a requirement. However, the configuration must be stored in a location that the kernel can access and retrieve SPF Configuration information from.

A set of system administrative tools or commands must exist so that system administrators can modify the System-SPF Configuration. The dashed line in Figure 3.2 indicates the path the system administrative tools must take to modify the System-SPF Configuration. As can be seen, the system administrative tools must go through the kernel security layer before it may modify the System-SPF Configuration. We do not want malicious users to modify the System-SPF
Configuration. Therefore, appropriate security checks must be placed in and executed within the system administrative tools.

3.3.3 System-SPF Design Advantages and Disadvantages

As can be seen in Figures 3.3 and 3.5, System-SPF is the only SPF design that has a centrally located SPF Configuration. Therefore, the system administration of System-SPF should be much easier than Process-SPF and Object-SPF. However, having a centrally located SPF Configuration is also a major disadvantage of the System-SPF design. It does not give system administrators the ability to control SPF in individual subsections of the system.

3.4 Process Security Performance Flexibility (Process-SPF)

Process-SPF allows a system administrator to control SPF at the level of a process. A system administrator can select specific applications that should have their security checks disabled in specific set of system calls. For example, consider a system that hosts a database server and a Video On Demand server. The system administrator can specify that the database server is only allowed to skip security checks related to seeking files, while the Video On Demand server is only allowed to skip security checks related to reading files. Other applications will be subject to all the security in the operating system.

3.4.1 Operations Supported by Process-SPF

Table 3.8 below lists the operations that we feel should be supported in Process-SPF. Notice that this table has most of the operations listed in Table 3.7 for System-SPF. The only operations that have been removed are changing directories, listing directory contents, and copying files. Process-SPF cannot offer performance improvements for these operations because they are not typical of processes and applications. Directory operations and copying files are typical of system users and general system activity. For example, it is common for users to log into a shared server, move around between directories, read files, and make copies of files. Changing
directories, listing files in a directory, and copying files is not typical of application layer programs.

<table>
<thead>
<tr>
<th>File Operations</th>
<th>Network Operations</th>
<th>IPC Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open File</td>
<td>Setup Connection</td>
<td>Read/Receive from IPC Object</td>
</tr>
<tr>
<td>Read File</td>
<td>Configure Network Connection</td>
<td>Write/Send to IPC Object</td>
</tr>
<tr>
<td>Write File</td>
<td>Send Message</td>
<td></td>
</tr>
<tr>
<td>Seek File</td>
<td>Receive Message</td>
<td></td>
</tr>
<tr>
<td>Get File Attributes</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.8: Operations supported by Process-SPF.

3.4.2 Process-SPF Architecture and Design

The architectural design of Process-SPF is shown in Figure 3.3. There are a number of additional objects shown in this figure that were not shown in previous figures. In user land, we now show a set of processes, labeled A through C, that are executing in the system. The processes may be independently executing or associated as parent and child. The Process-SPF design is independent of the relationship between different processes. In hardware, we now show a disk and a set of executable programs, labeled M through L. The executable programs on disk may or may not be the executable that created processes A, B, or C. The Process-SPF design is independent of this information.

The Process-SPF Configuration indicates what system call security checks, if any, a particular process is allowed to skip. As the diagram indicates, the Process-SPF Configuration is not centrally located, like it is with System-SPF. The Process-SPF Configuration is stored in two different locations. First, a Process-SPF Configuration is associated with each process running in the system. These are indicated in the diagram with Process-SPF Configurations A through C. Second, a Process-SPF Configuration is associated with each executable file on disk. These are indicated in the diagram as Process-SPF Configurations M through L. As noted in the figure, each of the Process-SPF Configurations should be a specific Process-SPF Configuration for that
executable or process. They do not necessarily have to be the same as other Process-SPF Configurations.

The reason Process-SPF is stored in two locations is for the following reason. When a program is executed, the operating system loads the executable into main memory and creates a process for that program. At this point, there is no information in the system that indicates what the Process-SPF Configuration for this process should be. The Process-SPF Configuration must be stored on disk and then loaded into the system when the respective executable is run. Figure 3.4 illustrates how the Process-SPF Configuration values should be propagated in the system. After a process has been loaded into main memory, the Process-SPF Configuration should be copied from disk into the process’ Process-SPF Configuration in main memory. In addition to propagating the Process-SPF Configuration from disk to the process, the Process-SPF Configuration must also be copied into a child process’ Process-SPF Configuration. This is also illustrated in Figure 3.4.

Similarly to System-SPF, applications make calls to the operating system through the system call interface. After an application has made a call to the operating system, the kernel consults the Process-SPF Configuration for process Y. The configuration indicates if this particular process is allowed to skip the security check for system call X. If the process is allowed to skip the security check, it continues execution in the rest of the kernel; otherwise it continues execution in the kernel security layer first. It is imperative that the kernel consults the Process-SPF Configuration of the calling process Y. As was mentioned before, each process in the system should have its own specific Process-SPF Configuration. Therefore, the respective Process-SPF Configuration must be consulted in order for the system to make the correct decision about skipping a kernel security check.

Figure 3.3 may suggest that a process’ Process-SPF Configuration is passed from the application layer into the kernel layer. However, this is not necessarily required. All that is required is that the kernel have some means of accessing process Y’s Process-SPF Configuration.
Figure 3.3: Process-SPF Architecture.
Finally, Figure 3.3 illustrates that the system administrative tools must perform two important tasks. The system administrative tools must give system administrators the ability to modify the Process-SPF Configurations associated with executable files and currently executing processes. Giving system administrators the ability to modify Process-SPF Configurations, on currently executing processes, allow SPF Configurations to be modified dynamically. Again, appropriate security must be placed within these tools, so that unauthorized individuals cannot change the Process-SPF Configurations in the system. The dashed lines in Figure 3.3 represent the path the system administrative tools must go through in order to modify any Process-SPF Configuration. The system administrative tools must go through appropriate kernel security checks before any modifications to Process-SPF Configurations are allowed.

A point should be noted about Figure 3.3 and 3.4. The figures suggest that the Process-SPF Configuration must actually be stored inside the executable file and process structures in an operating system. This is not a requirement. The actual implementation may or may not store the Process-SPF Configuration within those structures.
3.4.3 Process-SPF Design Advantages and Disadvantages

Process-SPF has an important advantage over the design of System-SPF. Process-SPF gives system administrators far more fine-grained control than with System-SPF. Within the Process-SPF framework, the system administrator can decide exactly what security checks can be skipped for specific applications and processes. The disadvantage of this increased control is the increased system administration complexity for Process-SPF. In addition, because the Process-SPF Configuration is not centrally located like in System-SPF, Process-SPF will require more disk space and memory to operate.

3.5 Object Security Performance Flexibility (Object-SPF)

Object-SPF is the final level in which we will discuss the implementation the SPF framework. Instead of SPF Configurations at a global level or a process level, they will now be associated with individual objects in the operating system. A system administrator can use the Object-SPF framework to specify configuration information for specific objects in the operating system. As an example, consider a database server that is managing several independent databases. The Object-SPF framework should allow system administrators to disable read security checks on some databases but keep read security checks on other databases. This is not possible under System-SPF or Process-SPF because the database server has a single System-SPF or Process-SPF Configuration associated with it. The System-SPF or Process-SPF Configuration can only indicate that read security checks are disabled or enabled for all databases it accesses.

As was mentioned earlier, this thesis will only consider file, network, and IPC objects for the Object-SPF design. However, the design of Object-SPF can easily be extended to other object types in a Trusted Operating System. For example, each of the individual IPC object types (pipes, semaphores, message queues, shared memory, etc.) may be considered separately than in our design. The type of objects that can be designed and implemented within the Object-SPF framework is dependent on how the operating system is viewed at.
For the remainder of this section, we will use the term Object-SPF sparingly. Instead, we will use the terms File-SPF, Network-SPF, and IPC-SPF when describing the design of Object-SPF for files, network connections, and IPC.

### 3.5.1 Operations Supported by Object-SPF

Tables 3.9 through 3.11 list the operations that File-SPF, Network-SPF, and IPC-SPF that should be supported in Object-SPF.

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open File</td>
</tr>
<tr>
<td>Read File</td>
</tr>
<tr>
<td>Write File</td>
</tr>
<tr>
<td>Seek File</td>
</tr>
<tr>
<td>Get File Attributes</td>
</tr>
</tbody>
</table>

Table 3.9: Operations supported by File-SPF.

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup Connection</td>
</tr>
<tr>
<td>Configure Network Connection</td>
</tr>
<tr>
<td>Send Message</td>
</tr>
<tr>
<td>Receive Message</td>
</tr>
</tbody>
</table>

Table 3.10: Operations supported by Network-SPF.

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read/Receive from IPC Object</td>
</tr>
<tr>
<td>Write/Send to IPC Object</td>
</tr>
</tbody>
</table>

Table 3.11: Operations supported by IPC-SPF.

### 3.5.2 Object-SPF Architecture and Design

The architecture and design of File-SPF, Network-SPF, and IPC-SPF, are illustrated in Figure 3.5. There are several new objects shown in Figure 3.5 that have not been shown before. Like Figure 3.3, there is a disk located in the hardware layer of the diagram. On this disk, we show two files, labeled A and B. There are also two network cards in the hardware to indicate that there is networking on this system. Two network cards are illustrated in this figure because
many high-end computing systems have multiple network cards. In the kernel layer, we have added figures to represent active network connections and IPC objects. The IPC objects are labeled F and G, and the network connections are labeled M and N. There are no specific requirements for the objects in Figure 3.5. The files may be executable files, video files, text files, etc. The network connections may be TCP, UDP, etc. The IPC may be shared memory, semaphores, message queues, etc. The design is independent of the specific details of each object.

As noted in the figure, File-SPF, Network-SPF, and IPC-SPF Configurations are associated with each type of object. The File-SPF, Network-SPF, and IPC-SPF Configurations are specific to each individual object and do not necessarily have to be the same as others. The configurations should indicate the system calls, if any, that can skip security checks on the specified object. Again, the figure may suggest that the configuration information must be stored inside the object structure. This is not necessarily required. The only requirement is that separate configuration data exist for each object and the data for that object is accessible by the kernel.

The figure conceptually separates the system call interface into file system, network connection, and IPC categories. This is because each of those categories will access SPF Configuration from different types of objects. When an application calls the operating system through the system call interface, the kernel should consult the configuration information for the respective object. File system calls check the File-SPF Configuration of the requested file, Network system calls check the Network-SPF Configuration of the requested connection, and IPC system calls check the IPC-SPF Configuration of the requested IPC object. The configuration information indicates whether or not the respective system call can skip kernel security checks for that specific object. For example, suppose a system call wants to receive a message from network connection M. The kernel consults the Network-SPF Configuration for that connection. It determines if the system call is allowed to skip security checks for network connection M. Just like with System-SPF and Process-SPF, the system call should continue on with the rest of the kernel if the configuration indicates it should skip the security check. Otherwise, it first continues on through the kernel security layer.
Finally, the system administrative tools need to give system administrators the ability to update or change the SPF Configuration in all objects. Again, security must be placed into the system administrative commands so that unauthorized individuals cannot modify SPF Configurations in the system. The dashed lines in Figure 3.5 show that the path system administrative commands take in the system must go through kernel security checks.

### 3.5.3 Object-SPF Design Advantages and Disadvantages

File-SPF, Network-SPF, and IPC-SPF give the system administrator a much more fine-grained ability to control SPF in the system. Unfortunately, the system administration of these objects will be very complex. It will probably be more complex than Process-SPF. The number of files, network connections, and IPC objects are likely to be greater than the number of programs and processes that run in a system. In addition, just like to Process-SPF, the storage requirements of the configurations in File-SPF, Network-SPF, and IPC-SPF may take up a significant amount of disk space or memory.

### 3.6 Analysis of System-SPF, Process-SPF, and Object-SPF

The advantages and disadvantages between System-SPF, Process-SPF, and Object-SPF are really a balance between system administrative complexity and system administrative control. System-SPF has the least control over a system’s configuration, but it is the easiest to maintain. Object-SPF offers the most control, but will probably be the most difficult to manage. Process-SPF seems to offer a nice balance between System-SPF and Object-SPF. Both Process-SPF and Object-SPF may require more disk space or memory to operate than System-SPF.
Figure 3.5: Object-SPF Architecture.
CHAPTER 4

Implementation

This chapter will discuss the design and implementation of SPF in SELinux. As has been mentioned before, this thesis will only study SPF and how it can improve file system performance and the applications that use file systems heavily. Thus, we will only implement the System-SPF, Process-SPF, and File-SPF designs from Chapter 3. Nothing will be implemented that supports SPF for networking or IPC.

We will first describe the development environment in Section 4.1. Then, we will give an overview of SELinux security implementation in Section 4.2. The implementation of System-SPF, Process-SPF, and File-SPF will be discussed in Sections 4.3, 4.4, and 4.5 respectively. Sections 4.3, 4.4, and 4.5 will also discuss the advantages, disadvantages, system administration issues, and potential security issues that we discovered while implementing SPF in each operating system level. Section 4.6 will conclude this chapter by discussing the advantages and disadvantages between System-SPF, Process-SPF, and File-SPF. Performance analysis between the three different implementations will be covered in Chapter 5.

This section assumes the reader has a fairly good knowledge of Linux operating systems design, architecture, and system call interface. For more information about the Linux operating system see [SILBERSCHATZ98] or [TANENBAUM01].

4.1 Development Environment

The development and test machine used in this thesis is a Pentium based PC. Therefore, all hardware specific modifications to the Linux kernel will be specific to PC based architectures. Details of the machine used in this thesis are discussed in Section 5.2.
4.2 SELinux File System Security Design and Implementation

This section will give a simplified overview of the design and implementation of file system security in SELinux. The details of the file system security and all other SELinux security can be found in [LOSCOCC001B].

For the purposes of this thesis, file system security can be divided into two different categories. There is security relevant to directories and security relevant to individual files. Tables 4.1 and 4.2 give an idea of what security relevant operations must be considered under each of these categories. Tables 4.1 and 4.2 also list the Linux system calls that are associated with these operations. SELinux has placed additional security checks within all of these system calls. These tables should not be considered thorough or complete. They are shown here to give an indication of the design of file system security in SELinux.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Relevant System Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make Directory</td>
<td>mkdir()</td>
</tr>
<tr>
<td>Remove Directory</td>
<td>rmdir()</td>
</tr>
<tr>
<td>Change Directory</td>
<td>chdir(), fchdir()</td>
</tr>
<tr>
<td>List Directory Contents</td>
<td>getdents()</td>
</tr>
</tbody>
</table>

Table 4.1: Security relevant directory operations and relevant system calls.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Relevant System Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create File</td>
<td>creat(), open()</td>
</tr>
<tr>
<td>Delete File</td>
<td>unlink()</td>
</tr>
<tr>
<td>Open File</td>
<td>open()</td>
</tr>
<tr>
<td>Read File</td>
<td>read(), readv(), pread()</td>
</tr>
<tr>
<td>Write File</td>
<td>write(), writev(), pwrite()</td>
</tr>
<tr>
<td>Copy File</td>
<td>sendfile()</td>
</tr>
<tr>
<td>Seek File</td>
<td>seek(), lseek()</td>
</tr>
<tr>
<td>Get File Attributes</td>
<td>stat(), fstat(), lstat()</td>
</tr>
<tr>
<td>Set File Attributes</td>
<td>chmod(), chown(), fchown(), lchown()</td>
</tr>
</tbody>
</table>

Table 4.2: Security relevant file operations and relevant system calls.

SELinux implements security through an enhancement of the standard Linux kernel. The majority SELinux security has been implemented in a set of hook functions located in a Linux
kernel module. These functions implement and enforce the type enforcement, RBAC, and MLS security of SELinux. These hook functions are called from important security checkpoints in Linux system calls and kernel functions. The hook functions check the SELinux security policy to see if the calling function is allowed to proceed into the remainder of the kernel. After a decision has been made in the security module, information is returned to the calling function, telling it whether or not kernel execution can continue. If permissions are granted, the system call or kernel function can continue executing. If permission is denied, an error is returned to the application.

```
asmlinkage ssize_t sys_read(unsigned int fd, char * buf, size_t count)
    ssize_t ret;
    struct file * file;
    ret = -EBADF;
    file = fget(fd);
    if (file) {
        if (file->f_mode & FMODE_READ) {
            ret = locks_verify_area(FLOCK_VERIFY_READ,
                file->f_dentry->d_inode,
                file, file->f_pos, count);
            if (!ret) {
                ssize_t (*read)(struct file *, char *, size_t, loff_t *);
                ret = -EINVAL;
                if (file->f_op && (read = file->f_op->read) != NULL) {
                    ret = security_ops->file_ops->permission (file,
                        MAY_READ);
                    if (!ret)
                        ret = read(file, buf, count, &file->f_pos);
                }
            }
        }
        if (ret > 0)
            dnotify_parent(file->f_dentry, DN_ACCESS);
        putfile(file);
    }
    return ret;
```

Figure 4.1: SELinux modification of the `read()` system call.

As an example, Figure 4.1 shows how SELinux modified the `read()` system call in Linux. The bolded lines indicate the additional security hook function and code that SELinux has added to the `read()` system call. The `read()` system call is modified to call the file permissions hook function `security_ops->file_ops->permission()`. If the returned value, `ret`, is equal to 0, the read
continues by calling the kernel `read()` routine. Otherwise, the kernel `read()` routine is skipped and an error is returned from the `read()` system call. Other security relevant system calls have been modified similarly to this.

