The Pennsylvania State University

The Graduate School

APPLYING NETLABEL TO NETWORK ACCESS CONTROL IN A VIRTUALIZED ENVIRONMENT

A Thesis in

Computer Science and Engineering

by

Radhesh Kamath

© 2008 Radhesh Kamath

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science

August 2008

The thesis of Radhesh Kamath was reviewed and approved* by the following:

Trent Jaeger

Associate Professor of Computer Science and Engineering

Thesis Adviser

Sencun Zhu

Assistant Professor of Computer Science and Engineering

Raj Acharya

Professor of Computer Science and Engineering

Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

We consider the problem of transmitting authorization data in a distributed environment comprised of applications hosted on paravirtualized Virtual Machines. The current mechanism, called Labeled IPSec, relies upon IPSec to provide secure communication channels to applications, while implicitly transferring authorization data between communication endpoints. But, Labeled IPSec incurs heavy performance penalties and hinders the scalability of Mandatory Access Control of Network Communication, because it does not leverage the underlying mechanisms offered by the hypervisor and Virtual Machine kernels to provide secure communication channels. We exploit the Netlabel mechanism provided by the Linux Kernel to transmit authorization data in packets, while leveraging the Trusted Computing Base comprised of the Xen hypervisor, paravirtualized guest kernels, and the Dom-0 to isolate and protect network communication. In order to test our claim that the Netlabel mechanism is more efficient than Labeled IPSec, we build a prototype that consists of modified guest kernels and Dom-0 kernel, along with user space tools in the Dom-0 which enable monitoring and dynamic control of the packet labeling state on guests. Using micro benchmarks that measure the impact of labeling on performance measures such as latency and bandwidth, we provide a quantitative comparison of Labeled IPSec and Netlabel in various communication scenarios. We find that Netlabel consistently outperforms Labeled IPSec on all performance measures, validating our claim that authorization data can be transmitted at lesser cost using Netlabel, and relying on the TCB to protect network traffic in transit. As a result, while we conclude that Netlabel can be applied successfully to provide Mandatory Access Control of Network Communication, we also believe that more extensive performance analyses, better support for security-aware applications and a standardized policy management interface will go a long way toward securing applications.

Table of Contents

List of Figures vi

List of Tables viii

Acknowledgments ix

Chapter 1
Introduction 1
1.1 Virtualization and systems security . . . 1
1.2 Access Control . . . 3
1.3 Layered Security Architecture . . . 6
1.4 System Description . . . 7
1.4.1 Local Communication and Remote Communication . . . 8
1.5 IPSec for Network Mandatory Access Control . . . 10
1.6 Proposed method for Network Mandatory Access Control . . . 13
1.7 Problem Statement . . . 14
1.8 Thesis Contributions . . . 15

Chapter 2
Background 17
2.1 SELinux . . . 17
2.1.1 LSM Framework . . . 17
2.1.2 Flask Security Architecture . . . 18
2.1.3 SELinux Implementation in the Linux Kernel . . . 18
2.2 CIPSO . . . 20
2.3 Netlabel . . . 21
2.3.1 Netlabel LSM Interface . . . 21
2.3.2 Netlabel Hooks . . . 22
2.3.3 Security Server . . . 22
2.3.4 Netlabel Command Handling . . . 22
2.4 Xen . . . 25
2.4.1 Dom-0 . . . 25

Chapter 3
Related Work 27
3.1 Security Administration Manager (SAM) . . . 27

3.2 SESAME . . . 28
3.3 IBM DCE . . . 32
3.4 Tivoli Access Manager . . . 35
3.5 Script Execution . . . 37
3.6 VM Monitoring: XenAccess . . . 39
3.7 sHype . . . 41
3.8 Labeled IPSec . . . 43
3.9 Selopt . . . 45

Chapter 4
Implementation **48**
4.1 Netlabel userspace tools . . . 48
4.2 Netlabel Subsystem in Kernel . . . 50
4.3 Xend and xm Implementation . . . 50
4.3.1 Xend actions for Domain Creation and Startup . . . 51
4.4 Need for modified userspace tools . . . 52
4.5 Need for Modified Guest Kernel . . . 53
4.6 Need for Modified Guest Kernel Interface . . . 54
4.7 Modifications to Guest Kernel Interface . . . 54
4.8 Netlabeld Implementation . . . 55
4.8.1 Data Structures used by Netlabeld . . . 57
4.8.2 Adding Initial DOIs . . . 58
4.9 Modifications to `libxenctrl` . . . 67
4.9.1 Changes to `alloc_magic_pages` . . . 68
4.9.2 Guest Mapping Updates . . . 70
4.9.3 Copying CIPSO Mappings into the Guest's Memory . . . 74
4.9.4 Sending Guest Details to Netlabeld . . . 75

Chapter 5
Results **82**
5.1 Experiment Setup . . . 82
5.2 Netlabel processing overhead . . . 84
5.3 Using Netlabel to Carry Authorization Data . . . 90
5.4 Using IPSec in transport mode to carry authorizations . . . 92
5.5 Conclusions and Further Work . . . 94

Bibliography **101**

List of Figures

1.1 Local Communication (only one direction shown) . . . 9
1.2 Communication with remote VM (only one direction shown) . . . 9
1.3 Using IPSec tunnels for authorization . . . 11
1.4 Encoding labels in packets for authorization . . . 13

2.1 CIPSO Tag Type-1 . . . 21
2.2 netlabel cipsov4 command . . . 23
2.3 netlabel map command . . . 24

3.1 SESAME Architecture . . . 30
3.2 Tivoli Access Manager Architecture . . . 35
3.3 Free Form Tag . . . 46
3.4 Named Tags used by Selopt . . . 46