This thesis will not discuss the details or implementation of the SELinux hook functions. It is not necessary for understanding the implementation of SPF into SELinux. Readers who are interested can see [SMALLEY02] for details.

4.3 System Wide Security Performance Flexibility (System-SPF)

4.3.1 System-SPF Implementation

Table 4.3 lists the operations from the System-SPF design in Section 3.3 that are related to file systems. The table also lists the system calls associated with these operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Relevant System Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Directory</td>
<td><code>chdir()</code>, <code>fchdir()</code></td>
</tr>
<tr>
<td>List Directory Contents</td>
<td><code>getdents()</code></td>
</tr>
<tr>
<td>Open File</td>
<td><code>open()</code></td>
</tr>
<tr>
<td>Read File</td>
<td><code>read()</code>, <code>readv()</code>, <code>pread()</code></td>
</tr>
<tr>
<td>Write File</td>
<td><code>write()</code>, <code>writev()</code>, <code>pwrite()</code></td>
</tr>
<tr>
<td>Get File Attributes</td>
<td><code>stat()</code>, <code>fstat()</code>, <code>lstat()</code></td>
</tr>
<tr>
<td>Copy File</td>
<td><code>sendfile()</code></td>
</tr>
<tr>
<td>Seek File</td>
<td><code>seek()</code>, <code>lseek()</code></td>
</tr>
</tbody>
</table>

Table 4.3: Operations supported by System-SPF and relevant system calls.

First, we determine how security checks will be skipped in the kernel. Based on how they will be skipped, we can determine how to implement the System-SPF Configuration in the system. As was mentioned in Section 4.2, the majority of SELinux security has been implemented in a set of hook functions. These hook functions are called by security relevant system calls and kernel functions. Therefore, skipping the security checks simplifies to surrounding the SELinux hook functions with `if` statements. A flag variable can indicate if the SELinux hook function should be skipped or not. Since System-SPF will be applied system wide in the SELinux kernel, we can implement the flags as global variables in the SELinux kernel. The global kernel
variables are easily accessible by all system calls, thus making it simple for system calls to consult the System-SPF Configuration.

However, after consideration, it seems inefficient to implement a flag variable for each operation that is relevant to System-SPF. The number of operations that are security relevant is far less than the number of bits in an integer variable (32 bits). Therefore, we can reduce the number of global kernel variables to just one. Each bit of this global variable can represent one of the operations in Table 4.3. This global variable the will be our System-SPF Configuration. A system call can determine if it should skip a SELinux security check, by looking at the variable and seeing if its respective operation bit is set or not. If the bit is set, the SELinux hook function should not be called. If the bit is not set, the SELinux hook function should still be called.

A decision must be made about how this kernel variable can be modified by a system administrator. If we make this global kernel variable a system tunable (also known as a kernel parameter), it can be modified by the `sysctl` command. The `sysctl` command is a command that allows system administrators to modify system tunables at run time.

In order to implement a new system tunable in the SELinux kernel, the system tunable has to be declared in the system tunables table and system tunables enumeration. Since we are implementing SPF for file systems, an integer system tunable has been placed into the `fs_table` structure in `kernel/sysctl.c`. The system tunable is called `fs.spf`. The constant `FS_SPF` has been added to the file system tunable enumeration in `include/linux/sysctl.h`.

Even though eight operations are listed above in Table 4.3, only six bits of `fs.spf` are actually necessary to implement System-SPF. Copying files can be broken down into read operations and write operations. Therefore, an additional bit for copying is not necessary. Changing directories and listing directory contents are essentially the same, because listing directory contents usually requires changing to that directory first. Therefore, these two operations can be combined into one. Figure 4.2 shows the System-SPF code that indicates which bits of `fs.spf` correlate to which operations. The `SPF_CHDIR` mask refers to both changing directories and
listing directory contents. SPF_STAT refers to getting file attributes, since the \textit{stat()} system call is used to obtain file attributes.


\begin{verbatim}
#define SPF_NONE       (0x0)
#define SPF_READ       (0x1)
#define SPF_WRITE      (0x1 << 1)
#define SPF_OPEN       (0x1 << 2)
#define SPF_STAT       (0x1 << 3)
#define SPF_CHDIR      (0x1 << 4)
#define SPFSEEK        (0x1 << 5)
\end{verbatim}

Figure 4.2: System-SPF bit masks.

In order to turn off security checks for a particular set of operations, a system administrator needs to configure the system tunable \textit{fs.spf}. The \textit{fs.spf} value should be set to the numeric value that results from a bitwise OR of the constants, in Figure 4.2, that correlate to the security that should be disabled. For example, to turn off security checks for opening files, reading files, and getting file attributes (which are the common file system operations performed in a web server), the \textit{fs.spf} variable should be set to 13. We get the value of 13 because:

\[
\begin{align*}
\text{SPF\textunderscore OPEN} & = (0x1 \ll 2) = 2^2 = 4 \\
\text{SPF\textunderscore READ} & = (0x1) = 2^0 = 1 \\
\text{SPF\textunderscore STAT} & = (0x1 \ll 3) = 2^3 = 8 \\
4 + 1 + 8 & = 13
\end{align*}
\]

Therefore, we can disable security checks for opening files, reading files, and getting file attributes by running

\texttt{sysctl -w fs.spf=13}

at the command line. This can only be run as root.

Once the System-SPF Configuration in \textit{fs.spf} has been set, we can program security checks to be skipped in the kernel. First, we must determine the functions that call the SELinux hook
functions. These functions are listed in Table 4.4. Within these functions, the code must check the \textit{fs.spf} system tunable to determine if the SELinux hook function should be called or not. We can determine if the SELinux hook function should be called by determining if the respective operation bit in \textit{fs.spf} is set or not. The respective \textit{fs.spf} bit can be analyzed using the bit masks in Figure 4.2. For example, we can see from Table 4.4 that \textit{sys\_write()} is one of the functions that calls a SELinux hook function for the write operation. In order to determine if a write security check should be skipped, we check the second bit of \textit{fs.spf} (because \[ \text{SPF\_WRITE} = (0x1 \ll 1) = 2 \]) to see if it is set or not. If it is set, the write security check should be skipped. If the bit is not set, the SELinux hook function for write security should still be called.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Functions Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Directory</td>
<td>\textit{sys_chdir(), sys_fchdir(), permission()}</td>
</tr>
<tr>
<td>Search Directory</td>
<td>\textit{vfs_readdir()}</td>
</tr>
<tr>
<td>Open File</td>
<td>\textit{open_name(), permission()}</td>
</tr>
<tr>
<td>Read File</td>
<td>\textit{sys_sendfile(), sys_read(), sys_readv(), sys_pread()}</td>
</tr>
<tr>
<td>Write File</td>
<td>\textit{sys_sendfile(), sys_write(), sys_writev(), sys_pwrite()}</td>
</tr>
<tr>
<td>Get File Attributes</td>
<td>\textit{cp_old_stat()}</td>
</tr>
<tr>
<td>Seek File</td>
<td>\textit{sys_lseek(), sys_llseek()}}</td>
</tr>
</tbody>
</table>

Table 4.4: Functions modified to implement System-SPF.

Figure 4.3 shows how the \textit{read()} system call has been modified to implement System-SPF. The bolded lines indicate the modifications made compared to the original SELinux code in Figure 4.1. The \textit{fs.spf} system tunable (which is \textit{spf} in the kernel) is checked with the \textit{SPF\_READ} mask to determine if the read bit is set or not. If it is set, and the file is a regular file, the \textit{read} routine is called without first calling the SELinux hook function. If the bit is not set, the SELinux hook function is called before the \textit{read} routine is called. The modifications made in Figure 4.3 are similar to the modifications made in other system calls.

\footnote{Since the \textit{read()} system call is also used for other operations, such as network operations, the file descriptor passed into the system call must be checked to ensure that is a regular file. \textit{S\_ISREG} is used for this check.}
One final consideration has been made about the open() system call. In our implementation, we decided that implementing System-SPF, such that security checks can be skipped for opening files, reading files, and writing files, is far too dangerous to implement. Therefore, we deviate from System-SPF design of Section 3.3 for the open() system call. The open() system call only allows SELinux security checks to be skipped if the file is opened as read only. This is the situation that will most likely occur on workloads such as web servers, database servers, and video on demand servers. Luckily, the open() system call interface always requires that a user to indicate if a file is read only, write only, or both [STEVENS93]. Therefore, it is easy to determine if the file passed to the open() system call is read only.

Figure 4.3: System-SPF modification of the read() system call.
4.3.2 System-SPF Implementation Advantages and Disadvantages

System-SPF has one distinct security advantage over Process-SPF and File-SPF. Unlike Process-SPF and File-SPF, System-SPF can be hard coded into the Linux kernel instead of being implemented as a system tunable. Rather than making the global kernel variable modifiable by a system administrator, we can define a global constant in the kernel. A system administrator can set each of the bits in the global constant and compile the kernel. This will hard code whether or not security checks will be skipped or not skipped in the kernel. Compiling System-SPF into a kernel takes away the ability for a system administrator to change the System-SPF Configuration during a system’s execution. However, it can make the system more secure overall. By implementing System-SPF in this fashion, a malicious attacker cannot reconfigure the System-SPF Configuration.

4.3.3 System-SPF System Administration

As will be seen in later sections, System-SPF is the simplest of the SPF implementations. This makes the system administration of System-SPF the easiest. This is what we expected based on the design of System-SPF in Section 3.3.

4.3.4 System-SPF Security Issues

The implementation of System-SPF does not seem to have any major security holes. Allowing System-SPF to turn off security checks for open(), read(), and write() is the only part of the original design that seemed too dangerous to implement. That is why it has been modified so that the open() system call can only skip security checks for read only files. Disabling System-SPF for the write() system call may make the system safer overall. This option should be considered for future implementations of System-SPF.

The only other security concern is that a malicious user can take advantage of System-SPF and modify the fs.spf system tunable. By modifying the fs.spf system tunable, a malicious user can turn off security checks for a variety of system calls. However, the modification of a system
tunable is already protected by secure trusted mechanisms in SELinux. Therefore, no major security holes seem to exist with System-SPF, unless as security hole already exists in SELinux.

4.4 Process Security Performance Flexibility (Process-SPF)

4.4.1 Process-SPF Implementation

Table 4.5 lists the operations from the Process-SPF design, in Section 3.4, that are related to file systems. The relevant system calls for these operations are also listed in Table 4.5.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Relevant System Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open File</td>
<td>open()</td>
</tr>
<tr>
<td>Read File</td>
<td>read(), readv(), pread()</td>
</tr>
<tr>
<td>Write File</td>
<td>write(), writev(), pwrite()</td>
</tr>
<tr>
<td>Get File Attributes</td>
<td>stat(), fstat(), lstat()</td>
</tr>
<tr>
<td>Seek File</td>
<td>seek(), lseek()</td>
</tr>
</tbody>
</table>

Table 4.5: Operations supported by Process-SPF and relevant system calls.

There are many parts to the implementation of Process-SPF that are similar to the implementation of System-SPF. Therefore, we will begin by explaining the parts of the Process-SPF implementation that are very similar to the System-SPF implementation. Then we will move into the Process-SPF implementation specifics.

For Process-SPF, SELinux hook functions are surrounded by if statements. Process-SPF Configuration values will be represented by an integer. Each bit of this Process-SPF Configuration represents one of the operations in Table 4.5. The if statements will check the Process-SPF Configuration, of the currently executing process, to determine if the SELinux hook function should be called or not. A particular system call should skip a security check, for the currently executing process, if the respective operation bit in the Process-SPF Configuration is set. The respective operation bit for each modified function can be checked using bit masks. The bit masks used for Process-SPF are shown in Figure 4.4. The functions modified to implement Process-SPF are listed in Table 4.6.
Now moving into Process-SPF specific issues, the design of Process-SPF in Section 3.4 indicates there are two places where Process-SPF Configuration values can be stored. They are within executable files and within currently running processes.

The first implementation issue that we will solve is how to store Process-SPF Configurations for each executable file on disk. The implementation is nearly identical to the implementation of File-SPF Configurations on disk. Therefore, discussion of this implementation will not be discussed. The details of this implementation can be found in Section 4.5.1 of the File-SPF implementation.

The next implementation issue that we will look at is how to store the Process-SPF Configurations for each executing process and how Process-SPF Configurations on disk can be loaded into this process. The task_struct structure in Linux holds the primary information about a running process, such as process ID, process priority, process owner, and process group. This is the natural place to store the configuration data for Process-SPF. Therefore, a variable named spf has been added into the task_struct structure located in include/linux/sched.h. This variable holds Process-SPF Configuration data for its process.

Next, we must determine how the Process-SPF Configuration stored on disk can be loaded into the task_struct of a new executing process. When a binary is executed with the execve() system call, binary information, such as command line arguments, are stored into the linux_binprm structure defined in include/linux/binfmts.h. The linux_binprm structure is passed to the appropriate binary loader for the executable. For most Linux binaries, the binary loader will be the load_elf_binary() function in fs/binfmt_elf.c or the load_aout_binary() function in

```
#define SPF_NONE       (0x0)
#define SPF_READ       (0x1)
#define SPF_WRITE      (0x1 << 1)
#define SPF_OPEN       (0x1 << 2)
#define SPF_STAT       (0x1 << 3)
#define SPFSEEK        (0x1 << 4)
```

Figure 4.4: Process-SPF bit masks.
fs/binfmt_aout.c. Within these functions, a binary is loaded into a system process and the elements of the process’ task_struct structure are initialized. Some of the values in the new process’ task_struct are copied from data passed in from the linux_binprm structure.

Therefore, we can load Process-SPF Configuration from disk, into the task_struct of a new process, by first copying the Process-SPF Configuration from an inode into the linux_binprm structure. Then we can copy the Process-SPF Configuration from the linux_binprm structure into the task_struct structure. In order to implement this, a variable named spf has been added to the linux_binprm structure. This variable will carry the Process-SPF Configuration from disk into the appropriate binary loader. Once inside the binary loader functions, the spf value from the linux_binprm structure is copied into the spf variable in the task_struct structure. This process effectively loads the Process-SPF Configuration from disk into the new executing process. This process is illustrated in Figure 4.5.

In addition, the Process-SPF design from Section 3.4 indicates that we must propagate Process-SPF Configurations from parent processes to child processes, as illustrated in Figure 3.4. Therefore, a small modification has been made to the fork() system call to copy the Process-SPF Configuration value from a parent to a child process.

Now that Process-SPF Configuration values are properly stored in each executing process, the only implementation issue that remains is how to retrieve the value of the currently executing process’ Process-SPF Configuration value. In order to access the currently executing process’ task_struct, we use the global current pointer. The current pointer in Linux points to the task_struct structure of the currently executing process. Therefore, it is easy to read the value of the spf variable in the currently executing process. Once it has been read, the system call can determine if the SELinux security check should be skipped or not.

Figure 4.6 shows how the read() system call has been modified to support Process-SPF. The bolded lines indicate the lines of code that have been added compared to Figure 4.1. The changes made to the system call are very similar to the changes made for the same system call in System-SPF. However, this time, the Process-SPF Configuration values are retrieved using the
current pointer. The value of the Process-SPF Configuration is retrieved through the statement current->spf. Changes to other functions in the kernel have been modified similarly. The functions modified to implement Process-SPF are listed in Table 4.6. Like System-SPF, we have decided to skip security checks in the open() system call only for those files that are read only.

![Diagram of Process-SPF Configuration](image.png)

**Figure 4.5**: Loading Process-SPF Configuration.
Finally, we will discuss the implementation of the system administrative commands and the system calls that support them. According to the Process-SPF design from Section 3.4, there are

---

```
#include <fcntl.h>

asmlinkage ssize_t sys_read(unsigned int fd, char * buf, size_t count) {
    ssize_t ret;
    struct file * file;

    ret = -EBADF;
    file = fget(fd);
    if (file) {
        if (file->f_mode & FMODE_READ) {
            ret = locks_verify_area(FLOCK_VERIFY_READ,
                    file->f_dentry->d_inode,
                    file, file->f_pos, count);
            if (!ret) {
                ssize_t (*read)(struct file *, char *, size_t, loff_t *);
                ret = -EINVAL;
                if (file->f_op && (read = file->f_op->read) != NULL) {
                    if (((current->spf & SPF_READ) > 0) &&
                        S_ISREG(file->f_dentry->d_inode->i_mode)) {
                        ret = read(file, buf, count, &file->f_pos);
                    } else {
                        ret = security_ops->file_ops->permission(file,
                                MAY_READ);
                        if (!ret)
                            ret = read(file, buf, count, &file->f_pos);
                    }
                } else {
                    if (ret > 0)
                        dnotify_parent(file->f_dentry, DN_ACCESS);
                        fput(file);
                }
            return ret;
        }
    }
    return ret;
}
```

---

Figure 4.6: Process-SPF modification of the `read()` system call.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Functions Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open File</td>
<td><code>open_namei()</code>, <code>permission()</code></td>
</tr>
<tr>
<td>Read File</td>
<td><code>sys_read()</code>, <code>sys_readv()</code>, <code>sys_pread()</code></td>
</tr>
<tr>
<td>Write File</td>
<td><code>sys_write()</code>, <code>sys_writev()</code>, <code>sys_pwrite()</code></td>
</tr>
<tr>
<td>Get File Attributes</td>
<td><code>cp_old_stat()</code></td>
</tr>
<tr>
<td>Seek File</td>
<td><code>sys_lseek()</code>, <code>sys_llseek()</code></td>
</tr>
</tbody>
</table>

Table 4.6: Functions modified to implement Process-SPF.
two different sets of system administrative tools that must be developed, because there are two ways Process-SPF Configuration can be set in the system.