4.1 `shared_info` structure (`include/xen/interface/xen.h`) . . . 55
4.2 netlabeld main loop . . . 56
4.3 cipso_table structure . . . 57
4.4 cipso_list structure . . . 57
4.5 cipso_v4_doi structure . . . 58
4.6 doidef_list structure . . . 58
4.7 add_cipso_libxc_data function . . . 62
4.8 setup_event_handling function . . . 63
4.9 write_dom0_evtchn_shinfo function . . . 65
4.10 Modified version of `xc_linux_build` function (`tools/libxc/xc_dom_compat_linux.c`) 67
4.11 Additions to `xc_dom_image` structure (`tools/libxc/xc_dom.h`) . . . 68
4.12 Modified function `alloc_magic_pages` (`tools/libxc/xc_dom_x86.c`) . . . 69
4.13 Pages allocated to guest (4kB pages) . . . 70
4.14 Modified `arch_setup_bootlate` function. (`tools/libxc/xc_dom_x86.c`) . . . 72
4.15 Modified `shared_info_x86_32` function. (`tools/libxc/xc_dom_x86.c`) . . . 75
4.16 `setup_cipso_page` function. (`tools/libxc/xc_dom_x86.c`) . . . 76
4.17 Modified `xc_dom_release` function. (`tools/libxc/xc_dom_core.c`) . . . 79
4.18 `send_nlbl_dom_data` function. (`tools/libxc/xc_dom_core.c`) . . . 80

5.1 Network Topology . . . 83
5.2 Unlabeled local communication . . . 85
5.3 Local communication with Netlabel . . . 86
5.4 Unlabeled remote communication . . . 86

5.5 Remote communication with Netlabel 86
5.6 TCP connection latency, local communication 87
5.7 TCP connection latency, remote communication 87
5.8 TCP latency, local communication 87
5.9 TCP latency, remote communication 88
5.10 UDP latency, local communication 88
5.11 UDP latency, remote communication 88
5.12 TCP Bandwidth, local communication 89
5.13 TCP Bandwidth, remote communication 89
5.14 Remote communication with Labeled IPSec,tunnel mode throughout 91
5.15 Remote communication with Vanilla IPSec,tunnel mode between Dom-0's 91
5.16 Remote communication with Labeled IPSec,tunnel mode between Dom-0's 91
5.17 Connection latency, remote communication,tunnel mode 92
5.18 TCP latency, remote communication,tunnel mode 92
5.19 UDP latency, remote communication,tunnel mode 93
5.20 TCP bandwidth, remote communication,tunnel mode 93
5.21 End-to-end local communication, vanilla IPSec+Netlabel 94
5.22 End-to-end remote communication, vanilla IPSec+Netlabel 94
5.23 End-to-end local communication, Labeled IPSec 95
5.24 End-to-end remote communication, Labeled IPSec 95
5.25 Connection Latency for end-to-end IPSec Connection, local communication 96
5.26 Connection latency for end-to-end IPSec Connection, remote communication 96
5.27 TCP Latency for end-to-end IPSec Connection, local communication 97
5.28 TCP latency for end-to-end IPSec Connection, remote communication 97
5.29 UDP Latency for end-to-end IPSec Connection, local communication 97
5.30 UDP latency for end-to-end IPSec Connection, remote communication 98

List of Tables

2.1 Selinux hooks used by Netlabel 22
2.2 Security Server component of Netlabel 23

4.1 Message types used by netlabeld 56
4.2 Architecture-dependent Hooks - x86 guest with PAE enabled shown (Modified Hooks in Red) 68

Acknowledgments

I would like to thank my advisor, Prof. Trent Jaeger, for giving me a valuable learning opportunity through this thesis. His extraordinary patience, hands-off management style, and constant willingness to discuss problems with me have made this thesis possible. Thank you, Professor, for your kindness and support. Thanks to Prof. Sencun Zhu, who has given his valuable time and effort to officiate as my committee member.

A special thanks to the following members of the Systems and Internet Infastructure Security Laboratory, who have contributed to this work: Yogesh Sreenivasan, who helped me set up performance measurement experiments; Luke St.Clair, who patiently answered many technical questions; Divya Muthukumaran, who allowed me to use her machine to run experiments.

I have had the unflinching support and wise counsel of many friends who have enriched my life at Penn State, a few of whom I mention by name: Arjun Nath, Ravichandra Sundararaj and Reetuparna Das. Thank you, folks — it has been a privilege to be your friend.

I owe a huge debt of gratitude to my uncles, R.G.Nagaraj and R.G.Chandrashekhar, who taught me to read, write, and brought me up as their own son. Thanks to my kid brother, Suhas Kamath, who has taken care of my parents while I have been abroad. Perhaps the greatest credit for my graduation goes to my younger sister, Rashmi Kamath, who beat me to commencement, both as an undergraduate *and* a graduate!

Finally, I thank my parents, Swarna and Muralidhara Kamath – without their support, motivation and love, life would have been difficult indeed.

Chapter 1

Introduction

1.1 Virtualization and systems security

Virtualization has been with us for decades now, being almost as old as "conventional" time-sharing, starting with the CP-40 systems designed by IBM. A Virtual Machine (VM), defined as "a hardware-software duplicate of a real computer system in which a statistically dominant subset of the virtual processor's instructions execute on the host processor in native mode" [4], provides a "multi-environment" system that enables multiple Operating Systems to run simultaneously. The Virtual Machine Monitor (VMM) is responsible for managing the "real resources" provided by the machine for use by the VMs. Donovan compared VM systems to conventional timesharing systems (e.g., Multics), in [18], concluding that while conventional timesharing systems promote the use of shared programs and data, they must also support a wider variety of interfaces. In contrast, VMMs are required to provide a small set of basic "hardware" interfaces.

The security properties of VMMs were investigated further by Donovan et. al. in [19]. The security properties listed below are still true for most VM systems and provide a strong rationale for their utility

as a security mechanism:

- **Higher number of protection levels:** A VM operates under at least two levels of protection, usually three: The VMM runs in the highest protection ring, with the VM kernel running in the next (lower) ring and the applications on the VM run in the least privileged ring. In contrast, conventional time-sharing systems usually operate with two protection levels, with the kernel operating at the higher level. Thus, applications are subject to three levels of protection in a VM system, as opposed to two on a conventional system.

- **Redundant security mechanisms:** The introduction of the VMM layer leads to additional security mechanisms that operate *independently* of higher layers. The memory protection mechanisms employed by VM/370 are a case in point. The OS/360 (VM) protects applications from one another using the "lock-and-key" mechanism, while the VM/370 (VMM) protects individual VMs from one another using Virtual Memory mechanisms. Therefore, malicious applications will have to subvert both the VM's and VMM's memory protection mechanisms to compromise the whole system.

- **Improved isolation:** The improved isolation between applications running on a VM is the result of the two properties described above. Applications that require substantially differing types of support from the operating system can be run within different VMs. As a result, each VM can be much simpler, providing the bare minimum of the specialized interfaces and services required. Further, the VMM is inherently simple, owing to the restricted interface it provides. The result is that the probability of a security violation *of the entire system* drops significantly. First, the simplicity of an individual VM reduces the probability of its compromise. Second, the simplicity of the VMM reduces the probability of its being compromised. The joint probability of an overall system compromise arising out of a single malicious application is much smaller than in the case of a conventional system.

In addition to their superior security properties, VMs reduce the total cost of ownership by increasing re-

source utilization, make provisioning easier, improve flexibility via VM migration capabilities and permit a high degree of service consolidation. Furthermore, hardware improvements have reduced the overhead associated with virtualization, leading to the proliferation of open-source and proprietary VM offerings, e.g., Xen, VMWare, OpenVZ, VirtualBox etc. Therefore, virtualization finds compelling reasons to be deployed in many scenarios. For example, virtualization has been used to provide scalable web services, test software, execute untrusted code, improve system reliability, provide redundancy in services, run legacy applications and to package software. Thus, it is safe to say that VMs are now ubiquitous and their usage will only grow in the future.

1.2 Access Control

Many popular Operating Systems such as BSD, Windows and Linux have support for *Discretionary Access Control* (DAC) policies. DAC policies are specified by the owner of the data and defined over three groups - owner, group and world. Access checks simply refer to the Access Control List (ACL), over the data and also to the identity (a numerical value) of the process accessing the data to verify group membership, and therefore, authorization. In contrast, Mandatory Access Control (MAC) policies are defined over the entire system and usually are *not subject to change by individual users*. Furthermore, MAC policies are enforced by a *Reference Monitor* that *completely mediates every access*. The notion of the Reference Monitor – a body of trusted, verified code that consults the system wide policy to obtain authorization on *every* resource access – and the *Trusted Computing Base* (TCB) – the Reference Monitor and the hardware and/or software components *outside it* – that are sufficient to enforce the security policy, was first introduced in [1].

MAC operating systems label all the subjects and objects with *labels* that implicitly convey authorization. Policy changes (e.g., changing labels or changing authorization associated with a label) can be made only by the system administrator. Traditionally, MAC systems have focused on confidentiality of data in the

system, but it is desirable that they also enforce the principles of *least privilege* and *privilege escalation*, thus reducing the impact of vulnerabilities of programs that might be exploited on other programs and the system as a whole. Until recently, the prevalent MAC policy model was the *Bell-LaPadula* (BLP) model. BLP policies seek to restrict information flow by enforcing the *simply security property and the *-property* that do not allow read-up and write-down respectively, thus ensuring confidentiality of data. The *-property recognizes the distinction between a user granting access and a program doing so on the user's behalf – a Trojan might violate confidentiality by writing down.

Although BLP has fit the needs of the intelligence community well, it has found little application in commercial scenarios. The *Multi-Level* view of the world, required by BLP, emphasizes confidentiality at the cost of other security criteria e.g., data integrity, policy flexibility, etc.; BLP is deemed too restrictive to be of practical use to a wide audience. Although other ways of modeling Security Policy (e.g., Role-based access control, Type Enforcement) have existed for a long time, MAC security support in Operating Systems was restricted to "Trusted Operating Systems" that catered exclusively to the needs of the U.S. military establishment – they were Multi-level secure systems.

In this networked world , applications and operating systems are growing more complex, offering users more opportunities to avail of services (e.g., carry out monetary transactions) and share their data (e.g., access medical records) over networks. Sadly, bugs, vulnerabilities and configuration errors in both applications and operating systems have also increased with the resulting complexity. As a result, there has been an explosion in malicious code in the form of worms, trojans, viruses etc., that seek to compromise confidentiality and/or integrity of the data stored on systems, or in transit over the network, using ever-increasing network bandwidth and anonymity provided by the Internet. In [17], the authors note the magnitude of the threat and argue that providing security at the application level is doomed to fail, unless there is strong support for MAC policy enforcement at the operating system level. In particular, userspace access control mechanisms can be subverted by malicious code that either bypasses the enforcement component completely or impersonates the user or changes the security attributes on the user's

behalf covertly. Cryptography without underlying operating system protection is not very useful either, as the cryptographic library itself is stored on the filesystem, and executes in memory, both of which have to be protected from tampering and leaks. The most important conclusion of [17] that bears repetition here, is that the most important security feature lacking in DAC systems is the *trusted path*. The trusted path provides a secure, tamper proof and non-bypassable channel between the user and the *Trusted Computing Base*. The trusted path, along with Mandatory Access Control, are seen as necessary to confine malicious applications, and to protect the confidentiality and integrity of data on the system. *In the context of downloaded (untrusted) code and highly distributed applications hosted in individual VMs, the trusted path expands to include paths that traverse (possibly open-access) networks as well. Thus, MAC of network communication plays a crucial role in securing applications.*

As a result of the renewed interest in security policy models and support for MAC policy in operating systems, efficient and flexible MAC implementations are available in several "mainstream" operating systems. Linux implements the Linux Security Module (LSM) framework that supports "Security Modules", similar to Linux Kernel Modules. LSMs enable administrators to express and enforce a wide variety of MAC policies. For instance, the SELinux LSM provides support to express subjects and objects as "domains" and "types" respectively, and define valid transitions between them ("Domain and Type Enforcement"); in addition, SELinux supports other models of access control, such as *Role-based access control*. The SMACK (Simple Mandatory Access Control Kernel) LSM provides support to express MLS labels on files and simple policies for access control. Further, many distributions of Linux, e.g., RedHat , Fedora etc., have SELinux enabled by default and also come packaged with a number of user space tools such as SLIDE (SELinux Policy Generation), `auditd` (an audit logger), `libselinux` (library that can be used by security-aware applications) etc. and predefined default policies for hundreds of applications. These factors have brought MAC security, once the province of "trusted systems", into the mainstream.

1.3 Layered Security Architecture

Although Virtualization and MAC offer strong security benefits, when applied to large, distributed systems, they contribute to policy complexity. First, the security policy might become unmanageably large, impeding scalability of enforcement, and therefore, of the entire system. Second, the overall usability of the system is restricted because of restraints on information sharing between VMs — this has discouraged many users from adopting MAC security. Finally, as a consequence of policy complexity, it becomes difficult to reason clearly about the overall security provided by the system. [25] examines this problem, and proposes an architecture whose purpose is to reduce policy complexity, bringing it down to manageable proportions, making MAC possible in a large, distributed system.