The commands created to support system administrative actions are named \texttt{set\_bin\_spf}, \texttt{get\_bin\_spf}, \texttt{set\_proc\_spf}, and \texttt{get\_proc\_spf}. In the current implementation, only root is allowed to execute these commands. The \texttt{set\_bin\_spf} command takes a Process-SPF Configuration value and the name of an executable binary at the command line. It sets the executable binary’s Process-SPF Configuration to the value indicated. For example, suppose we wanted to disable \texttt{seek()} security checks for all files accessed by a database program. Under Process-SPF, we can set this by executing:

\texttt{set\_bin\_spf 16 my\_database\_program}

at the command line. The \texttt{get\_bin\_spf} command retrieves the Process-SPF value currently stored on disk for the file name passed in at the command line.

The \texttt{set\_proc\_spf} and \texttt{get\_proc\_spf} commands are similar, except they take process IDs (PIDs) at the command line instead of filenames. Suppose, in our example, that we later decide we want our database program to check security again on \texttt{seek()} system calls. In order to use the \texttt{set\_proc\_spf} command, we first need to determine the PID of our database program. This can be determined using the \texttt{ps} command. Suppose the PID of our program is 1000. To enforce \texttt{seek()} security checks in the program again, we run:

\texttt{set\_proc\_spf 0 1000}

at the command line. The Process-SPF Configuration is set to 0 in this example because we want to unset the \texttt{SPF\_SEEK} bit from the Process-SPF Configuration. The \texttt{get\_proc\_spf} command similarly retrieves the Process-SPF Configuration value for the PID passed in at the command line.
Figure 4.7 illustrates the use of these two different system administrative commands for Process-SPF. Initially, myprogram has its Process-SPF Configuration set to 3 using the set_bin_spf command. When the program is initially loaded, the Process-SPF Configuration is copied to the respective Process-SPF Configuration for the process, just like in Figure 3.4. Later, the Process-SPF Configuration value is changed from 3 to 5 using the set_proc_spf command. Notice that the Process-SPF Configuration stays at 3 for the executable, even though the Process-SPF Configuration is 5 for the process myprogram. Our implementation does not propagate changes made by set_proc_spf back to the original executable.

![Diagram of Process-SPF commands](image)

(1) Set Process-SPF Configuration on executable:

```
set_bin_spf 3 myprogram
```

(2) Process-SPF Configuration in myprogram set to 3.

(3) Execute myprogram.


(5) Change Process-SPF Configuration for this process from 3 to 5.

```
set_proc_spf 5 2000.
```

In order to support these commands, system calls were developed. The system calls are called set_bin_spf(), get_bin_spf(), set_proc_spf(), and get_proc_spf(). In order to implement these system calls, we added them to the include/asm-i386/unistd.h file and arch/i386/kernel/entry.S file.

The set_bin_spf() and get_bin_spf() system calls are implemented nearly identically to the set_file_spf() and get_file_spf() system calls for File-SPF. Details of the implementation are left
to Section 4.5.1. The interface for set_bin_spf() and get_bin_spf() is shown in Figure 4.8. Table 4.7 lists the functions that have been modified along with set_bin_spf() and get_bin_spf() to support the storage and retrieval of Process-SPF Configurations on disk.

The set_proc_spf() and get_proc_spf() system calls are very similar to the setpriority() and getpriority() systems calls, since both sets of system calls deal with a variable stored within a process’ respective task_struct structure. The set_proc_spf() and get_proc_spf() system calls loop through the set of task_struct structures in the system until the requested process’ task_struct is found. Then set_proc_spf() and get_proc_spf() set or get the spf variable respectively. Since these system calls are so similar to the setpriority() and getpriority() system calls, they are given the same security as those system calls. The system call interface for set_proc_spf() and get_proc_spf() is shown in Figure 4.8.

```
#include <linux/spf.h>
#include <linux/unistd.h>

_syscall1(int, get_bin_spf, char*, filename);
_syscall2(int, set_bin_spf, char*, filename, int, spf_option);
_syscall1(int, get_proc_spf, pid_t, pid);
_syscall2(int, set_proc_spf, pid_t, pid, int, spf_option);

int get_bin_spf(const char * filename);
    Returns: Process-SPF Configuration value if OK, -1 on error

int set_bin_spf(const char * filename, int spf_option);
    Returns: 0 if OK, -1 on error

int get_proc_spf(pid_t pid);
    Returns: Process-SPF Configuration value if OK, -1 on error

int set_proc_spf(pid_t pid, int spf_option);
    Returns: 0 if OK, -1 on error
```

Figure 4.8: System call interface for Process-SPF.

71
Support Role | Functions Modified
---|---
Retrieving Process-SPF Configurations on Disk | ext2_read_inode(), ext3_read_inode(),
dentry_open()
Storing Process-SPF Configurations on Disk | ext2_write_inode(), ext3_write_inode(),
open_namei()
System Administration of Process-SPF Configurations | inode_setattr(), setattr_mask()
Loading Process-SPF Configurations into a Process | load_elf_binary(), load_aout_binary(),
do_fork(), do_execve()

Table 4.7: Functions modified to support Process-SPF.

4.4.2 Process-SPF Implementation Advantages and Disadvantages

With the exception of the same File-SPF advantages and disadvantages about storing File-SPF Configurations on disk, there seem to be no advantages or disadvantages with the implementation of Process-SPF. The File-SPF advantages and disadvantages are discussed in Section 4.5.2.

4.4.3 Process-SPF System Administration

By giving system administrators the ability to change Process-SPF Configurations for already running processes, it allows Process-SPF Configurations to be modified dynamically. This feature may be very useful for system administrators. Process-SPF Configurations can be changed without having to shutdown a process or service.

The system administration of Process-SPF will be more difficult than System-SPF. With Process-SPF, each individual binary or process must be analyzed and then set with the appropriate Process-SPF Configuration.

4.4.4 Process-SPF Security Issues

There is one major security issue in Process-SPF. With the current implementation, every forked process is initialized to the same Process-SPF Configuration as their parent. This may make Process-SPF dangerous for programs that have modifiable code. Malicious actions can occur
within a forked process if the code is modifiable and the child process has security disabled for a set of system calls. Technically speaking, the Process-SPF implementation can disable the ability for Process-SPF Configurations to propagate to child processes. The modification made to the `fork()` system call can simply be removed. That way, a child process’ Process-SPF Configuration is never initialized and will be equivalent to `SPF_NONE`.

However, eliminating the propagation of the Process-SPF Configuration to child processes limits the types of applications Process-SPF can help improve the performance of. There are many applications that spawn helper processes during execution. The best examples are web servers, video on demand servers, and database servers. These programs may spawn child processes to handle requests or queries. If we did not allow Process-SPF Configurations to propagate to child processes, these applications would not benefit from Process-SPF. Therefore, propagating Process-SPF Configurations to child processes may depend on the type of environment Process-SPF will be used in. Future development of Process-SPF may elect not to propagate Process-SPF Configurations to child processes if the security of the configuration propagation is a concern.

Finally, there may be security concerns with the `set_bin_spf()`, `get_bin_spf()`, `set_proc_spf()`, and `get_proc_spf()` system calls. Security has been placed in these system calls that is identical to the security for very similar Linux system calls. If this is not an appropriate policy, a new type of security policy can be created just for these system calls.

### 4.5 File Security Performance Flexibility (File-SPF)

#### 4.5.1 File-SPF Implementation

We will begin by briefly explaining the implementation details of File-SPF that are very similar to System-SPF and Process-SPF. Then we will move into the details of File-SPF specific implementation.
Again, for File-SPF, SELinux hook functions will be surrounded by *if* statements. File-SPF Configuration values will be represented by integers. Each bit of the File-SPF Configuration represents one of the operations listed in Table 4.8. Each File-SPF Configuration indicates which SELinux hook functions can by bypassed when a particular file is read from, written to, or sought. Inside each of the security relevant functions in the kernel, we will check if the File-SPF Configuration indicates the security check should be skipped. If the respective operation bit in the File-SPF Configuration is set, the SELinux hook function will not be called. The bit masks used for File-SPF are shown in Figure 4.9. The functions modified to implement File-SPF are listed in Table 4.9.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Relevant System Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read File</td>
<td><em>read(), readv(), pread()</em></td>
</tr>
<tr>
<td>Write File</td>
<td><em>write(), writenv(), pwrite()</em></td>
</tr>
<tr>
<td>Seek File</td>
<td><em>seek(), lseek()</em></td>
</tr>
</tbody>
</table>

Table 4.8: Operations supported by File-SPF and relevant system calls.

```
#define SPF_NONE       (0x0)
#define SPF_READ       (0x1)
#define SPF_WRITE      (0x1 << 1)
#define SPFSEEK        (0x1 << 2)
```

Figure 4.9: File-SPF bit masks.

Note that the list of operations in Table 4.8 is shorter than the File-SPF operations listed in Table 3.9 of the File-SPF design. In particular, we have elected not to support opening files and getting file attributes with File-SPF. The reason for this will be discussed in detail in Section 4.5.2.

As was described in the File-SPF design in Section 3.5, File-SPF Configurations must be implemented in a way that individual filenames can be associated with specific File-SPF Configurations. The easiest way to accomplish this is to store the File-SPF Configuration values inside the individual file structure on disk.
Therefore, we will first show how a new File-SPF Configuration value can be stored on disk and be retrieved by the Linux kernel. The Linux operating system has an inode structure associated with every file on disk. The inode structure stores generic information about the file, such as file size, file owner, and access permissions. Therefore, this seems to be the best place to store the File-SPF information.

Unfortunately, adding a variable to a file’s inode is not as simple as adding a variable to the inode structure of the Linux kernel. There are actually two types of inodes in the Linux operating system. There are the operating system inodes, which are abstract views of files on disk. They are used to let the kernel operate with files more easily. The actual file system has a different inode structure that is used for organizing how the data will be stored on disk and later retrieved off of disk. We will refer to these two different inodes as os_inode and raw_inode respectively.

Initially, we thought we could add an additional File-SPF Configuration variable to both inode structures. It is safe to add a File-SPF Configuration to the os_inode structure. Therefore, a variable named \textit{spf} has been added to the os_inode structure located in \texttt{include/linux/fs.h}. This \textit{spf} variable holds the File-SPF Configuration of the file when it is first opened.

Unfortunately, it is not easy to place a new variable in the raw_inode. These structures are very specific and relate to exactly how data is stored on disk. Adding another variable to the file system’s raw_inode structure is potentially dangerous. Other parts of the file system are dependent on the raw_inode structure. Therefore, any changes to it may propagate errors to other parts of the file system. When the original SELinux developers added security label information to files, they got around the raw_inode problem by creating a separate table in the file system. This table mapped files in the file system to a security label.

We determined there are three different options we can undertake to store the File-SPF Configuration in a file’s disk inode. The first method requires modification of the entire file system, so that a File-SPF Configuration is included in the raw_inode structure. This will be the most difficult approach to undertake. In addition, most file systems have become standard in the
Linux community. The Linux community may not accept any modification to the raw_inode. Therefore, we decided not to implement this first method.

The second method takes advantage of the security label table implemented in SELinux. We can store the File-SPF Configurations into the same table that stores security label information about every file. This may seem like the natural approach, since the security labels have already been implemented to avoid the problems we encounter with the raw_inode. However, we decided not to implement this method because it can degrade performance of File-SPF. As was mentioned earlier in Section 4.2, all SELinux functions have been confined to a set of hook functions in a Linux module. Therefore, if we add the File-SPF Configurations to the security label table, we will need to add additional functions in the SELinux security module to retrieve the File-SPF Configuration data. This means we have to call a function within the Linux module to retrieve the File-SPF Configuration. The File-SPF Configuration will be retrieved from the SELinux security module before we can determine if security checks can be skipped or not. This additional function call may degrade performance. Therefore, we decided not to implement the File-SPF Configurations on disk using this method. However, later, we discovered that implementing File-SPF with this method might not be as bad as we originally thought. The reason is that the overhead of calling the Linux module may not be incurred too often.

The third option for storing the File-SPF Configurations on disk is to somehow include the File-SPF Configuration into the existing file system inode structure. As we mentioned earlier in this section, we have chosen to implement only three file-system operations for File-SPF: read, write, and seek. Therefore, we only need 3 bits inside a file system raw_inode to store the File-SPF Configuration data. Luckily, within both the ext2 and ext3 Linux file systems, there exists an integer inode flag, called i_flag, for each file in the file system. This flag is used for a variety of file system specific purposes. However, only the lower 13 bits and the highest bit of this variable are used. Since integers are 32 bits long, that means 18 bits of this flag are never used. We can use these 18 bits to store the File-SPF Configuration on disk.

This method is not the prettiest implementation. However, it guarantees us that we avoid the additional function call to the SELinux kernel module. Therefore, it is the implementation that
we elect to use for File-SPF in SELinux. If we did not have enough space on the raw-inode to hold the File-SPF Configuration, we may be forced to implement File-SPF using a different method.

Figure 4.10 shows which bits of the ext2 i_flag are used by the system and which bits are used by File-SPF to store the File-SPF Configuration on disk. The bolded lines indicate the additions we made to store the File-SPF Configuration on disk. A nearly identical method is used in the ext3 file system. The ext2 and ext3 file systems are the primary file systems used in Linux, therefore they are the only ones that we support with File-SPF.

```
/*
   * Inode flags
   */
#define EXT2_SECRM_FL 0x00000001
#define EXT2_UNRM_FL 0x00000002
#define EXT2_COMPR_FL 0x00000004
#define EXT2_SYNC_FL 0x00000008
#define EXT2_IMMUTABLE_FL 0x00000010
#define EXT2_APPEND_FL 0x00000020
#define EXT2_NODUMP_FL 0x00000040
#define EXT2_NOATIME_FL 0x00000080
#define EXT2_DIRTY_FL 0x00000100
#define EXT2_COMPBLK_FL 0x00000200
#define EXT2_NOCOMP_FL 0x00000400
#define EXT2_ECOMPR_FL 0x00000800
#define EXT2_BTREE_FL 0x00001000
#define EXT2_RESERVED_FL 0x80000000
#define EXT2_FL_USER_VISIBLE 0x00001FFF
#define EXT2_FL_USER_MODIFIABLE 0x000000FF

#define EXT2_SPF_READ 0x00010000
#define EXT2_SPF_WRITE 0x00020000
#define EXT2_SPF_SEEK 0x00040000
```

Figure 4.10: File-SPF ext2 file system inode flags.

Now that we have determined where to store the File-SPF Configuration data on disk, we must implement methods to store and retrieve the values on disk. The respective read_inode() and write_inode() functions of both the ext2 and ext3 file systems have to be modified to read or write the File-SPF appropriate bits of the raw-inode’s i_flag.
When reading an inode from disk, the `read_inode()` function must extract the appropriate bits from the `i_flag` variable in raw-inode and store them to the `spf` variable we have created in the os-inode structure. Similarly, the `write_inode()` function must map bits from the os-inode `spf` variable into the appropriate bits for storage in the raw-inode `i_flag`. The code to implement this within `ext2_read_inode()` and `ext2_write_inode()` is shown in Figure 4.11 and Figure 4.12. The bolded lines indicate the additions made to these functions. In the code for Figures 4.11 and 4.12, the variable `inode` points to the os-inode structure we have been describing. The `inode` pointer can be used to read or write the `spf` variable for File-SPF. The variable `raw_inode` points to the raw-inode structure we have been describing. This pointer is used to access the disk’s
*i_flag* variable. As can be seen in Figures 4.11 and 4.12, a combination of bitwise OR and bitwise XOR operations can properly get and set the appropriate bits in the raw-inode *i_flag*. The implementation for the *ext3_read_inode()* and *ext3_write_inode()* functions in the *ext3* file system are implemented very similarly to the code in Figures 4.11 and 4.12. The functions that have been modified to support File-SPF on disk are listed in Table 4.10.

Now that we have a means to store File-SPF Configuration data on disk, we need to implement system administrative tools to modify the File-SPF Configurations. In order to read and write the File-SPF Configuration to a file, we created two commands, *get_file_spf* and *set_file_spf*. Each respectively gets and sets the File-SPF Configuration for a file that is passed in at the command line. The *set_file_spf* command also takes the File-SPF Configuration value and assigns this value to the file indicated at the command line. The File-SPF Configuration value that is passed in at the command line should be the bitwise OR value of the constants, shown in Figure 4.9, that correlate to the operations we wish to disable security for on a particular file. For example, if we want turn off security checks for reads and seeks on the database file *mydatabase*, we run

```
set_file_spf 5 mydatabase
```

at the command line. In order to read the current File-SPF Configuration, we run the *get_file_spf* command and pass in the file names of the files we wish to see the File-SPF Configuration of. In the current implementation, these commands must be run as root.