The architecture proposed in [25] reduces policy complexity at individual levels by increasing the granularity of resources subject to MAC. The hypervisor is responsible mainly for multiplexing hardware among VMs and can "see" information flows among VMs, so it is responsible for enforcing access control policy at the VM level. This is achieved by labeling individual VMs and associating authorizations with them within the hypervisor. At the OS level, access control becomes more fine-grained, with information flow between confined processes being enforced by MAC systems within the VM. Finally, security-aware applications might further restrict flows via language-based security mechanisms. Multi-labeled VMs are trusted to ensure that there are no flows between processes that are labeled differently. Further, the paper also describes how label translations might occur at interfaces between the hypervisor and VMs. This architecture serves as the basis of our system; we describe relevant features in greater detail as and when needed in subsequent sections.

1.4 System Description

Before offering a concrete statement of the problems we address through our work, we describe our system at a high level. We also present two network communication scenarios and show how Network Mandatory Access Control can be achieved using existing methods. Next, we discuss the disadvantages of using existing methods and motivate the need for new mechanisms to achieve Network Mandatory Access Control.

The system under consideration is a set of VMMs that host distributed applications. The VMMs together enforce security policies that are defined for *coalitions*, composed of individual VMs. Possession of *coalition label* by a VM implies membership. Each VM might belong to more than one coalition, and each VMM might run VMs from different coalitions. The shared reference monitors (VMMs) that enforce coalition policy are known as a *Shamon*. In the Shamon, security mechanisms presuppose the existence of *secure channels* that provide confidential communication among the principals involved. Our objective is to provide *efficient* secure channels that are subject to *fine-grained* Mandatory Access Control, which we will call *Network Mandatory Access Control*.

In our system, the VMM in consideration is the Xen Virtual Machine Monitor [3] and it runs paravirtualized Linux Virtual Machines, known as domains or guests. Each VMM runs a privileged domain, known as Dom-0 and several unprivileged domains (DomU's). The Dom-0 has access to special interfaces to the hypervisor that are used to perform privileged operations on behalf of the guests. Coalition policies are enforced at the VMM level by `sHype`, the reference monitoring component of the Xen hypervisor. sHype controls access to real resources at a coarse-grained level (e.g., `grant table and event-channel` level).

We illustrate the desired reference monitoring of network communication in such a setup through two cases: "local" communication and "remote" communication.

1.4.1 Local Communication and Remote Communication

Consider the scenario illustrated in figure 1.1 (one direction shown for clarity). In this case, a network application on the domain `green_vm` communicates with another application on domain `bluegreen_vm`. In a typical Xen installation, packets are transferred through the `frontend` of the network device driver to the `backend`, in Dom-0. Packets are multiplexed over the physical device `peth0` by the backend, which has exclusive access. Packets pass through `virbr`, which is a virtual bridge device layered over `peth0`. Further, packets undergo protocol processing up until the IP layer of the stack in Dom-0, wherein some basic sanity checks are performed on the packets. At the IP layer, the packet is either handed off to the higher layer of the protocol stack, if destined for Dom-0, or forwarded, if destined for another host.

By default, most installations include minimal firewalling functionality via `iptables` that utilizes the netfilter infrastructure in the Dom-0 kernel. Packet filtering usually occurs at the bridge level and also at the network level.

In this case, the packet sent out by `green_vm` is transferred using the Xen split-driver mechanism to the backend, where it is filtered at the bridge and network levels. The solid red lines in figure 1.1 denote the secure channel.

Consider the scenario illustrated in figure 1.2. This case differs from local communication in that the recipient is hosted on another VMM. Another difference is that the VMMs themselves are on different physical hosts. The solid red lines in figure 1.2 denote the secure channel.

From the use cases discussed, we make the following observations:

- **Scope of Network Policy:** The coalition concept implies that several domains enforce the same network communication policy. For example, for the access decision on a purely internal kernel object, e.g., an inode, MAC policy at the domain level is sufficient, as the objects in question have a completely local scope. In contrast, for *network objects*, MAC policies for *both* the peers have to

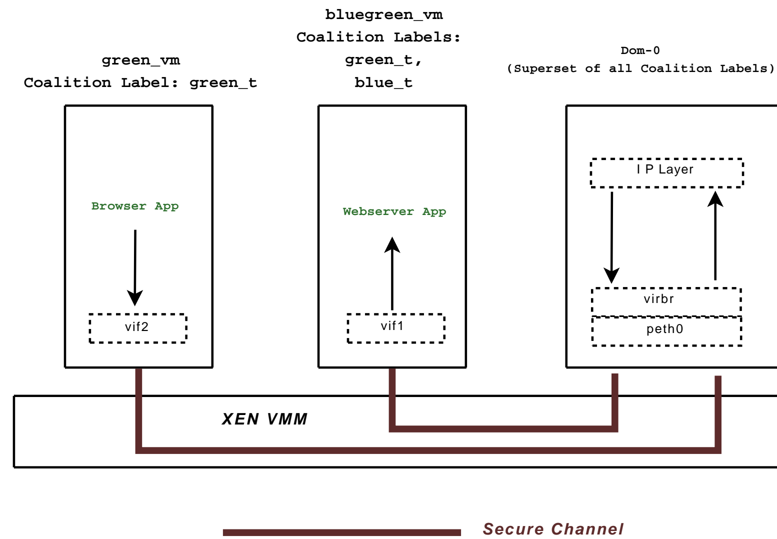

Figure 1.1. Local Communication (only one direction shown)

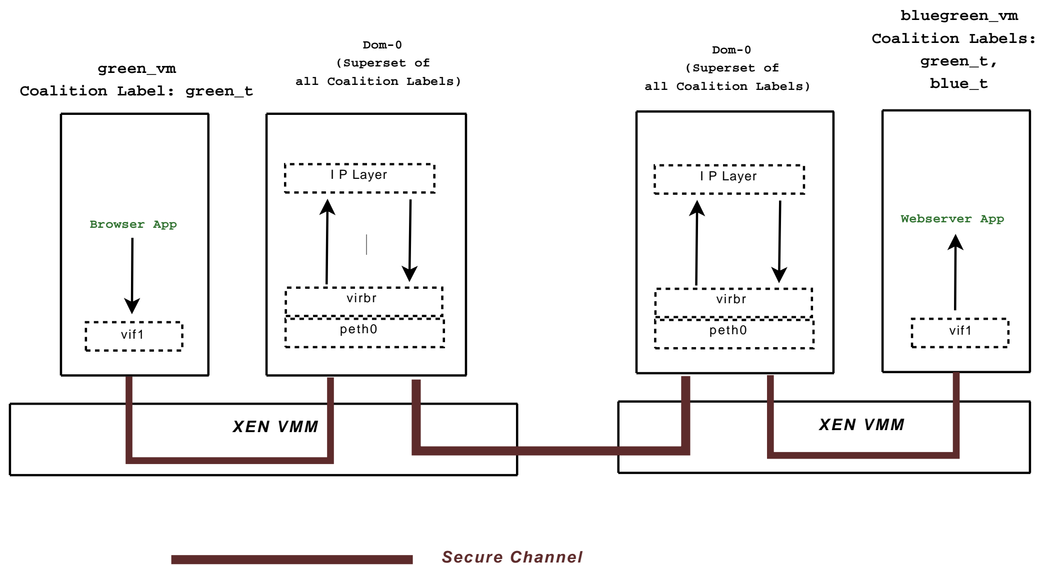

Figure 1.2. Communication with remote VM (only one direction shown)

agree — this indicates that Network Policy is necessarily global in nature.

- **Nature of Access Control Mechanisms:** In our setup, access control mechanisms are necessarily distributed in nature, because enforcement responsibilities lie with many entities. As a result, enforcement must occur at all points along the path traversed by packets to be complete. Domains have to mediate access to resources visible only to them - e.g., internal resources such as inodes that

are shared on an *intra-domain* basis. VMMs have to mediate access to resources visible to them -e.g., resources such as *event-channels* that are shared on an *inter-domain* basis. As a result, while access to resources such as inodes and event-channels are *completely mediated* by the domain and VMM respectively, complete mediation for network resources occurs only with the involvement of *four* entities at the same time: the unprivileged domains that are peers, the Dom-0 which has to ensure enforcement of the global network policy, and also the Hypervisor, that is responsible reference-monitoring the inter-domain communication that occurs during network communication.

- **Exchange of contextual information:** No reference-monitoring component has *all* the information needed for an access decision in our system. In addition, the requisite information changes with time. The principals involved in the communication are dynamic, as is the policy. For example, a socket peer may no longer exist, or the set of rights associated with the peer might have changed with time. Further, the scope of some of the objects accessed in the course of network communication is local, restricting the visibility of their security attributes. Thus, the security attributes of subjects and objects in network communication are *contextual* in nature. We observe that Network Access Control requires knowledge of contextual security attributes of the principals involved. *Therefore, the most important function of secure channels in the Shamon is to transmit security attributes that specify authorization — to provide trusted paths for applications*. Thus, for MAC to work on the distributed system as a whole, Network Access Control becomes *necessary*.

1.5 IPSec for Network Mandatory Access Control

IPSec, described in more detail in section 3.8, is a protocol suite that provides integrity and/or confidentiality guarantees for IP datagrams. In addition, IPSec can also help two peers establish each others' identity using public / secret key cryptography, thus authenticating peers to one another. Further, IPSec can protect all IP traffic between two gateways (Virtual Private Networks).

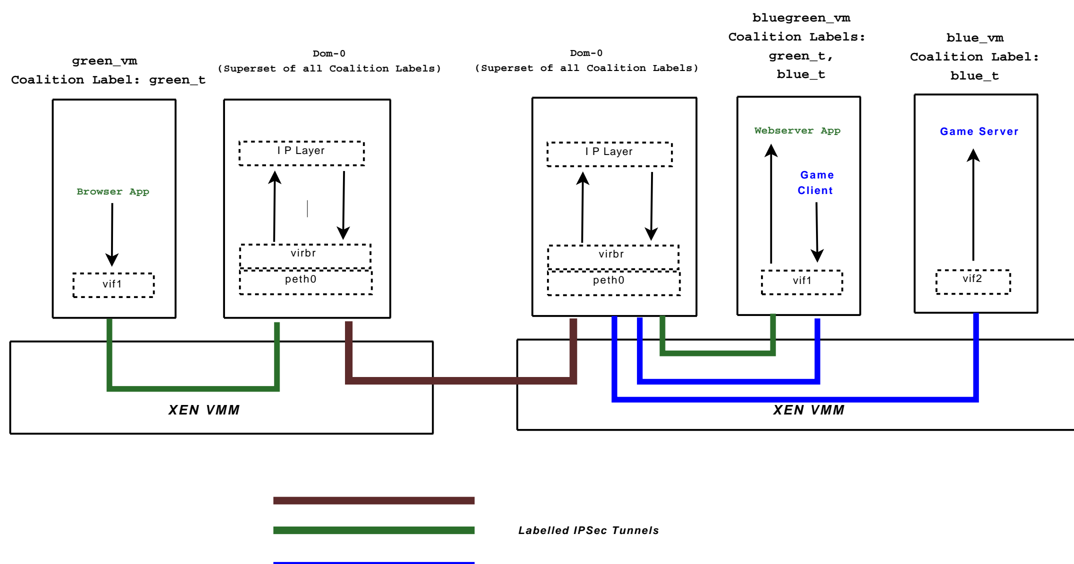

Figure 1.3. Using IPSec tunnels for authorization

Figure 1.3 shows how communication between two applications can be protected in our scenario. Packets in transit between `green_vm` and `bluegreen_vm` can be protected end-to-end (*transport mode*) or by Dom-0, which can serve as a VPN gateway (*tunnel mode*). Userspace software in the guest (VM or Dom-0) supplies the keys that will be used for authenticating the peers. Although IP packets can be sent unencrypted (with support for integrity and anti-replay alone - *AH Mode*), usually, confidentiality services are applied to packets by IPSec (as they might possibly traverse a public network - *ESP mode*). Further, public-key cryptography is usually used in large systems, so that, by integrating with an existing PKI, costs are reduced, and certificate management becomes centralized, making management easier (e.g., revocation). Finally, in the figure 1.3, the IPSec implementation is integrated with the SELinux LSM of the guest, enabling Network Access Control that provides authorization data (implicitly) to the peers. The arrangement shown in figure 1.3 *does achieve* Mandatory Access Control of Network Communication, but it suffers from some severe problems in a distributed, virtualized environment such as ours.