To support these two new commands, we must implement appropriate system calls that can get and set the File-SPF Configurations. The system calls are named *get_file_spf()* and *set_file_spf()* respectively. The system call interface for these system calls is shown in Figure 4.13. The system calls are implemented by adding them to the system call tables in *include/asm-i386/unistd.h* file and *arch/i386/kernel/entry.S*.

These two system calls are implemented similarly to the *chmod()* and *stat()* system calls. Much like *chmod()* and *stat()* , *set_file_spf()* and *get_file_spf()* set or get file attributes respectively.
Small modifications have been made to a number of file attribute structures and functions so these two new system calls can operate properly. For example, we added an additional \textit{ia_spf} variable to the \textit{iaattr} structure in \texttt{include/linux/fs.h}. The functions that have been modified to support File-SPF can be found in Table 4.10.

\begin{verbatim}
#include <linux/spf.h>
#include <linux/unistd.h>

_syscall1(int, get_file_spf, char*, filename);
_syscall2(int, set_file_spf, char*, filename, int, spf_option);

int get_file_spf(const char * filename);

Returns: File-SPF Configuration value if OK, -1 on error

int set_file_spf(const char * filename, int spf_option);

Returns: 0 if OK, -1 on error
\end{verbatim}

Figure 4.13: System call interface for File-SPF.

Like was mentioned in the File-SPF design of Section 3.5, security must be put in place to stop illegitimate users from modifying the File-SPF Configuration. The \texttt{set_file_spf()} system call is very similar to the \texttt{chmod()} and the \texttt{chown()} system calls. They both modify file attributes and require system administrative ability to modify file attributes. Therefore, the \texttt{set_file_spf()} system call is implemented with the same security as the \texttt{chmod()} and \texttt{chown()} system calls. Similarly, the \texttt{get_file_spf()} system call is very similar to the \texttt{stat()} system call, in that they both retrieve file attributes. Therefore, \texttt{get_file_spf()} is given the same security as \texttt{stat()}.

As will be discussed in more detail in Section 4.5.2, File-SPF may suffer from additional disk I/O. Since File-SPF Configuration is stored on disk, our code may have to perform a disk I/O before a decision can be made to skip a security check. This can cause a large amount of disk I/O, and possibly negate the performance improvements of File-SPF. Therefore, we look for an efficient means to load the File-SPF values into memory so that they do not have to be continually read File-SPF Configuration values from disk. This is imperative for improving the performance of the File-SPF implementation.
Luckily, there seems to be an easy way for the File-SPF Configuration data to be stored and accessed efficiently in main memory. The read(), write(), and seek() system calls are all called along with a file descriptor. A file descriptor is loaded into the kernel when a program first uses the open() system call to open a file. Since files must be opened before the read(), write(), and seek() system calls can be executed, the open() system call is an ideal function to load the File-SPF information from disk into memory. In addition, the File-SPF Configuration can be stored into the operating system’s file descriptor table. By storing the File-SPF Configuration in the file descriptor table, the File-SPF Configurations can be accessed efficiently through the use of the file descriptor passed into system calls. Now, File-SPF Configurations can be read from the file descriptor table. Security relevant system calls and kernel functions can retrieve the File-SPF Configuration in the file descriptor table to determine if a security check should be skipped. Figure 4.14 illustrates how File-SPF Configurations are modified, loaded into the file descriptor table, and then accessed by system calls.

Figure 4.14: Loading File-SPF Configuration.
### Operation Functions Modified

<table>
<thead>
<tr>
<th>Operation</th>
<th>Functions Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read File</td>
<td>sys_sendfile(), sys_read(), sys_readv(), sys_read()</td>
</tr>
<tr>
<td>Write File</td>
<td>sys_sendfile(), sys_write(), sys_writev(), sys_pwrite()</td>
</tr>
<tr>
<td>Seek File</td>
<td>sys_lseek(), sys_llseek()</td>
</tr>
</tbody>
</table>

Table 4.9: Functions modified to implement File-SPF.

<table>
<thead>
<tr>
<th>Support Role</th>
<th>Functions Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieving File-SPF Configurations on Disk</td>
<td>ext2_read_inode(), ext3_read_inode(), dentry_open()</td>
</tr>
<tr>
<td>Storing File-SPF Configurations on Disk</td>
<td>ext2_write_inode(), ext3_write_inode(), open_name()</td>
</tr>
<tr>
<td>System Administration of File-SPF Configurations</td>
<td>inode_setattr(), setattr_mask()</td>
</tr>
</tbody>
</table>

Table 4.10: Functions modified to support File-SPF.

```c
asmlinkage ssize_t sys_read(unsigned int fd, char * buf, size_t count) {
    ssize_t ret;
    struct file * file;
    ret = -EBADF;
    file = fget(fd);
    if (file)
        if (file->f_mode & FMODE_READ) {
            ret = locks_verify_area(FLOCK_VERIFY_READ, file->f_dentry->d_inode, file, file->f_pos, count);
            if (!ret)
                ssize_t (*read)(struct file *, char *, size_t, loff_t *);
                ret = -EINVAL;
            if (file->f_op && (read = file->f_op->read) != NULL) {
                if (((file->spf & SPF_READ) > 0) && S_ISREG(file->f_dentry->d_inode->i_mode))
                    ret = read(file, buf, count, &file->f_pos);
                } else {
                    ret = security_ops->file_ops->permission(file, MAY_READ);
                    if (!ret)
                        ret = read(file, buf, count, &file->f_pos);
                }
        }
    if (ret > 0)
        dnotify_parent(file->f_dentry, DN_ACCESS);
    fput(file);
    return ret;
}
```

Figure 4.15: File-SPF modification of the `read()` system call.
In order for the file descriptor table to hold the File-SPF Configuration value, the file descriptor structure in `include/linux/fs.h` has been modified to include an additional variable, named `spf`. In addition, the File-SPF Configuration must be retrieved from disk during an `open()` system call and copied into the file descriptor table. The appropriate functions modified to accomplish this are listed in Table 4.10.

Figure 4.15 shows how the `read()` system call has been modified to support File-SPF. The bolded lines in Figure 4.15 indicate the modifications made to the `read()` system call compared to the original SELinux code in Figure 4.1. The variable `file` is a pointer to the file descriptor table entry associated with the file descriptor `fd` passed into the system call. The `file` pointer is illustrated in Figure 4.14. By using the `file` pointer, the File-SPF Configuration `spf` can be accessed with the statement `file->spf`. The modification to other system calls follows the same pattern as Figure 4.15. Table 4.9 lists which specific functions have been modified to implement File-SPF.

4.5.2 File-SPF Implementation Advantages and Disadvantages

As was indicated earlier, fewer operations are likely to benefit from our File-SPF implementation compared to System-SPF or Process-SPF. The reason for this is not a design issue, but really implementation of the system call interface in Linux. The majority of file system calls work with a file or directory entry on disk. For example, system calls like `stat()` and `open()` operate this way. When a call is made to `stat()` or `open()`, the file must be accessed off of disk in order for the operation to be performed. If File-SPF is implemented in these system calls, the File-SPF Configuration must be read off disk before we can determine if the security check should be skipped. It is impossible to implement File-SPF for some system calls without incurring this additional disk I/O. This is the major disadvantage of File-SPF compared to the other SPF implementations. This problem limits the types of operations File-SPF can benefit.

Another implementation disadvantage is with the storage of security options on disk. As was stated in the implementation section, the best way to store File-SPF Configurations on disk is by using the unused bits in the `ext2` or `ext3` file system’s inode `i_flag` variable. If by chance there
are not enough unused bits in the \textit{i\_flag} variable or future file systems, our implementation cannot be done.

On the other hand, there are several side benefits to our implementation of File-SPF that we originally did not expect. Every file in the operating system will have the File-SPF Configuration bits unset in the \textit{i\_flag} variable, because these bits are unused in the inode \textit{i\_flag} variable. That means every single file in SELinux has no security checks turned off by default. This can help makes system administration easier when the system starts up, because we expect that the majority of files will not require File-SPF Configurations to be modified. In addition, our method can save some disk space by not adding additional fields to the raw-inode.

\subsection*{4.5.3 File-SPF System Administration}

The system administration of File-SPF may be easy or hard, depending on the particular needs of the system and programs. If all files remain on disk, regardless of the activity of a process or application, File-SPF will be quite easy to administer. The system administrator only needs to set the appropriate File-SPF Configurations on each file. The files that require File-SPF Configuration will most likely be stored in a small number of directories. The system administrator can use the \texttt{set\_file\_spf} command to set the File-SPF Configuration on files in those directories.

However, File-SPF assumes that files are sitting in directories and can be modified with the \texttt{set\_file\_spf} command. If the files used by an application are created and destroyed during the execution lifetime of an application, the performance benefits of File-SPF cannot be taken advantage of unless it is programmed into the application. The program must be modified to call the \texttt{set\_file\_spf(\)} system call and set the File-SPF Configuration on newly created files. We assume that the code for programs will actually be available. If the source code is not available, then File-SPF may not be usable in some applications.

Unlike System-SPF and Process-SPF, our implementation of File-SPF does not allow for dynamic modification of the SPF Configuration. In System-SPF and Process-SPF, the SPF
Configuration of the system can be configured at run time, even as programs are executing. File-SPF Configuration values are loaded during the open() system call and stored within Linux’s file descriptor table. Once it is loaded, the File-SPF Configuration values stay the same in the file descriptor table. Applications will not see any File-SPF changes until the original file is closed and re-opened.

There are several possible solutions to this problem, however we decided that neither is appropriate nor necessarily useful. When the set_file_spf command is executed on a file, we can check every file in the open file descriptor table to see if the file is already opened. However, this can slow down the execution for the set_file_spf() system call significantly, since the file descriptor table is quite large.

Another way would be to create a set of system calls, and associated commands, that let a system administrator modify the File-SPF Configurations for open files. This can be implemented similarly to the fcntl() system call, which modifies flags on open files. However, such system calls and commands are unnecessary, if even useless. Aside from creating yet another set of system calls and commands, there is no way for a system administrator to view the current file descriptor table. In other words, there is no file descriptor equivalent of the ps command.

Therefore, there is no easy way for a system administrator to determine which file descriptor ID is currently used by which process and which file each file descriptor ID is associated with. Thus, a system administrator cannot determine which file descriptor’s File-SPF Configuration needs to be modified. Therefore, if we want the capability to change File-SPF values on open files, we might need to implement them inside the application. If this must be done, the system administration issues behind File-SPF may become even more difficult than already thought.

### 4.5.4 File-SPF Security Issues

Overall, there does not seem to be any major security holes opened up by our implementation of File-SPF in SELinux. If there are security holes due to SPF, they are due to previously unknown security holes in SELinux. The get_file_spf() and set_file_spf() system calls are protected by the same SELinux security mechanisms that protect other file attribute system calls. If this security
policy is not appropriate, a new type of security policy can be created just for these two system calls.

Another security issue may come up when files are copied. We must decide if File-SPF Configurations should be copied from one file to another when files are copied. In our File-SPF implementation, File-SPF Configurations are not copied when files are copied. We did not want File-SPF Configurations to propagate to a possibly large number of files in the operating system.

4.6 Summary of Implementation

As can be seen from the implementation details in Sections 4.3, 4.4, and 4.5, there are a variety of advantages and disadvantages that come from implementing SPF in each of the different levels of the operating system. Each method has its unique advantages and disadvantages. Most of the implementation methods seem to be implemented such that system administration is fairly simple. The lone exception is with File-SPF. File-SPF may be extremely difficult to administer if code must be modified. It may be impossible to administer if code is not available for modification.

The security of SPF at each operating system level is relatively safe. The only major security concern is the fact that SPF exists in the system. An important feature of Trusted Operating Systems is security redundancy. If security breaches occur, additional security is there to protect the system. However, a malicious user may take advantage of SPF and use it to make security breaches that are not possible in Trusted Operating System. In order to make SPF more secure, specific SPF options can be taken out of the implementation. For example, the SPF placed within the write() system call can be taken out. These types of issues should be considered if future development of SPF is done.

Overall, it seems that System-SPF may be the easiest type of SPF implementation to manage. It is also the most secure way to implement SPF if we hard code the System-SPF Configuration into the kernel. By doing so, it removes the possibility that a malicious user can take advantage of System-SPF, since the configuration of System-SPF cannot be modified. However,
System-SPF lacks the ability to offer fine-grained security control that may be desired by system administrators.

That is why Process-SPF or File-SPF may be the better operating system level to implement SPF within. However, there are a number of downsides with these methods. Process-SPF does not assure as much safety as System-SPF and File-SPF. File-SPF cannot offer performance improvements for a number of system operations and cannot let system administrators change the File-SPF Configurations of currently opened files.

Without considering the performance difference between System-SPF, Process-SPF, and File-SPF, it is really difficult to determine if implementing SPF at any particular level of the operating system can be considered the best. There are clear advantages and disadvantages of each implementation. The best SPF implementation technique will most likely depend on the type of workload that will execute on the system. If the system will serve primarily as a video on demand server, then the File-SPF implementation may be the best implementation. However, for a system that is dedicated as a web server, System-SPF may be the best. For a generic system, Process-SPF may be the best because it provides a nice balance between System-SPF and File-SPF.
CHAPTER 5

Performance Evaluation of Security Performance Flexibility

This Chapter will go over the various benchmarks that are used to evaluate the potential performance improvements SPF can bring and the results of these benchmarks. The benchmarks that will be used to evaluate SPF will be described briefly in Section 5.1. Next, Section 5.2 will discuss our experimental setup and testing methodology. In Section 5.3, we will discuss the performance metrics we will use to evaluate SPF. Section 5.4 will discuss the theoretical performance goals that are desired from SPF. Section 5.5 will show the results of running our benchmarks on Redhat, SELinux without SPF, SELinux with System-SPF, SELinux with Process-SPF, and SELinux with File-SPF. The data in Section 5.5 will show that SPF can offer differing amounts of performance improvements, depending on the particular benchmark and the workload the benchmark executes. Section 5.6 summarizes and concludes this chapter by discussing the benchmark results, providing analysis of the results, comparing the results in Section 5.5 to the results we expected in Section 5.4, and discussing the overall potential that SPF may have.

5.1 Benchmarks

In order to validate and evaluate the performance impact the SPF framework can have, a number of benchmarks have been selected or created. The benchmarks are a combination of standard benchmarks, used in the Linux community, as well as several originally created benchmarks. The standard benchmarks come from the collection found at the Linux Benchmark Suite Homepage [LBS]. Of the standard benchmarks, there are two types, microbenchmarks and application benchmarks. Microbenchmarks are primarily used for measuring latency and throughput of system operations and are not meant to measure actual workloads. Application benchmarks attempt to model and measure real system workloads.
The following sections give an overview of the benchmarks selected from [LBS] or created. A number of these benchmarks have options that will modify the workload or execution of the benchmarks. For those who are familiar with these benchmarks, the details of the selected parameters can be found in Appendix A. Any implementation details or modifications to benchmarks are detailed in Appendix B.

5.1.1 Microbenchmarks

5.1.1.1 AIM9

AIM9 is a microbenchmark that measures the rate a variety of system operations, including I/O, system calls, functions, and general computation, can be processed [AIM]. To the author’s knowledge, the only version of the benchmark is located at [AIM].

5.1.1.2 UnixBench

UnixBench is a microbenchmark that measures the throughput of a variety of system calls and I/O operations. UnixBench version 4.0.1 will be used in this thesis.

5.1.2 Standard Application Benchmarks

5.1.2.1 AIM7

AIM7 models common system workloads through a mix of executables. It measures the rate jobs can be executed under these workloads and the elapsed time a particular workload takes to execute [AIM]. The benchmark comes pre-configured to test multiuser/shared system workloads, computational workloads, database workloads, and file server workloads. The multiuser/shared system workload attempts to model a remote server used by multiple users to accomplish tasks, such as a typical University machine used by students. The database workload models typical database applications by executing interprocess communication (IPC) and file operations. The file server workload models a Network File System (NFS) server through a mix
of disk operations and network operations. The computational workload executes a mixture of computational tasks and disk I/O. To the author’s knowledge, the only version of the benchmark is located at [AIM].

5.1.2.2 WebStone

WebStone stresses a web server to measure throughput and latency of the server [WEBSTONE]. WebStone version 2.5 will be used with Apache HTTP server 2.0.39 [APACHE]. Unfortunately, due to the security placed on University machines, WebStone cannot be executed in its default manner. Details of how WebStone is executed, despite these limitations, can be found in Appendix B.

5.1.3 Original Application Benchmarks

5.1.3.1 MPEG Frame Rate

A MPEG video test has been created that measures the frame rate MPEG videos can be played at. A slightly modified Berkeley MPEG player is used to play a video clip at differing qualities while a workload executes in the background [MPEGPLAY]. The workload selected to run in the background is a modification of the Quick and Dirty Development Application Benchmark (QDDA) [QDDA]. Berkeley MPEG player version 2.4 will be used. To the author’s knowledge, the only version of QDDA is located at [QDDA]. Details of modifications done to the Berkeley MPEG player and QDDA can be found in Appendix B.