IPSec comes with a performance penalty that places limits on service consolidation in a virtualized, distributed environment. First, *every* network application that runs on *each* VM will have to be negotiate security parameters with its peers. Second, each IPSec connection, if long-lived, will incur the cost of

re-generating session keys for packet encryption. Third, each connection setup will involve extensive communication back-and-forth between the userspace key management tools and the IPSec subsystem in the kernel — this is particularly expensive for a virtualized system. IPSec processing and the concomitant cryptographic operations consume significantly more CPU time than otherwise, thus placing limits on service consolidation. Finally, if Dom-0 is used as a VPN gateway (or it might be the driver domain), then the situation is worsened further, because Dom-0 will have to do IPSec processing we have mentioned, with *all* the peers with which *each* VM communicates.

The trust model that requires IPSec processing is one in which the principal(s) and the network path taken by packets are untrusted. Thus, IPSec will be used to authenticate peers and to encrypt packets in such a scenario, which leads to associated key management overhead and other performance penalties we have described. But, the nature of the system we have described forces a re-evaluation of this trust model. First, the IPSec model of a network does not match that of network communication among VMs hosted on the same VMM. In Xen, network communication between VMs uses memory almost entirely, not a "wire" on which an adversary can monitor all communication, capturing all packets. A network transfer involves a "page flip", i.e., transfer of ownership of the pages that hold the data from the sender to Dom-0, and from Dom-0 to the receiver. Access to this memory area is subject to independent reference-monitoring by the hypervisor. The IPSec trust model does not take this into account. Second, the sender and the recipient are both VMs and network communication is actually a form of *inter-domain* communication, from the hypervisor's point of view. As such, this communication is further subject to independent reference-monitoring by the hypervisor. Finally, the "gateway" is the Dom-0, which is all-powerful in our model, and is completely trusted to isolate VMs from one another and enforce information flows according to a VMM-wide policy. Thus, the IPSec trust model is a complete mismatch to our system.

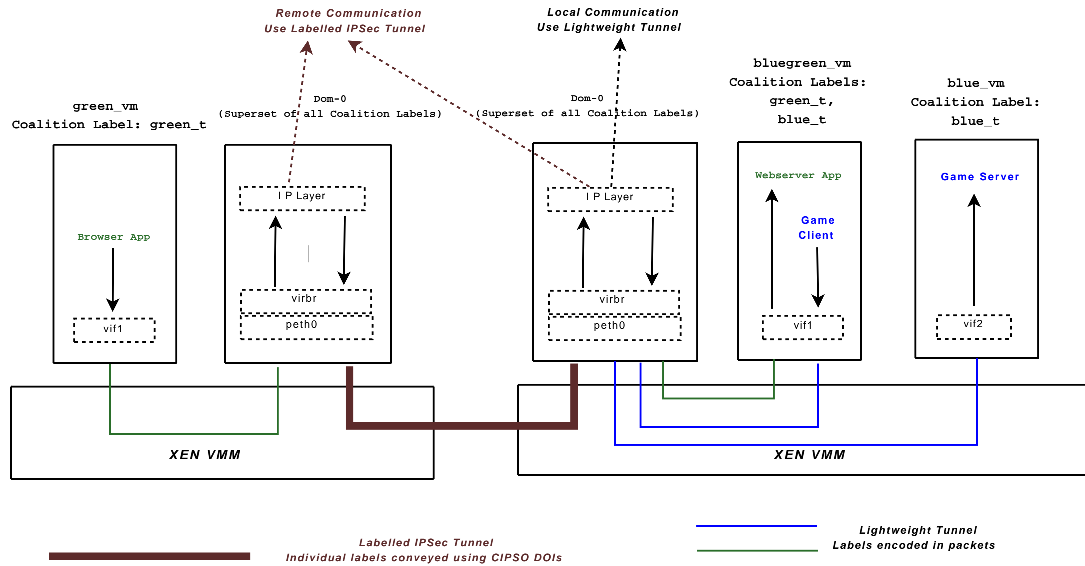

Figure 1.4. Encoding labels in packets for authorization

1.6 Proposed method for Network Mandatory Access Control

Using IPSec for Network Mandatory Access Control in our system is a waste of resources and hinders scaling. We propose an alternative method for Network Mandatory Access Control that leverages existing Xen protection mechanisms and reference-monitoring functionality, that eliminates the resource expenditure associated with IPSec, enabling higher levels of consolidation, while providing the same guarantees as IPSec. The proposed method is shown in figure 1.4. As shown in figure 1.4, VMs that are hosted on the same VMM communicate using a “lightweight tunnel”. This tunnel provides the same confidentiality and integrity guarantees over network data as IPSec, *not by using expensive cryptographic operations*, but by using *existing memory protection and hardware access mechanisms* offered by the hypervisor. For network communication between VMs hosted on *different* VMMs, packets that “leave” the VMM and traverse an open-access network are subject to IPSec processing that ensures their confidentiality and integrity, but they are protected using a lightweight tunnel while in transit between the VM and the Dom-0, that serves as a VPN gateway. Further, to reduce IPSec performance penalty, all flows to a particular VMM from a VM are aggregated in a single IPSec tunnel, thus amortizing authentication and key

re-generation costs over the lifetime of connections between the two Dom-0's, as shown in the figure.

1.7 Problem Statement

Our proposal guarantees the same confidentiality and integrity properties as IPSec for network communication, but in order to achieve Network Mandatory Access Control, *authorization data* must also be transmitted over the secure channels, to provide the same functionality as Labeled IPSec. This motivates the following high-level problems:

1. **How to convey authorization data?** Authorization data is implicitly conveyed by labeled IPSec in the form of SELinux contexts that are needed to access the Security Association. How can this be provided in an aggregated IPSec tunnel to the other endpoint by Dom-0? One of the ways of conveying authorization data would be to include it in individual IP packets. Thus, we need a standard for conveying authorization data that would enable Dom-0's to interoperate seamlessly. Further, this standard must be implemented in the TCP/IP stack of the Linux kernel to "label" packets in an efficient manner.

2. **How to establish correct meanings of system-wide authorization data?** Assume that there is a standard to label packets with authorization data, and it is implemented by the Linux kernel (we describe this in further detail in Chapter 2). The authorization data received in packets must be translated to meaningful security information by the endpoints, for enforcement to occur. It follows, therefore, that for enforcement to be *correct and consistent*, the translation of labels on packets to locally meaningful security attributes (on endpoints) has to be monitored and controlled. Both monitoring and control activities have to leverage trusted code in the endpoints (the unprivileged guests, in our case) and the Dom-0.

3. **How to ensure enforcement of Network Access Controls by Guest VMs?** Finally, provided that we convey authorization data in individual packets efficiently and also ensure that this authorization

data has some meaning within guests, we have to also *ensure that the authorization data is actually used in enforcement*. In the system under consideration, authorization data on individual packets is translated into SELinux contexts, because SELinux is the Mandatory Access Control system that is responsible for enforcement. Thus, provided we know the SELinux policy being enforced at the endpoints, we have to ensure that packet labels are translated into the *correct* SELinux contexts by them. By doing so, we can be reasonably confident that the reference monitors within the (trusted) guest kernels prevent undesirable information flows.

1.8 Thesis Contributions

This thesis makes the following contributions:

1. We identify a low-cost, efficient standard (CIPSO) that can be used for conveying authorization data in individual packets. Further, we also investigate its implementation in the Linux Kernel, the implementation of the userspace tools and the semantics of packet labeling commands that can be used.

2. We build a prototype implementation that enables transparent monitoring of guest VMs' packet labeling policy, and its control from the Dom-0. This prototype leverages the Trusted Computing Base (TCB) comprised of the Dom-0 kernel, the unprivileged guest kernel, the Xen hypervisor and certain userspace tools in Dom-0. Specifically, we re-evaluate the trust model in the context of our system and its requirements, and modify the TCB and certain trusted interfaces between guests and the hypervisor to enable transparent[1] monitoring and control of their packet labeling policy.

3. Using the `lmbench` micro benchmark, we provide quantitative measurements of key characteristics such as latency and bandwidth of network communication in a number of scenarios. We characterize the adverse performance impact of Labeled IPSec, and demonstrate the performance

[1] transparent to applications in unprivileged guests

improvements that are obtained using our proposed method of Network Access Control.

4. We identify further work that should be done to improve the usability of our prototype; additional measurements that could reveal the true performance benefits of our proposed method and finally, improvements in the packet labeling infrastructure in the Linux Kernel and how they might influence performance metrics.

The rest of the thesis is organized as follows: in chapter 2, we examine the relevant elements of the TCB, such as the SELinux Mandatory Access Control system; the implementation of packet labeling and its interface with the SELinux reference monitor and the `Xen` paravirtualization system. In chapter 3, we discuss related work in the areas of distributed Access Control, packet labeling, the Reference Monitoring component in Xen - `sHype` and VM monitoring. In chapter 4, we describe in detail our prototype implementation and the changes required to the TCB. In chapter 5, we present performance measurements that show the cost of Labeled IPSec and compare its performance with that of our proposed solution. We conclude by identifying future work and possible improvements to our system.

Chapter 2

Background

In this chapter, we introduce the basic components of the system, and discuss their features. The material is categorized into: the architectural aspects of Mandatory Access Control in the Linux Kernel and the predominant MAC module in use today - `SELinux`; the CIPSO standard that provides conventions for network-wide object labeling and its implementation in the Linux Kernel, `netlabel`; the interface between netlabel and SELinux; and finally, a brief overview of the `Xen` VMM.

2.1 SELinux

2.1.1 LSM Framework

The Linux Security Module (LSM) framework in the Linux Kernel provides an access control infrastructure; it permits Security Modules to access the kernel data structures and mediate access to them based on security policy. Further, there are 'hooks' - points in the kernel code paths - which can be used by security modules to

1. Allocate security-related attributes of the object in question or
2. Perform reference-monitoring activity, e.g., check whether the subject has the appropriate permissions to access the object in question and return an *access decision*.

2.1.2 Flask Security Architecture

The LSM framework has been heavily influenced by SELinux. SELinux was originally implemented as a service in the *Flask microkernel* [28]. The aim of the Flask security architecture was to provide the maximum possible policy flexibility. To this end, the Flask design separated policy enforcement from policy decision-making, encapsulating the decision-making process within a "Security Server". The Security Server managed the security attributes of all the subjects and objects in the system, which included allocating and deallocating them ("labeling decisions"). The rest of the system was oblivious of the security attributes, only passing them along to the Security Server to obtain access decisions. By assigning all responsibility for managing security attributes and access decisions to the Security Server, the Flask designers ensured policy consistency throughout the entire system, along with other benefits such as To address efficiency concerns, a cache that is situated between the enforcement components and the Security Server was added. This cache, known as the 'Access Vector Cache'(AVC), stores the bit vector of permissions for objects, and is consulted first. Hits in the AVC remove the latency associated with obtaining an access decision from the Security Server. The Security Server is responsible for managing the AVC and keeping it current, e.g., a policy change requires invalidating the AVC.

2.1.3 SELinux Implementation in the Linux Kernel

SELinux [16] was originally released as a patch to the Linux Kernel; at this time, the need for a framework that standardized the interaction between access control code and decision-making code was not felt. The original patch

1. Embedded security-relevant data structures within the kernel data structures (as `void` pointers).
2. Code to allocate and de-allocate these data structures along with the kernel objects.
3. Inserted separate permission checking functions to request decisions from the security server.
4. Introduced new system calls that were extended versions of their Linux counterparts, which processed additional `sid` parameters.

The implementation was revamped with the introduction of the LSM framework. Some of the implementation changes were:

1. Specialized `security` data structures were embedded into the kernel data structures, managed by the Security Server.
2. The extended system calls were dropped, replaced by new calls that set security attributes and then proceeded with the normal Linux system calls.
3. A pseudo-filesystem called `selinuxfs` ,that provided a standardized interface for application-level libraries, rather than extended system calls was introduced.
4. The hooks were moved to the standard Linux permission-checking functions.

`selinuxfs` is used by applications, e.g., `libselinux` ,to interact with the security server in the kernel. It is implemented as a pseudo file system. Selinuxfs can be used for the following purposes:

1. Control the enforcement mode of SELinux: enforcing / permissive
2. Ascertain the details about the current policy, e.g., obtain its version number, obtain the booleans, whether it is MLS-capable ,etc.
3. Selinux context-to-sid conversion or vice-versa.
4. Control the status of booleans in the policy.