5.1.3.2 Video on Demand Frame Rate

The same issues, described in Appendix B for WebStone, also apply to the Video On Demand benchmark. Therefore, a real Video On Demand server cannot be used to measure Video On Demand frame rate. A Video On Demand benchmark has been created to simulate a typical Video On Demand server’s operations. The Video On Demand benchmark measures the frame
rate each stream of a Video On Demand server is played at. The details of the benchmark’s implementation are described in Appendix B.

5.2 Experimental Setup and Methodology

The benchmarks from Section 5.1 will be executed on a 933 MHz Pentium III with 256 Megs of RAM and Redhat 7.2 installed. The AIM9, UnixBench, AIM9, and MPEG frame rate benchmarks will be executed under user accounts. The WebStone and the Video On Demand frame rate benchmarks will be executed under root because web servers and Video On Demand servers are typically executed under a root account.

With one exception, each benchmark will be executed on five different kernels: Redhat, SELinux without SPF, SELinux with System-SPF, SELinux with Process-SPF, and SELinux with File-SPF. The performance measurements gathered on these five kernels will be analyzed and compared to each other. The Linux kernel version used in this thesis is version 2.4.18. The appropriate SPF Configurations will be set prior to the benchmark executing on any of the SPF kernels. When executing benchmarks in SELinux with System-SPF, the appropriate System-SPF conditions will be set using the `sysctl` command. For multimedia benchmarks, the appropriate File-SPF Configurations will be set on the video files using the `set_file_spf` command. Some benchmarks need to be modified to support the benchmarking of File-SPF. In some benchmarks, files are created and destroyed as the benchmarks are executed. Therefore, the benchmarks must to be reprogrammed so the `set_file_spf()` system call can set the appropriate File-SPF Configuration on the benchmark files. When executing benchmarks under the Process-SPF, the appropriate Process-SPF Configuration is set on the benchmark binary using the `set_bin_spf` command.

5.3 Measurement Metrics

Section 5.1 described the benchmarks selected or created to validate and evaluate SPF. Many of the standard benchmarks have a number of different tests or metrics to measure performance. This section will indicate which tests and metrics from these benchmarks will be used to evaluate
SPF performance. The metrics used to measure SPF improvement over SELinux will also be discussed.

5.3.1 Microbenchmarks

The microbenchmarks will be used to measure and determine the overhead cost of implementing SPF in SELinux. System-SPF, Process-SPF, and File-SPF all incur some amount of overhead. The microbenchmarks will be used to measure this overhead and determine if any particular SPF implementation has performance advantages over the others.

Security in SELinux and SPF do not affect a number of the microbenchmark tests. For example, mathematical operations are not affected by the security of SELinux and SPF is not implemented with interprocess communication (IPC) features of SELinux. The microbenchmark tests that concentrate on mathematical operations will not show any performance differences between Redhat and SELinux. Similarly, IPC tests will not show any performance differences between SELinux and a SPF implemented kernel. Therefore, these tests will not be shown. The only microbenchmark tests that will be used are file system measurements. Each of the microbenchmark tests measures performance in different units. Therefore, the units of those tests will be indicated with the results.

In order to measure the performance improvements of SPF, we must first calculate the performance degradation of SELinux without SPF, SELinux with System-SPF, SELinux with Process-SPF, and SELinux with File-SPF. The performance degradation will be calculated relative to Redhat as the baseline measurement. Let R be the performance measurement of the benchmark test under Redhat, S be the measurement under SELinux without SPF, SSPF be the measurement under SELinux with System-SPF, PSPF be the measurement under SELinux with Process-SPF, and FSPF be the benchmarks measurement under SELinux with File-SPF. Mathematically, performance degradation can be calculated as follows:

SELinux Performance Degradation = ((S – R)/R)*100
SELinux with System-SPF Performance Degradation = ((SSPF – R)/R)*100
The metric that will be used to measure the performance improvement of SPF will be the difference between the performance degradation of SELinux with SPF and SELinux without SPF. Let us abbreviate the above calculations so that SELinux Performance Degradation is SPD, SELinux with System-SPF Performance Degradation is SSPF-PD, SELinux with Process-SPF Performance Degradation is PSPF-PD, and SELinux with File-SPF Performance Degradation is FSPF-PD. Mathematically, the performance improvements will be calculated as follows:

System-SPF Improvement = SSPF-PD - SPD
Process-SPF Improvement = PSPF-PD - SPD
File-SPF Improvement = FSPF-PD - SPD

Therefore, if a SPF implemented kernel is able to decrease the performance degradation, caused by SELinux security checks, System-SPF Improvement, File-SPF Improvement, and Process-SPF Improvement will be a positive value. A negative value indicates that SPF makes the performance even worse. If the absolute value of SELinux Performance Degradation is equal to the System-SPF, Process-SPF, or File-SPF improvement, then this means SPF has regained all of the performance lost to SELinux security. In other words, SPF is able to eliminate the performance loss caused by SELinux security.

5.3.2 Standard Application Benchmarks

5.3.2.1 AIM7

As was mentioned in Section 5.1, there are 4 different workload mixes that can be executed in the AIM7 benchmark: multiuser/shared system workload, database workload, file server workload, and compute server workload. Of these four workloads, the database workload benchmark will be the only one executed to measure the performance improvements SPF can bring. It is unlikely that a network file system (NFS) will desire any particular file system
security checks to be disabled. The computational workload is hardly affected by security in SELinux. The multiuser benchmark can potentially benefit from System-SPF. However, the security needs of such an environment cannot be determined.

The AIM7 benchmark produces two different metrics to measure system performance of the database workload. The metrics are job throughput (measured in jobs per minute) and elapsed execution time (measured in seconds). The elapsed execution time is directly proportional to the job throughput, so only the job throughput will be used to evaluate performance.

The metric that will be used to measure the performance improvement of SPF will be the difference between the job throughput degradation under SELinux with SPF and the job throughput degradation under SELinux without SPF. The mathematical calculations for this metric are identical to the formulas in Section 5.3.1.

5.3.2.2 WebStone

As was mentioned in Section 5.1, WebStone measures web server throughput and latency to evaluate the performance of a web server under a certain amount of stress. However, WebStone outputs these measurements for each client that stresses the web server. It does not aggregate the throughput and latency measurements of each client that stresses the web server.\(^5\) The metric we will use from WebStone will be the average throughput of all the WebStone clients. Latency will not be used as a metric to evaluate web server performance. Accurate latency results cannot be obtained on the shared University network.

In order to measure the performance improvement, SPF can bring to web servers, we will take the difference between the web server throughput degradation under SELinux with SPF and web server throughput degradation under SELinux without SPF. The mathematical calculations of this metric are identical to the formulas in Section 5.3.1.

\(^5\) Normally, WebStone can aggregate measurements. However, WebStone cannot under this thesis’ experimental environment. Please see Appendix B for additional information.
Unlike the other benchmarks in this section, WebStone will be the only benchmark that will not be executed under File-SPF. In web servers, the majority of data is small and will be opened, read, and closed immediately. File-SPF benefits applications that will open a file and operate on that file multiple times, such as a MPEG player. Therefore, File-SPF is not suitable for web server workloads.

On the other hand, System-SPF and Process-SPF are suitable for web server workloads. Many web servers are run on dedicated machines. System-SPF is suitable for web server workloads because System-SPF is applied globally to all applications and programs. Process-SPF can also be used for web servers because most web servers have multiple executable components. Besides the actual web server, web servers often have CGI executables. Process-SPF allows a system administrator to place different Process-SPF Configurations on each executable within the system.

5.3.3 Original Application Benchmarks

5.3.3.1 MPEG Frame Rate

The MPEG frame rate benchmark outputs the frame rate a MPEG video is played at. We will measure the performance improvement of SPF on multimedia by taking the difference between frame rate degradation under SELinux with SPF and frame rate degradation under SELinux without SPF. The mathematical calculations for this metric are identical to the formulas in Section 5.3.1.

5.3.3.2 Video On Demand Frame Rate

The Video On Demand benchmark outputs the frame rate of each video stream played in the benchmark. We will use average frame rate of all the video streams as the metric to evaluate SPF under a Video On Demand server. Just like the MPEG frame rate benchmark, the metric we will use to measure the performance improvements of SPF will be the difference between frame
rate degradation under SELinux with SPF and frame rate degradation under SELinux without SPF. The mathematical calculations for this metric are identical to the formulas in Section 5.3.1.

5.4 Security Performance Flexibility Performance Expectations

Figure 5.1 illustrates the performance degradation and performance improvements we expect to see when we execute our application benchmarks. The Y-axis indicates the theoretical percentage performance degradation of a benchmark when it is executed between a SELinux kernel (with or without SPF) and Redhat. Along the X-axis the workload for a particular benchmark increases. A point above the X-axis indicates that SELinux outperforms Redhat. A point below the X-axis indicates SELinux performs worse than Redhat.

The solid line in Figure 5.1 represents our expected percentage performance degradation of SELinux without SPF when compared to Redhat. The hope we have is that the performance difference between SELinux without SPF and Redhat is negative. In other words, SELinux without SPF performs worse than Redhat. Initially, when the workload is relatively small, we do not expect security to take up a large amount of system resources. Therefore, the percentage performance degradation should be relatively small. As the workload continues to increase, we expect the performance difference between SELinux and Redhat to grow as SELinux security
takes up more resources. However, we anticipate there will be some degradation limit between SELinux and Redhat.

The dotted line in Figure 5.1 represents the performance improvements we hope SPF can bring. SPF will hopefully make the performance decrease in SELinux smaller. Due to the overhead of SPF, it may not be possible for the performance to equal the original Redhat performance, but hopefully it can make a significant improvement on performance.

5.5 Performance of Security Performance Flexibility

This section will show the results of the benchmarks when executed on the Redhat, SELinux without SPF, SELinux with System-SPF, SELinux with Process-SPF, and SELinux with File-SPF kernels. The results will be analyzed to see how well each of the SPF implementations is able to improve performance of SELinux when compared to Redhat.

Originally, separate sections were dedicated to the benchmark results for each System-SPF, Process-SPF, and File-SPF implementations. However, after benchmark results were collected from each of the implementations, we discovered that there is very little performance difference between the System-SPF, Process-SPF, and File-SPF implementations. Even though each is implemented differently, they all have overhead that is far smaller than the original security checks. It is difficult to determine if any particular implementation has a performance advantage over the others. The results below will show the results of System-SPF, Process-SPF, and File-SPF results side by side. It can be seen that there is very little performance difference between the three implementations.

5.5.1 Microbenchmarks

Tables 5.1 through 5.6 show the results from the AIM9 and UnixBench microbenchmarks that are relevant to SPF. Tables 5.1 and 5.4 show the results of the AIM9 and UnixBench with the security checks still executed. Tables 5.2 and 5.5 show the results of the microbenchmarks with the security checks skipped. For each of the benchmark tests in Tables 5.2 and 5.5, the
appropriate SPF Configuration is set so the microbenchmark operation skips security checks. Tables 5.3 and 5.6 list the percentage performance degradation that SELinux without SPF suffers when compared to Redhat. The tables also list the performance increase that each SPF implemented kernel is able to regain from the original SELinux performance loss. The numbers are calculated as discussed in Section 5.3.1. The AIM9 directory benchmarks are omitted from the Process-SPF and File-SPF columns of Tables 5.1 through 5.3. As Chapters 3 explained, directory operations are not supported under Process-SPF and File-SPF. To obtain accurate results, the microbenchmarks are executed 5 times on each of the kernels. The numbers are averaged to give us the numbers in Tables 5.1, 5.2, 5.4, and 5.5.

The results confirm that SELinux security can have a significant affect on the performance of specific operations. Tables 5.3 and 5.6 indicate the percentage performance degradation SELinux without SPF suffered when compared to Redhat. The percentage losses are quite significant. Performance loss ran as high as 41.89% in one case, but most of the tests had performance loss ranging from approximately 5% to 15%, which is still quite significant. The percentage improvement numbers in Tables 5.3 and 5.6 indicate that SPF can regain the majority of the performance lost to SELinux security.

The data in Tables 5.1 through 5.6 indicate that the overhead of SPF is quite small. Tables 5.3 and 5.6 show that in almost all benchmark tests, SPF is able to regain almost all the performance lost due to SELinux security. The performance that cannot be gained is due to the overhead of SPF. The percentage improvement numbers in Tables 5.3 and 5.6 indicate that SPF overhead lowers the performance of these microbenchmarks by approximately 1 to 2 percent. Overall, the results indicate exactly what we expect. The overhead of SPF is far less than the overhead of SELinux security. This difference in overhead enables us to gain performance through the use of the SPF framework.

The data in the tables below do not give a clear indication of any SPF implementation (System-SPF, File-SPF, Process-SPF) that has a performance advantage over the others. The AIM9 benchmarks seem to indicate that System-SPF or Process-SPF may have lower overhead than File-SPF. However, the UnixBench benchmark indicates that File-SPF may have the lower
implementation overhead. Overall, these results suggest that the overhead of all three implementations may be approximately equal.

<table>
<thead>
<tr>
<th>File System Tests</th>
<th>Redhat</th>
<th>SELinux No SPF</th>
<th>SELinux System-SPF</th>
<th>SELinux Process-SPF</th>
<th>SELinux File-SPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Disk Reads (K)/second</td>
<td>100547.12</td>
<td>94167.04</td>
<td>93135.59</td>
<td>94531.9</td>
<td>93106.73</td>
</tr>
<tr>
<td>Random Disk Writes (K)/second</td>
<td>84529.51</td>
<td>79188.24</td>
<td>79508.83</td>
<td>80496.64</td>
<td>79600.98</td>
</tr>
<tr>
<td>Sequential Disk Reads (K)/second</td>
<td>371456.00</td>
<td>335527.37</td>
<td>325591.04</td>
<td>332840.96</td>
<td>326901.76</td>
</tr>
<tr>
<td>Sequential Disk Writes (K)/second</td>
<td>159010.60</td>
<td>149616.64</td>
<td>153174.03</td>
<td>152419.35</td>
<td>149088.44</td>
</tr>
<tr>
<td>Disk Copies (K)/second</td>
<td>110075.60</td>
<td>102252.55</td>
<td>102744.05</td>
<td>103895.04</td>
<td>100761.60</td>
</tr>
<tr>
<td>Directory Searches/second</td>
<td>65973.45</td>
<td>38335.35</td>
<td>37657.05</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Directory Operations/second</td>
<td>2756600.00</td>
<td>2705540.00</td>
<td>2698520.00</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: AIM9 Benchmark Results, Security Checks Executed.

<table>
<thead>
<tr>
<th>File System Tests</th>
<th>Redhat</th>
<th>SELinux No SPF</th>
<th>SELinux System-SPF</th>
<th>SELinux Process-SPF</th>
<th>SELinux File-SPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Disk Reads (K)/second</td>
<td>100547.12</td>
<td>94167.04</td>
<td>99762.33</td>
<td>100403.2</td>
<td>97102.04</td>
</tr>
<tr>
<td>Random Disk Writes (K)/second</td>
<td>84529.51</td>
<td>79188.24</td>
<td>84768.48</td>
<td>85123.42</td>
<td>82826.39</td>
</tr>
<tr>
<td>Sequential Disk Reads (K)/second</td>
<td>371456.00</td>
<td>335527.37</td>
<td>363571.20</td>
<td>361014.22</td>
<td>362158.08</td>
</tr>
<tr>
<td>Sequential Disk Writes (K)/second</td>
<td>159010.60</td>
<td>149616.64</td>
<td>159727.37</td>
<td>160055.67</td>
<td>150279.23</td>
</tr>
<tr>
<td>Disk Copies (K)/second</td>
<td>110075.60</td>
<td>102252.55</td>
<td>110315.52</td>
<td>109041.4</td>
<td>103270.40</td>
</tr>
<tr>
<td>Directory Searches/second</td>
<td>65973.45</td>
<td>38335.35</td>
<td>40259.85</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Directory Operations/second</td>
<td>2756600.00</td>
<td>2705540.00</td>
<td>2753680.00</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.2: AIM9 Benchmark Results, Security Checks Skipped.

<table>
<thead>
<tr>
<th>File System Tests</th>
<th>SELinux No SPF Performance Degradation</th>
<th>SELinux System-SPF Improvement Over SELinux No SPF</th>
<th>SELinux Process-SPF Improvement Over SELinux No SPF</th>
<th>SELinux File-SPF Improvement Over SELinux No SPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Disk Reads (K)/second</td>
<td>-6.35%</td>
<td>+5.57%</td>
<td>+6.21%</td>
<td>+2.92%</td>
</tr>
<tr>
<td>Random Disk Writes (K)/second</td>
<td>-6.32%</td>
<td>+6.60%</td>
<td>+7.02%</td>
<td>+4.31%</td>
</tr>
<tr>
<td>Sequential Disk Reads (K)/second</td>
<td>-9.67%</td>
<td>+7.55%</td>
<td>+6.86%</td>
<td>+7.17%</td>
</tr>
<tr>
<td>Sequential Disk Writes (K)/second</td>
<td>-5.91%</td>
<td>+6.36%</td>
<td>+6.57%</td>
<td>+0.42%</td>
</tr>
<tr>
<td>Disk Copies (K)/second</td>
<td>-7.11%</td>
<td>+7.33%</td>
<td>+6.17%</td>
<td>+0.93%</td>
</tr>
<tr>
<td>Directory Searches/second</td>
<td>-41.89%</td>
<td>+2.91%</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Directory Operations/second</td>
<td>-1.85%</td>
<td>+1.74%</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.3: AIM9 Benchmark Improvement, Security Checks Skipped.
### Table 5.4: UnixBench Benchmark Results, Security Checks Executed.

<table>
<thead>
<tr>
<th>File System Tests</th>
<th>Redhat (KB/sec)</th>
<th>SELinux No SPF (KB/sec)</th>
<th>SELinux System-SPF (KB/sec)</th>
<th>SELinux Process-SPF (KB/sec)</th>
<th>SELinux File-SPF (KB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Read 256B Buffer</td>
<td>164178.20</td>
<td>137348.80</td>
<td>136456.20</td>
<td>137537.8</td>
<td>134197.60</td>
</tr>
<tr>
<td>File Write 256B Buffer</td>
<td>160825.00</td>
<td>135722.80</td>
<td>130492.80</td>
<td>133475.6</td>
<td>133879.20</td>
</tr>
<tr>
<td>File Copy 256B Buffer</td>
<td>76769.60</td>
<td>64952.60</td>
<td>63179.20</td>
<td>64980.8</td>
<td>64088.40</td>
</tr>
<tr>
<td>File Read 1KB Buffer</td>
<td>367885.20</td>
<td>335115.00</td>
<td>328872.40</td>
<td>331970.8</td>
<td>327929.80</td>
</tr>
<tr>
<td>File Write 1KB Buffer</td>
<td>200432.40</td>
<td>189583.60</td>
<td>191452.80</td>
<td>191578.8</td>
<td>191409.40</td>
</tr>
<tr>
<td>File Copy 1KB Buffer</td>
<td>129604.60</td>
<td>120578.20</td>
<td>120877.80</td>
<td>121285.8</td>
<td>120653.60</td>
</tr>
<tr>
<td>File Read 4KB Buffer</td>
<td>538517.40</td>
<td>516731.80</td>
<td>515341.60</td>
<td>514826.8</td>
<td>514716.20</td>
</tr>
<tr>
<td>File Write 4KB Buffer</td>
<td>224121.60</td>
<td>221207.60</td>
<td>224421.20</td>
<td>224154.8</td>
<td>223731.00</td>
</tr>
<tr>
<td>File Copy 4KB Buffer</td>
<td>156394.40</td>
<td>152110.20</td>
<td>154932.60</td>
<td>154295.2</td>
<td>153487.40</td>
</tr>
</tbody>
</table>

### Table 5.5: UnixBench Benchmark Results, Security Checks Skipped.

<table>
<thead>
<tr>
<th>File System Tests</th>
<th>Redhat (KB/sec)</th>
<th>SELinux No SPF (KB/sec)</th>
<th>SELinux System-SPF (KB/sec)</th>
<th>SELinux Process-SPF (KB/sec)</th>
<th>SELinux File-SPF (KB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Read 256B Buffer</td>
<td>164178.20</td>
<td>137348.80</td>
<td>161259.60</td>
<td>159056.4</td>
<td>161715.60</td>
</tr>
<tr>
<td>File Write 256B Buffer</td>
<td>160825.00</td>
<td>135722.80</td>
<td>158814.60</td>
<td>155229.2</td>
<td>159405.40</td>
</tr>
<tr>
<td>File Copy 256B Buffer</td>
<td>76769.60</td>
<td>64952.60</td>
<td>75257.40</td>
<td>75209.0</td>
<td>75530.0</td>
</tr>
<tr>
<td>File Read 1KB Buffer</td>
<td>367885.20</td>
<td>335115.00</td>
<td>365526.60</td>
<td>364978.8</td>
<td>366961.60</td>
</tr>
<tr>
<td>File Write 1KB Buffer</td>
<td>200432.40</td>
<td>189583.60</td>
<td>204503.80</td>
<td>205676.8</td>
<td>206377.00</td>
</tr>
<tr>
<td>File Copy 1KB Buffer</td>
<td>129604.60</td>
<td>120578.20</td>
<td>130455.60</td>
<td>130986.4</td>
<td>130452.00</td>
</tr>
<tr>
<td>File Read 4KB Buffer</td>
<td>538517.40</td>
<td>516731.80</td>
<td>536233.60</td>
<td>533637.2</td>
<td>534017.80</td>
</tr>
<tr>
<td>File Write 4KB Buffer</td>
<td>224121.60</td>
<td>221207.60</td>
<td>229098.60</td>
<td>231857.0</td>
<td>231340.80</td>
</tr>
<tr>
<td>File Copy 4KB Buffer</td>
<td>156394.40</td>
<td>152110.20</td>
<td>158921.00</td>
<td>158012.8</td>
<td>158374.40</td>
</tr>
</tbody>
</table>

### Table 5.6: UnixBench Benchmark Improvement, Security Checks Skipped.

<table>
<thead>
<tr>
<th>File System Tests</th>
<th>SELinux No SPF Performance Degradation</th>
<th>SELinux System-SPF Improvement Over SELinux No SPF</th>
<th>SELinux Process-SPF Improvement Over SELinux No SPF</th>
<th>SELinux File-SPF Improvement Over SELinux No SPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Read 256B Buffer</td>
<td>-16.34%</td>
<td>+14.56%</td>
<td>+13.22%</td>
<td>+14.84%</td>
</tr>
<tr>
<td>File Write 256B Buffer</td>
<td>-15.61%</td>
<td>+14.36%</td>
<td>+12.13%</td>
<td>+14.73%</td>
</tr>
<tr>
<td>File Copy 256B Buffer</td>
<td>-15.39%</td>
<td>+13.42%</td>
<td>+13.36%</td>
<td>+13.78%</td>
</tr>
<tr>
<td>File Read 1KB Buffer</td>
<td>-8.91%</td>
<td>+8.27%</td>
<td>+8.12%</td>
<td>+8.66%</td>
</tr>
<tr>
<td>File Write 1KB Buffer</td>
<td>-5.41%</td>
<td>+7.44%</td>
<td>+8.03%</td>
<td>+8.38%</td>
</tr>
<tr>
<td>File Copy 1KB Buffer</td>
<td>-6.96%</td>
<td>+7.62%</td>
<td>+8.03%</td>
<td>+7.61%</td>
</tr>
<tr>
<td>File Read 4KB Buffer</td>
<td>-4.05%</td>
<td>+3.63%</td>
<td>+3.14%</td>
<td>+3.21%</td>
</tr>
<tr>
<td>File Write 4KB Buffer</td>
<td>-1.30%</td>
<td>+3.52%</td>
<td>+4.75%</td>
<td>+4.52%</td>
</tr>
<tr>
<td>File Copy 4KB Buffer</td>
<td>-2.74%</td>
<td>+4.36%</td>
<td>+3.77%</td>
<td>+4.01%</td>
</tr>
</tbody>
</table>
5.5.2 Standard Application Benchmarks

5.5.2.1 AIM7

The AIM7 database workload is executed on each of the kernels. On the SPF implemented kernels, the security checks in the `read()` and `seek()` system calls are disabled. Disabling the security in these system calls corresponds to a database in which the integrity of the database is the primary security concern, not its confidentiality. We expect `read()` and `seek()` system calls to be executed heavily in such databases because read and seek operations are required to execute database queries. Disabling security in the `read()` and `seek()` system calls correlate to setting the System-SPF Configuration `fs.spf` to 33, Process-SPF Configuration on the benchmark binary to 17, and File-SPF Configuration to 5 on the files used by the benchmark.

The results of executing this benchmark are shown below in Tables 5.7 and 5.8. Table 5.7 indicates the size of the database workload executed and the job throughput observed. Table 5.8 indicates the job throughput degradation SELinux without SPF suffers when compared to Redhat and the percentage performance improvements each of the SPF implemented kernels is able to get compared to SELinux without SPF. The numbers in Table 5.8 are calculated as discussed in Section 5.3.1. To obtain accurate results, each database workload size is executed 5 times on each of the kernels. The numbers from each run are averaged to give us the numbers in Table 5.7.

The numbers in Tables 5.7 and 5.8 approximate the results that we expected from Section 5.4. When the workload in the system is relatively small, we expected very little performance difference between Redhat and SELinux. At workload sizes of 1, 50, and 100, SELinux is actually able to slightly outperform Redhat in this benchmark. However, we attribute this to statistical variation since the execution of a low number of processes may not always be consistent. Once the workload increases to a relatively large size, in particular sizes 150 through 250, we begin to see the expected throughput decrease. The throughput degradation is quite significant, decreasing anywhere from 2.63 to 11.21 percent.
For workload sizes of 150 to 250, SPF is able to improve the performance of the job throughput. Under workload sizes of 150 to 250, the SPF implemented kernels regained 1.97 to 7.97 percent of the throughput. These numbers are consistent with our expectations in Figure 5.1.

<table>
<thead>
<tr>
<th>Workload</th>
<th>Redhat (Jobs/Min)</th>
<th>SELinux No SPF (Jobs/Min)</th>
<th>SELinux System-SPF (Jobs/Min)</th>
<th>SELinux Process-SPF (Jobs/Min)</th>
<th>SELinux File-SPF (Jobs/Min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14.54</td>
<td>15.066</td>
<td>15.124</td>
<td>13.276</td>
<td>15.07</td>
</tr>
<tr>
<td>50</td>
<td>232.62</td>
<td>235.904</td>
<td>234.538</td>
<td>232.46</td>
<td>232.57</td>
</tr>
<tr>
<td>100</td>
<td>273.04</td>
<td>273.35</td>
<td>275.566</td>
<td>273.648</td>
<td>276.367</td>
</tr>
<tr>
<td>150</td>
<td>284.79</td>
<td>277.298</td>
<td>276.658</td>
<td>282.242</td>
<td>279.46</td>
</tr>
<tr>
<td>200</td>
<td>290.09</td>
<td>262.934</td>
<td>268.65</td>
<td>284.158</td>
<td>284.8667</td>
</tr>
<tr>
<td>250</td>
<td>286.1</td>
<td>254.02</td>
<td>269.556</td>
<td>276.84</td>
<td>272.69</td>
</tr>
<tr>
<td>300</td>
<td>135.45</td>
<td>166.938</td>
<td>137.466</td>
<td>155.366</td>
<td>130.0733</td>
</tr>
</tbody>
</table>

Table 5.7: AIM7 Database Throughput Results, read() and seek() security disabled.

<table>
<thead>
<tr>
<th>Workload</th>
<th>SELinux No SPF Job Throughput Degradation</th>
<th>System-SPF Improvement Over SELinux No SPF</th>
<th>Process-SPF Improvement Over SELinux No SPF</th>
<th>File-SPF Improvement Over SELinux No SPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.62%</td>
<td>+0.40%</td>
<td>-12.31%</td>
<td>+0.03%</td>
</tr>
<tr>
<td>50</td>
<td>1.41%</td>
<td>-0.59%</td>
<td>-1.48%</td>
<td>-1.43%</td>
</tr>
<tr>
<td>100</td>
<td>0.11%</td>
<td>+0.82%</td>
<td>+0.11%</td>
<td>+1.12%</td>
</tr>
<tr>
<td>150</td>
<td>-2.63%</td>
<td>-0.23%</td>
<td>+1.74%</td>
<td>+0.76%</td>
</tr>
<tr>
<td>200</td>
<td>-9.36%</td>
<td>+1.97%</td>
<td>+7.32%</td>
<td>+7.56%</td>
</tr>
<tr>
<td>250</td>
<td>-11.21%</td>
<td>+5.43%</td>
<td>+7.97%</td>
<td>+6.49%</td>
</tr>
<tr>
<td>300</td>
<td>23.25%</td>
<td>-21.76%</td>
<td>-8.55%</td>
<td>-27.22%</td>
</tr>
</tbody>
</table>

Table 5.8: AIM7 Database Throughput Improvement, read() and seek() security disabled.

An interesting measurement is discovered when we look at the performance of this benchmark at a workload of 300. At this workload point, SELinux without SPF is able to outperform Redhat and all of the SELinux kernels with SPF. In addition, the performance difference is not small. SELinux without SPF is able to outperform Redhat and the SELinux with SPF kernels by over 20 percent.

At first glance, the measurement at a workload size of 300 looks like an outlier. However, we will continue to see a similar pattern of results with the remainder of the benchmarks in this
Chapter. We will see SELinux performance degradation lessen or even outperform Redhat at the highest workload sizes. The explanation for this effect is discussed in Section 5.6.

5.5.2.2 WebStone

Apache web server 2.0.39 is executed on the test machine so that the test machine becomes a dedicated web server. WebStone web clients are executed on a separate machine. The other machine is a dual 1667 MHz Athalon machine with 2 Gigabytes of RAM.

Before executing the WebStone benchmark, the appropriate SPF Configuration is set on the test machine. In the SELinux with System-SPF kernel, the $fs.spf$ system tunable is set to 13. This value disables read(), stat(), and open() system call security checks. Since a web server is expected to have almost entirely public information, this disables all security checks related to the protecting the confidentiality of web server data. In order to benchmark Process-SPF performance with WebStone, the Apache web server binary is set with a Process-SPF Configuration of 13. This value also disables read(), stat(), and open() system call security checks.

The web server throughput results from this benchmark are shown below in Tables 5.9 and 5.10. Each WebStone client size is executed 5 times on each kernel so that accurate results can be obtained. The results from each client size and kernel are averaged to give us the numbers in Table 5.9. Table 5.9 indicates the number of clients that stressed the web server and the resulting throughput when the benchmark is executed on Redhat, SELinux without SPF, SELinux with System-SPF, and SELinux with Process-SPF. As was mentioned in Section 5.3, WebStone will not be executed under SELinux with File-SPF because web server workloads are not suitable for execution under File-SPF. Table 5.8 indicates the web server throughput degradation that occurs in SELinux when compared to Redhat. It also lists the percentage performance improvements System-SPF and Process-SPF are able to contribute. These improvement numbers are calculated with the formulas indicated in Section 5.3.1.
Both SELinux with System-SPF and SELinux with Process-SPF are able to gain performance improvements over SELinux without SPF for all workload sizes greater than 1. The throughput improvements range from several tenths of a percentage to over 3 percent. These numbers may seem small, but they are quite significant when they are analyzed relative to the size of typical web server workloads. A heavily loaded web server may have to handle billions of web page requests per second. A small percentage increase in throughput can make a big difference for such a server. A small percentage increase in throughput allows a web server to handle millions of additional requests per second.

<table>
<thead>
<tr>
<th>Clients</th>
<th>Redhat (bytes/sec)</th>
<th>SELinux No SPF (bytes/sec)</th>
<th>SELinux System-SPF (bytes/sec)</th>
<th>SELinux Process-SPF (bytes/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>869612</td>
<td>870441</td>
<td>877709</td>
<td>860724.5</td>
</tr>
<tr>
<td>10</td>
<td>84224.6</td>
<td>83798.8</td>
<td>83957.7</td>
<td>83885.53</td>
</tr>
<tr>
<td>20</td>
<td>45504.4</td>
<td>45089.6</td>
<td>45386</td>
<td>45373.96</td>
</tr>
<tr>
<td>30</td>
<td>31451.1</td>
<td>30271.8</td>
<td>31404.8</td>
<td>31328.4</td>
</tr>
<tr>
<td>40</td>
<td>24348.6</td>
<td>23865.4</td>
<td>23959.7</td>
<td>23991.67</td>
</tr>
</tbody>
</table>

Table 5.9: WebStone Throughput Results, `read()`, `stat()`, and `open()` security disabled.

<table>
<thead>
<tr>
<th>Clients</th>
<th>SELinux No SPF Throughput Degradation</th>
<th>SELinux System-SPF Improvement Over SELinux No SPF</th>
<th>SELinux Process-SPF Improvement Over SELinux No SPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.10%</td>
<td>+0.83%</td>
<td>-1.12%</td>
</tr>
<tr>
<td>10</td>
<td>-0.51%</td>
<td>+0.19%</td>
<td>+0.11%</td>
</tr>
<tr>
<td>20</td>
<td>-0.91%</td>
<td>+0.65%</td>
<td>+0.62%</td>
</tr>
<tr>
<td>30</td>
<td>-3.75%</td>
<td>+3.60%</td>
<td>+3.36%</td>
</tr>
<tr>
<td>40</td>
<td>-1.98%</td>
<td>+0.38%</td>
<td>+0.51%</td>
</tr>
</tbody>
</table>

Table 5.10: WebStone Throughput Improvement, `read()`, `stat()`, and `open()` security disabled.

At the highest workload size of 40 clients, notice that the throughput degradation of SELinux relative to Redhat performance is significantly less than the throughput degradation when the client size is 30. Although SELinux did not have higher throughput than Redhat, like AIM7 did at a workload size of 300, the numbers suggest that SELinux without SPF can outperform Redhat if the workload size is increased further. Despite this measurement, the web server benchmark
still approximates our expectations from Section 5.4. For most of the workload sizes, SPF is able to decrease the throughput loss suffered by a web server running on SELinux.

It is interesting to note how the results in Table 5.9 are far different than the results gathered in [LOSCOCCO01A]. In [LOSCOCCO01A], WebStone was also executed on SELinux to determine how much SELinux security affected web server throughput. In their results, when 32 web clients were executed, SELinux had web server throughput decrease of only 0.12% when compared to Redhat. This result is far different from the results shown in Table 5.9. The difference comes from the file distributions used in [LOSCOCCO01A] and this thesis. [LOSCOCCO01A] used the standard WebStone file distribution, while this thesis used a completely different file distribution. In our opinion, the standard WebStone file distribution is very unrepresentative of real web server workloads. Therefore, a different file distribution has been developed for this thesis. The details of the standard WebStone distribution and the distribution developed for this thesis are discussed in Appendix A.

5.5.3 Original Application Benchmarks

5.5.3.1 MPEG Frame Rate

The MPEG frame rate benchmark is executed with a workload running in the background of the Berkeley MPEG player. The Berkeley MPEG player attempts to play a small MPEG video at 30 frames per second and 60 frames per seconds. The small video clip runs for 30 seconds long when run at 30 frames per second. The average frame size of this video clip is 2882 bytes.

We expect a significant frame rate decrease for this benchmark because the frame size is quite small. The SELinux overhead of every read from disk should be relatively large. To increase the frame rate, the security checks in the \texttt{read()} system call will be disabled. In System-SPF, this corresponds to setting the \texttt{fs.spf} system tunable to 1. In Process-SPF, the Berkeley MPEG player Process-SPF Configuration is set to 1 using the \texttt{set_bin_spf} command. In File-SPF, configurations are set to 1 on the MPEG movie file using the \texttt{set_file_spf} command.
The frame rate measurements from this benchmark are shown below in Figures 5.11 through 5.14. Tables 5.11 and 5.13 indicate the workload size executed in the background and the measured frame rate when the MPEG file is executed on the 5 different kernels. Tables 5.12 and 5.14 show the frame rate degradation that SELinux without SPF suffers when compared to Redhat. Tables 5.12 and 5.14 also show the improvements each of the SPF implementations gave. The numbers in 5.12 and 5.14 are calculated as discussed in Section 5.3.1. To obtain accurate results, each benchmark workload size is executed 10 times on each kernel. The frame rate measurement from each of these runs is averaged to give us the numbers in Figures 5.11 and 5.13.

Tables 5.31 through 5.34 show how MPEG frame rate is affected with the addition of SPF into SELinux. For each of the SPF implementations, the frame rate is affected very little between Redhat and SELinux when the workload is 0, 5, or 10. Therefore, SPF had little ability to improve MPEG video frame rate for those workload sizes.

We do not see a significant frame rate decrease in SELinux until the workload executing in the background is 15. When the benchmark attempted to play the MPEG video at 30 FPS, the average frame rate is approximately 0.8 less frames per second. When the Berkeley MPEG player tried to play at 60 FPS, the average frame rate decreased by over 2 frames per second. At workload sizes of 15, SPF is able to gain back approximately half the frames lost due to security in SELinux. At 30 FPS, SPF is able to regain approximately 0.4 frames per second and at 60 FPS SPF is able to regain a little over 1 frame per second.

These decreases in frame rate are quite small, so they may not seem like much. However, these small differences in frame rate may make a difference in a QoS aware operating system. In a QoS system, these differences in frame rate may make the difference in a MPEG video being schedulable or un-schedulable. The frame rate increase SPF brings will give the video a better chance of being schedulable.
Table 5.11: MPEG Frame Rate, 30 Frames/Second, read() security disabled.

<table>
<thead>
<tr>
<th>Workload (frames/sec)</th>
<th>Redhat (frames/sec)</th>
<th>SELinux No SPF (frames/sec)</th>
<th>SELinux System-SPF (frames/sec)</th>
<th>SELinux Process-SPF (frames/sec)</th>
<th>SELinux File-SPF (frames/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31.5714221</td>
<td>31.9548801</td>
<td>31.9362</td>
<td>31.87586</td>
<td>31.92471</td>
</tr>
<tr>
<td>5</td>
<td>32.6630158</td>
<td>32.8379257</td>
<td>32.83387</td>
<td>32.81257</td>
<td>32.84367</td>
</tr>
<tr>
<td>10</td>
<td>32.8575414</td>
<td>32.80736</td>
<td>32.79403</td>
<td>32.77317</td>
<td>32.79073</td>
</tr>
<tr>
<td>20</td>
<td>4.7472983</td>
<td>4.9486752</td>
<td>4.600971</td>
<td>5.002774</td>
<td>4.806086</td>
</tr>
</tbody>
</table>

Table 5.12: MPEG Frame Rate Improvement, 30 Frames/Second, read() security disabled.

<table>
<thead>
<tr>
<th>Workload (frames/sec)</th>
<th>SELinux No SPF Frame rate Decrease</th>
<th>SELinux System-SPF Improvement Over SELinux No SPF</th>
<th>SELinux Process-SPF Improvement Over SELinux No SPF</th>
<th>SELinux File-SPF Improvement Over SELinux No SPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.21%</td>
<td>-0.05%</td>
<td>-0.25%</td>
<td>-0.09%</td>
</tr>
<tr>
<td>5</td>
<td>0.54%</td>
<td>-0.02%</td>
<td>-0.08%</td>
<td>+0.01%</td>
</tr>
<tr>
<td>10</td>
<td>-0.15%</td>
<td>-0.04%</td>
<td>-0.11%</td>
<td>-0.05%</td>
</tr>
<tr>
<td>15</td>
<td>-7.88%</td>
<td>+3.26%</td>
<td>+4.74%</td>
<td>+3.47%</td>
</tr>
<tr>
<td>20</td>
<td>4.24%</td>
<td>-7.32%</td>
<td>+1.14%</td>
<td>-3.00%</td>
</tr>
</tbody>
</table>

Table 5.13: MPEG Frame Rate, 60 Frames/Second, read() security disabled.

<table>
<thead>
<tr>
<th>Workload (frames/sec)</th>
<th>Redhat (frames/sec)</th>
<th>SELinux No SPF (frames/sec)</th>
<th>SELinux System-SPF (frames/sec)</th>
<th>SELinux Process-SPF (frames/sec)</th>
<th>SELinux File-SPF (frames/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>50.0356416</td>
<td>49.9469018</td>
<td>49.90746</td>
<td>49.95166</td>
<td>49.97162</td>
</tr>
<tr>
<td>5</td>
<td>49.6785326</td>
<td>49.8051139</td>
<td>49.66672</td>
<td>49.73647</td>
<td>49.75259</td>
</tr>
<tr>
<td>10</td>
<td>49.4209304</td>
<td>48.8996599</td>
<td>49.0372</td>
<td>49.49178</td>
<td>48.75685</td>
</tr>
<tr>
<td>15</td>
<td>12.7768424</td>
<td>10.3065058</td>
<td>11.36492</td>
<td>11.70975</td>
<td>11.55966</td>
</tr>
<tr>
<td>20</td>
<td>4.9600751</td>
<td>5.8672039</td>
<td>5.677815</td>
<td>6.052625</td>
<td>5.502688</td>
</tr>
</tbody>
</table>

Table 5.14: MPEG Frame Rate Improvement, 60 Frames/Second, read() security disabled.
Surprisingly, for both 30 FPS and 60 FPS, SELinux has a better frame rate than Redhat when the workload is 20. This is similar to the results we saw in the AIM7 database benchmark. The highest workload under both of these benchmarks exhibited higher performance in SELinux without SPF than Redhat or any of the SPF implemented kernels.

Overall, it seems that SPF can help improve MPEG frame rate, but not too much. Unlike the previous benchmarks, SPF does not help increase frame rate for a very wide range of workload conditions. It seems MPEG frame rate can only be improved under specific workload conditions. In most situations, the specific system load will not be known. Therefore, SPF may not be too useful for providing increased frame rate for MPEG video.

5.5.3.2 Video on Demand Frame Rate

Similarly to the MPEG frame rate benchmark in the previous section, the Video On Demand benchmark measures video frame rate. However, it will measure frame rate on multiple video streams, not just one. Just like the MPEG benchmark, the Video On Demand benchmark tries to play videos at 30 frames per second and 60 frames per second. The simulated video clips in this benchmark runs for 2 minutes, if played at 30 frames per second, and have a frame size of 2000 bytes.

Just like the MPEG benchmark, the security checks in the read() system call will be disabled for this benchmark. By disabling the security checks in the read() system call, we hope the average video streaming frame rate can be increased. In System-SPF, we set the fs.spf system tunable to 1, using the sysctl command. In File-SPF, configurations are set to 1 on the simulated video files, using the set_file_spf command. In Process-SPF, the benchmark binary is set to 1, using the set_bin_spf command.

The results of the Video On Demand benchmark are shown below in Tables 5.15 through 5.18. Tables 5.15 and 5.17 display the number of video streams executed and the average frame rate the video streams are played at. As was mentioned in Section 5.3, the benchmark outputs the
frame rate of each video stream played in the benchmark. Therefore, the numbers in Tables 5.15 and 5.17 are the average frame rate measured for all the video streams. To obtain accurate results, each benchmark workload size is executed 10 times on each kernel. The average frame rate from each execution is averaged to give us the numbers in Tables 5.15 and 5.17.

Tables 5.16 and 5.18 show the average frame rate degradation that SELinux without SPF suffers when compared to Redhat. Tables 5.16 and 5.18 also show the attempted frame rate improvements of each of the SPF kernel implementations. The numbers of Tables 5.16 and 5.18 are calculated as discussed in Section 5.3.1.

The Video on Demand benchmark shows the most interesting data of any of the benchmarks. As we might expect, SELinux security has very little effect on the frame rate of the video streams played when the number of streams executed is relatively low. For 300 streams or lower at 30 FPS, the security of SELinux does not seem to affect the frame rate of the video streams being played. Similarly, for 100 streams or lower at 60 FPS, SELinux security does not seem to affect frame rate of the video streams. This does not mean that the frame rate does not decrease. When the number of video streams increases in the system, the average frame rate of each video stream will decrease. However, SELinux security does not yet play a role in the frame rate decrease. The frame rate decrease is due to the number of video streams being processed.

It is not until the workload is at 400 streams, for 30 FPS, and 200 streams, for 60 FPS, that we see some remarkable results. The frame rate of the benchmark, when executed in SELinux without SPF, is actually able to outperform Redhat. When the maximum number of streams, 500, is executed, the average frame rate in SELinux without SPF is as high as 23% better than Redhat’s average frame rate. This is now the third instance we have seen SELinux without SPF perform better than Redhat when the benchmark is executed at its highest workload. The reason for this will be discussed in Section 5.6.

This data suggests that SPF may not be very useful for Video On Demand services. If SELinux is able to outperform Redhat at higher workloads, it may not be necessary to implement SPF at all in systems dedicated to video streaming. The results in Tables 5.15 and 5.17 suggest that there are only a few specific instances that SPF may increase video streaming frame rate.
<table>
<thead>
<tr>
<th>Streams (frames/sec)</th>
<th>Redhat (frames/sec)</th>
<th>SELinux System-SPF (frames/sec)</th>
<th>SELinux Process-SPF (frames/sec)</th>
<th>SELinux File-SPF (frames/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25.0069</td>
<td>25.009</td>
<td>25.0085</td>
<td>24.0092</td>
</tr>
<tr>
<td>100</td>
<td>24.6897</td>
<td>24.6697</td>
<td>24.6901</td>
<td>24.6793</td>
</tr>
<tr>
<td>200</td>
<td>22.1947</td>
<td>22.4275</td>
<td>22.4552</td>
<td>22.38</td>
</tr>
<tr>
<td>300</td>
<td>15.5161</td>
<td>15.5094</td>
<td>15.8172</td>
<td>16.5509</td>
</tr>
<tr>
<td>500</td>
<td>5.14118</td>
<td>6.3393</td>
<td>5.79032</td>
<td>5.96556</td>
</tr>
</tbody>
</table>

Table 5.15: VOD Frame Rate, 30 Frames/Second, read() security disabled.

<table>
<thead>
<tr>
<th>Streams (frames/sec)</th>
<th>SELinux No SPF Frame rate Decrease</th>
<th>SELinux System-SPF Improvement Over SELinux No SPF</th>
<th>SELinux Process-SPF Improvement Over SELinux No SPF</th>
<th>SELinux File-SPF Improvement Over SELinux No SPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.01%</td>
<td>-0.42%</td>
<td>-2.00%</td>
<td>-5.94%</td>
</tr>
<tr>
<td>100</td>
<td>-0.08%</td>
<td>+10.99%</td>
<td>+2.23%</td>
<td>-0.06%</td>
</tr>
<tr>
<td>200</td>
<td>1.05%</td>
<td>+4.48%</td>
<td>-2.80%</td>
<td>+12.10%</td>
</tr>
<tr>
<td>300</td>
<td>-0.04%</td>
<td>+0.00%</td>
<td>+0.00%</td>
<td>+0.00%</td>
</tr>
<tr>
<td>400</td>
<td>13.84%</td>
<td>-0.03%</td>
<td>-0.01%</td>
<td>-0.02%</td>
</tr>
<tr>
<td>500</td>
<td>23.30%</td>
<td>-0.09%</td>
<td>-0.80%</td>
<td>-1.46%</td>
</tr>
</tbody>
</table>

Table 5.16: VOD Frame Rate Improvement, 30 Frames/Second, read() security disabled.

<table>
<thead>
<tr>
<th>Streams (frames/sec)</th>
<th>Redhat (frames/sec)</th>
<th>SELinux System-SPF (frames/sec)</th>
<th>SELinux Process-SPF (frames/sec)</th>
<th>SELinux File-SPF (frames/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>49.73026</td>
<td>49.74769</td>
<td>49.75622</td>
<td>49.76562</td>
</tr>
<tr>
<td>100</td>
<td>42.84741</td>
<td>42.78885</td>
<td>42.68881</td>
<td>42.38365</td>
</tr>
<tr>
<td>400</td>
<td>6.60393</td>
<td>7.099228</td>
<td>7.82482</td>
<td>7.24657</td>
</tr>
<tr>
<td>500</td>
<td>4.719973</td>
<td>5.301158</td>
<td>5.512473</td>
<td>5.168863</td>
</tr>
</tbody>
</table>

Table 5.17: VOD Frame Rate, 60 Frames/Second, read() security disabled.
<table>
<thead>
<tr>
<th>Streams</th>
<th>SELinux No SPF Frame rate Decrease</th>
<th>SELinux System-SPF Improvement Over SELinux No SPF</th>
<th>SELinux Process-SPF Improvement Over SELinux No SPF</th>
<th>SELinux File-SPF Improvement Over SELinux No SPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.04%</td>
<td>+0.01%</td>
<td>+0.03%</td>
<td>+0.02%</td>
</tr>
<tr>
<td>100</td>
<td>-0.14%</td>
<td>-0.23%</td>
<td>-0.94%</td>
<td>-1.60%</td>
</tr>
<tr>
<td>200</td>
<td>3.90%</td>
<td>+0.48%</td>
<td>-0.03%</td>
<td>-2.82%</td>
</tr>
<tr>
<td>300</td>
<td>9.67%</td>
<td>-0.42%</td>
<td>-2.00%</td>
<td>-5.94%</td>
</tr>
<tr>
<td>400</td>
<td>7.50%</td>
<td>+10.99%</td>
<td>+2.23%</td>
<td>-0.06%</td>
</tr>
<tr>
<td>500</td>
<td>12.31%</td>
<td>+4.48%</td>
<td>-2.80%</td>
<td>+12.10%</td>
</tr>
</tbody>
</table>

Table 5.18: VOD Frame Rate Improvement, 60 Frames/Second, read() security disabled.

5.6 Security Performance Flexibility Performance Analysis

The benchmark results from Section 5.5 indicate that SPF offers the potential to increase performance. The results indicate that the degree to which it can offer performance increases range from small to moderate percentage increases, depending on the benchmark and the workload size. The results also indicate that there are no performance advantages between System-SPF, File-SPF, and Process-SPF. The numbers indicate that the overhead of each implementation is approximately equal. As was expected, the overhead of SPF is far less than the overhead incurred by SELinux security checks. The difference in overhead is how we are able to obtain performance improvements for a number of our benchmarks.

A common pattern is exhibited with the standard and original benchmark applications from Section 5.5. When the workload executed in the benchmark is low, there is a small performance decrease between SELinux and Redhat. As the workload increases, the performance difference between SELinux and Redhat grows. This is exactly what we expected when we discussed our performance expectations in Section 5.4.

However, as the workload increases even more, the difference between SELinux and Redhat performance seems to decrease. For example, in the WebStone benchmark, we saw that the throughput difference between SELinux and Redhat decreases as the number of clients grew
from 30 to 40. Although the numbers are not shown here, the other AIM7 benchmark workloads also exhibit this pattern.

In other benchmarks, SELinux is able to outperform Redhat at very high workloads. We saw this occur in the AIM7 Database benchmark, the MPEG benchmark, and the Video On Demand benchmark. At the highest workloads in those benchmarks, SELinux is able to outperform Redhat.

Figure 5.2: Performance of SELinux relative to Redhat, Large Workload Range.

Figure 5.3: Performance of SELinux relative to Redhat, Small Workload Range.
These numbers suggest that the performance of SELinux when compared to Redhat is not like the solid line in Figure 5.1. Instead, the performance is much more like the solid line in Figure 5.2. We originally expected the SELinux performance degradation to grow increasingly worse than the performance in Redhat. However, the benchmark results in Section 5.5 suggest that SELinux is able to outperform Redhat at high workload points.

Despite the realization that workloads and SPF did not perform as we expected, SPF is still able to offer performance improvements for certain workloads. The AIM7 Database benchmark and the WebStone web server benchmark results indicate that performance can be increased for a wide range of workload sizes. The AIM7 Database benchmark and WebStone showed that SPF offers performance enhancements similar to the dotted line in Figure 5.2.

However, the solid line in Figure 5.2 is not the situation that always occurred. There are occasions where the results resembled the dashed line in Figure 5.3. The dashed line in Figure 5.3 suggests that there are only a handful of workload points in which SPF can offer significant performance improvements. In this situation, SPF may not be very useful.

As an example, let us consider a Video On Demand server. Suppose SPF is able to offer performance improvements to this Video On Demand server when the number of streams is between 200 and 600 streams. Suppose this Video On Demand server always streams between 300 and 700 streams. In this situation, SPF will be able to provide frame rate increases the majority of the time the server is streaming videos. This workload range is illustrated with the brace in Figure 5.2. The brace indicates that a relatively large amount of workload sizes may benefit from SPF. Under this situation, the benefits of SPF may outweigh the security issues and system administration complexity posed by the SPF implementation.

However, if the range of workload sizes is instead represented by the brace in Figure 5.3, SPF may not be useful for the workload. Suppose in our example Video On Demand server, SPF is only able to improve frame rate when the workload is between 550 and 700. Now, there is very little opportunity for SPF to improve frame rate. In most cases, SPF will not be able to improve frame rate. In this particular situation, SPF may not be able to outweigh the security issues and
system administration complexity involved with SPF. Therefore, SPF may not be desirable in such a system.

The MPEG benchmark and Video On Demand benchmark results are similar to the dashed line from Figure 5.3. It indicates that SPF can only be useful for a small few workload situations. There is very little opportunity for SPF to actually make a performance difference in the workload. It is quite interesting that both of the multimedia benchmarks indicate that SPF may not be very useful. Multimedia is one of the initial workloads that SPF targeted for improvement. The numbers suggest that perhaps multimedia workloads are more dependent on scheduling algorithms and quality of service (QoS) for their performance, rather than the amount of time it takes to read a frame from disk.

Finally, let us discuss why high workloads in SELinux are able to outperform Redhat. It seems counterintuitive that SELinux can ever outperform Redhat. At higher workloads, security checks will take up an increasing percentage of resources, such as CPU cycles and disk bandwidth. Higher workloads in SELinux should lead to higher percentage decreases in performance when compared to Redhat.

We believe the scheduling algorithm in the SELinux kernel may be reason for this performance anomaly. Every scheduling algorithm is tuned to optimize some metric in an operating system, such as fairness, throughput, or turnaround time [SILBERSCHATZ98]. Now that there are SELinux security mechanisms in the Linux kernel, the original Redhat scheduling algorithm is no longer properly tuned to maximize performance. Therefore, SELinux may be able to outperform Redhat under specific workload conditions, because the scheduling algorithm may be tuned in SELinux’s advantage. For example, under certain workloads, context switching or CPU throughput may be the major bottlenecks in the system. Since security mechanisms in the kernel take longer in SELinux than in normal Redhat, fewer context switches may occur while kernel work is executing in parallel with I/O requests. If the workload mix is perfect, fewer context switches may occur in SELinux and thus to higher throughput. These situations should be considered highly dependent on workloads and specific scheduling algorithms. However, they must be taken into account.
CHAPTER 6

Conclusions and Future Work

6.1 Summary and Conclusion

It has been shown that a number of different workloads, applications, or services may not need or may not desire all of the security placed in a Trusted Operating System. If these programs do not require all of the security put inside a Trusted Operating System, performance can be degraded because security checks can take a significant number of CPU cycles away from the executing program.

This thesis proposes the Security Performance Flexibility (SPF) framework as a means to improve performance in Trusted Operating Systems. SPF allows a system administrator to disable security checks within certain parts of the operating system. By giving the system administrator the ability to modify the security mechanisms placed inside a Trusted Operating System, programs can improve their performance.

There are several issues surrounding SPF. In particular, SPF may add additional system administrative complexity. Therefore, it is important that SPF be easy for system administrators to setup and maintain. There must also be assurances that SPF will not open security holes within the Trusted Operating System. If such holes occur, the security benefits that Trusted Operating Systems possess could be eliminated.

SPF can be implemented at different levels in a Trusted Operating System. We have identified three different levels SPF can be implemented in a Trusted Operating System: system wide SPF, process based SPF, and object based SPF. There are advantages and disadvantages of implementing SPF in each of these levels. Particular levels have distinct advantages in terms of ease of system administration, security, or the specific workloads the SPF can apply to. Overall,
each of the three methods have advantages over others in certain systems and environments. The best implementation is highly dependent on the type of system SPF is implemented in.

Chapter 5 went over the results of various benchmarks to see if SPF can really improve the performance of Trusted Operating Systems. The benchmarks indicate that SPF can offer some performance improvements for Trusted Operating System workloads. In some cases, the performance improvements seem to be small. However, the numbers should be looked at in the overall scope of system performance. For example, the results show that web server throughput can increase by several percentage points. For large web servers, this small improvement can allow several million additional web page transactions to occur per second. The MPEG benchmark only showed small frame rate improvements. However, this can make the difference between a video being schedulable or un-schedulable in a QoS aware system.

The data from some of the benchmarks indicate that SPF can only improve performance under specific workload conditions. This is most likely due to an apparent scheduling anomaly seen between SELinux and Redhat. Due to this scheduling irregularity, SELinux is able to outperform Redhat at high workload sizes. Therefore, SPF may only be useful for systems with known workloads sizes. This hurts the overall potential use of SPF. For SPF to make a significant performance improvement across a large range of workloads, the scheduling algorithm placed in SELinux may have to be retuned from its Redhat counterpart.

None of the specific SPF implementations (System-SPF, Process-SPF, File-SPF) show performance advantages over the others. While there is an overhead associated with each type of SPF implementation, the results indicate the overhead of SPF is much smaller than the overhead for security checks. Each of the SPF implementations gives approximately the same amount of performance improvement.

Overall, the conclusion of this thesis is that SPF can be very useful in Trusted Operating Systems. It can help balance security and performance needs of an organization. The system administration does not seem to be terribly difficult and is relatively safe from opening security holes. However, SPF will only be useful depending on the specific needs of an organization.
6.2 Future Work

The study of SPF can be taken far more in depth that what has been done in this thesis. As has been mentioned before, SPF is only implemented within the file system of SELinux. This can be taken further and done with networking or even IPC. Implementing SPF within the networking subsystem of a Trusted Operating System may help decrease latency of network messages.

In addition, the implementation of System-SPF, Process-SPF, and File-SPF in separate kernels is intentionally done so that we can study the issues surrounding each implementation. Future work can see if combinations of these implementations can be integrated into one kernel. In addition, user studies can be done to determine what the best possible implementation of SPF should be.

The security configuration used in this thesis is the default configuration given with the SELinux distribution. However, for some systems, the security configuration may be far more complicated or far simpler. This can have an impact on the depth at which performance can decrease. As was stated in Section 2.1.3, some systems have the ability to cache recently accessed security permissions. Depending on the complexity of the security configuration, software caching may play a more significant or less significant role in performance of a Trusted Operating Systems.

As was mentioned throughout this thesis, there seems to be scheduling irregularities between SELinux and Redhat. This apparent scheduling irregularity causes SELinux to outperform Redhat under certain workload conditions. This suggests that the Linux scheduling algorithm used in Redhat should be retuned under a highly secure environment. Future work can be done to analyze scheduling issues for highly secure operating systems and how scheduling algorithms might be modified.

For the most part, this thesis should be considered a evaluation of the possibilities for SPF in Trusted Operating Systems. At the time of this writing, Trusted Operating Systems are still not
widely used. The hope is that when they do become popular, the ideas in this thesis can be considered for implementation.

Since SELinux is itself a Linux distribution used only for research, education, and evaluation, the code done for this thesis will probably have little real-world use. There is indication that the SELinux project at the NSA will be discontinued in the near future because legal issues concerning open source software being released by the government [LEMOS].

6.3 Future Projects

In the author’s opinion, it takes time for researchers from different research areas to realize that work from both areas can be put together. For example, systems researchers are beginning to realize that artificial intelligence techniques may be useful for system adaptation. While researching for this thesis, many similarities were discovered between QoS and Trusted Operating Systems.

The architecture for several QoS aware operating systems and Trusted Operating Systems are quite similar. For example, [MIYOSHI01] discusses resource control lists, which are very similar to access control lists. [BRUNO99] and [BRUSTOLONI99] discuss how QoS is put into an operating system by creating a separate file system that stores QoS information for each application in a system. This separate file system is similar to how SELinux creates a separate table to store security label information. There are indications that the architectural ideas from QoS and Trusted Operating Systems can perhaps be integrated into one system together. This leads to the possibility that several of these features can be combined and implemented into a single kernel.
APPENDIX A

Benchmark Specifications

Default values are used unless stated below.

AIM9

Each test in AIM9 is executed for 500 seconds. The choice of 500 seconds is somewhat arbitrary. However, it allows the benchmark to finish in a reasonable amount of time and it ensures minimal statistical variance.

MPEG Video Quality

In Linux, processes are raised to the highest priority in the system if that process displays graphics to the X-windows console. This means a MPEG video player will play video flawlessly in Linux, even if the system is under high load. In order to make the benchmark realistic, all processes in the benchmark (including those running in the background) are given equal priorities.

WebStone

In WebStone, each web client makes requests to a web server based on a distribution of files it is given. The file list and distribution used to stress Redhat and SELinux is shown in Figure A.1. The names of the files indicate the size of the file requested. The numbers next to the file name indicates the file’s weight in the distribution. The file list and the distribution of requests in Figure A.1 approximate the numbers shown in [ARLITTT97]. In particular, the median file size of this distribution is 5K and the mean file size is approximately 15K. WebStone actually comes with the standard file distribution list shown in Figure A.2. However, based on the data from [ARLITTT97], it is not representative of real web server file distributions. Every WebStone client
in the benchmark is executed for sixty minutes. Sixty minutes is selected because it allows the benchmark to finish in a reasonable amount of time and it ensures little statistical variance.

<table>
<thead>
<tr>
<th># @(#)filelist</th>
<th>/file16.html 100</th>
<th>/file42k.html 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>/file0.html 500</td>
<td>/file17k.html 100</td>
<td>/file43k.html 100</td>
</tr>
<tr>
<td>/file100.html 500</td>
<td>/file18k.html 100</td>
<td>/file44k.html 100</td>
</tr>
<tr>
<td>/file200.html 500</td>
<td>/file19k.html 100</td>
<td>/file45k.html 100</td>
</tr>
<tr>
<td>/file300.html 500</td>
<td>/file20k.html 100</td>
<td>/file46k.html 100</td>
</tr>
<tr>
<td>/file400.html 500</td>
<td>/file21k.html 100</td>
<td>/file47k.html 100</td>
</tr>
<tr>
<td>/file500.html 500</td>
<td>/file22k.html 100</td>
<td>/file48k.html 100</td>
</tr>
<tr>
<td>/file600.html 500</td>
<td>/file23k.html 100</td>
<td>/file49k.html 100</td>
</tr>
<tr>
<td>/file700.html 500</td>
<td>/file24k.html 100</td>
<td>/file100k.html 10</td>
</tr>
<tr>
<td>/file800.html 500</td>
<td>/file25k.html 100</td>
<td>/file200k.html 10</td>
</tr>
<tr>
<td>/file900.html 500</td>
<td>/file26k.html 100</td>
<td>/file300k.html 10</td>
</tr>
<tr>
<td>/file1k.html 1000</td>
<td>/file27k.html 100</td>
<td>/file400k.html 10</td>
</tr>
<tr>
<td>/file2k.html 1000</td>
<td>/file28k.html 100</td>
<td>/file500k.html 10</td>
</tr>
<tr>
<td>/file3k.html 1000</td>
<td>/file29k.html 100</td>
<td>/file600k.html 10</td>
</tr>
<tr>
<td>/file4k.html 1000</td>
<td>/file30k.html 100</td>
<td>/file700k.html 10</td>
</tr>
<tr>
<td>/file5k.html 1000</td>
<td>/file31k.html 100</td>
<td>/file800k.html 10</td>
</tr>
<tr>
<td>/file6k.html 1000</td>
<td>/file32k.html 100</td>
<td>/file900k.html 10</td>
</tr>
<tr>
<td>/file7k.html 1000</td>
<td>/file33k.html 100</td>
<td>/file1m.html 5</td>
</tr>
<tr>
<td>/file8k.html 1000</td>
<td>/file34k.html 100</td>
<td>/file2m.html 5</td>
</tr>
<tr>
<td>/file9k.html 1000</td>
<td>/file35k.html 100</td>
<td>/file3m.html 5</td>
</tr>
<tr>
<td>/file10k.html 100</td>
<td>/file36k.html 100</td>
<td>/file4m.html 5</td>
</tr>
<tr>
<td>/file11k.html 100</td>
<td>/file37k.html 100</td>
<td>/file5m.html 5</td>
</tr>
<tr>
<td>/file12k.html 100</td>
<td>/file38k.html 100</td>
<td>/file10m.html 1</td>
</tr>
<tr>
<td>/file13k.html 100</td>
<td>/file39k.html 100</td>
<td>/file20m.html 1</td>
</tr>
<tr>
<td>/file14k.html 100</td>
<td>/file40k.html 100</td>
<td></td>
</tr>
<tr>
<td>/file15k.html 100</td>
<td>/file41k.html 100</td>
<td></td>
</tr>
</tbody>
</table>

Figure A.1: Thesis WebStone File Distribution.

<table>
<thead>
<tr>
<th># @(#)filelist.standard 1.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>file500.html 350</td>
</tr>
<tr>
<td>file5k.html 500</td>
</tr>
<tr>
<td>file50k.html 140</td>
</tr>
<tr>
<td>file500k.html 9</td>
</tr>
<tr>
<td>file5m.html 1</td>
</tr>
</tbody>
</table>

Figure A.2: Standard WebStone File Distribution.
APPENDIX B

Benchmark Implementation or Modification

MPEG Video Quality

The Berkeley MPEG player used in this benchmark is slightly modified. The Berkeley MPEG player uses the `fopen()`, `fread()`, and `fwrite()` library routines to deal with file I/O. At the time of this writing, SELinux did not have security implemented for these library routines. Therefore, the MPEG player has been ported to use the `open()`, `read()`, and `write()` system calls respectively. Technically, this modification can reduce the performance of the MPEG player due to the overhead the system calls incur. However, the overhead differences between library calls and system calls will have little influence when the system is highly stressed. In addition to this modification, a handful of minor adjustments have been made so testing can be facilitated.

QDDA is a benchmark comprised of Latex formatting, C++ compilation, and numerical computation. The normal QDDA benchmark has been modified in several ways for this thesis. First, the C++ compilation has been removed from this benchmark. The C++ compilations are removed because C++ compilations are scheduled irregularly in Linux. Removing the C++ compilations ensures that each benchmark run is executed under nearly identical situations. Second, the benchmark executables have been modified to generate a higher workload. The original benchmark would have executed and finished within several seconds, which is not suitable for this benchmark. A higher workload is generated in two ways. First, the computations in the executables have been increased. The main body of each executable has been copied numerous times to increase the length of time each executable runs. Second, identical QDDA tests are executed concurrently to put a higher workload on the system.

In addition, to ensure that all MPEG video tests are executed under identical situations, all disk-caching affects have to be eliminated. A small program has been created that writes and
then reads a very large file from disk. This executable should eliminate caching effects between each run of the MPEG benchmark.

**Video on Demand Quality**

Due to the security restrictions placed on University machines, a Video On Demand benchmark cannot be programmed similarly to WebStone. The Video On Demand benchmark created for this thesis instead simulates a Video On Demand server. The benchmark simulates a Video On Demand system by spawning a set of video stream processes. Each process reads a video file frame by frame from disk as if the video is being streamed. However, the video frames read from disk are not displayed on a screen or sent out on a socket. The frame is ignored by the benchmark. The frame rate that each video stream is logged into a file that can be analyzed later.

**WebStone**

The WebStone benchmark is executed through a master process, called the webmaster, and a set of slave processes, called web clients. The webmaster remotely executes web client binaries on other machines. Each client is responsible for stressing a web server with a particular file distribution of requests. Once the client completes its set of requests, timing and throughput measurements are sent back to the master process. The master process collects these results and processes them accordingly.

The master process remotely executes clients through the `rexec()` call. Due to University security restrictions, remote execution cannot be done on the machines available. Therefore, all web clients are executed on a single machine. Some problems exist with this method. In particular, the client machine may become a bottleneck, leading to inaccurate results. While there are other potential ways to execute this benchmark without the benefits of remote execution, executing all web clients on a single machine is the best option. No modifications need to be made to the web client binary so that it can execute on one machine. Only small modifications have been made to the WebStone code to make statistics gathering easier.
REFERENCES


(ARGUS01A) Argus Systems Group, Inc. OS-level security for Solaris and AIX, 2001


[MPEGPLAY] Berkeley MPEG player.
http://bmrc.berkeley.edu/frame/research/mpeg/mpeg_play.html.


http://www.mindcraft.com/webstone/.