5. Get AVC statistics.

2.2 CIPSO

CIPSO (Commercial IP Security Option) is a draft by the IETF that never made it to the status of a standard, but it is still widely used as the de-facto mechanism used to transmit labels in IP packets. Full details about cipso can be found in [7], only the salient features are discussed here.

The CIPSO draft defines a standard that allows hosts to encode labels indicating the sensitivity levels and categories of the data in the IP packet in the Options part of the IP Header. CIPSO provides room for a larger amount of security data to be passed in the header than its precursors, the DoD Basic Security Option (BSO) and the DoD Extended Security Option (ESO).

The CIPSO format defines permits hosts to define "Domains of Interpretation" (DOIs) that establish common meanings for security data encoded in packet. DOIs permit hosts to interoperate and transmit information about sensitive data contained in the packets in a consistent manner at a network-wide level.

The CIPSO option, that is permitted to appear only once in the packet, specifies "tags" that define what follows them: the sensitivity level, and the categories associated with the data in the packet. The encoding of the categories may vary, and this is specified by the tag type, which can be 1, 2, or 5.

Tag Type 1, which is of interest to us, is known as the "restrictive bitmap type" and as the name suggests, the categories are encoded as a bitmap, following an octet that specifies the numerical value of the sensitivity level.

Tag type 1 is shown in figure 2.1.

Tag Types 2 and 5 permit categories to enumerated (e.g., category 12 is here, category 15 is here etc, when the number of categories is very large) and specified by ranges (e.g., highest category = 255, lowest category = 1) respectively.

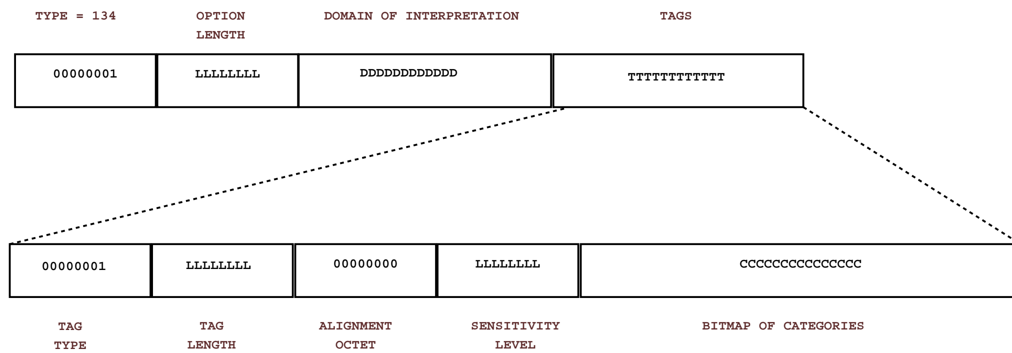

Figure 2.1. CIPSO Tag Type-1

Along with the tag formats, the draft also specifies how to handle unlabeled packets (packets not "labeled" using CIPSO), gateway behavior (gateways must be able to convert from one DOI specification to another, for instance), etc.

2.3 Netlabel

Netlabel is a labeled networking implementation in the Linux Kernel. It consists of a userspace component `netlabelctl` [22] that communicates with the kernel using `netlink` [6, 27] sockets. Netlabel implements the CIPSO standard partially. At present, tag type 1 is supported in Netlabel. Netlabel also has an LSM interface. At present, the Netlabel hooks in the kernel interact with the SELinux Security Server which manages the SELinux security attributes associated with packets and sockets. Also, the Netlabel code encodes the CIPSO options in packets and provides an internal kernel API that can be used to retrieve/fill up encoded options in packets received/sent by the system.

2.3.1 Netlabel LSM Interface

Netlabel has an internal kernel API that exposes Netlabel variables for use by LSMs. The LSM-relevant attributes are present in a structure `netlabel_lsm_secattr`. The SELinux Security Server code does conversions between the SELinux MLS data structures and `netlabel_lsm_secattrs` , while the net-

work part of the CIPSO code (in `net/ipv4/cipso_ipv4.c`) and the internal netlabel kernel API code (in `net/netlabel/netlabel_kapi.c`) does conversions between the security data encoded in the packet and the `netlabel_lsm_secattr` data structures.

2.3.2 Netlabel Hooks

At present, Netlabel uses the following hooks (in `security/selinux/hooks.c`), shown in table 2.1.

Hook Function	***Explanation***
`sk_alloc_security`	Initial allocation of security struct
`selinux_file_permission`	Manage file access to socket
`selinux_socket_post_create`	Socket labeled with sid of task that created it
`selinux_socket_sendmsg`	Write permission on the socket is checked
`selinux_socket_setsockopt`	Setopt permission is checked
`selinux_socket_sock_rcv_skb`	–
`selinux_sk_clone_security`	Used when socket buffer is cloned
`selinux_sock_graft`	–
`selinux_inet_csk_clone`	–

Table 2.1. Selinux hooks used by Netlabel

2.3.3 Security Server

Netlabel also has a security server component (in `security/selinux/ss/services.c`), shown in table 2.3.3.

2.3.4 Netlabel Command Handling

Netlabel commands are used to define DOIs,that in turn contain CIPSO mappings and send them to the Netlabel subsystem in the Kernel. The Netlabel userspace uses Netlink sockets for communication between user and kernel space.

A netlabel command is shown below:

Function	Explanation
`selinux_netlbl_secattr_to_sid`	Conversion of netlabel `secattr` to selinux `sid`
`selinux_netlbl_skbuff_getsid`	Get the `secattr` for the packet and convert it into selinux `sid` to assign to the packet
`selinux_netlbl_skbuff_setsid`	Label the socket using the sid given
`selinux_netlbl_sk_security_init`	Determine whether Netlabel'ing is required
`selinux_netlbl_socket_post_create`	If Netlabel'ing is required, label the socket using selinux `sid`
`selinux_netlbl_sock_graft`	The socket's Netlabel is set to match the `sid` of the network-level `sock` object. The peer's sid is determined by netlabel.
`selinux_netlbl_inode_permission`	Verify that the socket is labeled by Netlabel (`PF_INET` sockets always are)
`selinux_netlbl_sock_rcv_skb`	The packet is inspected for Netlabel security attributes.
`selinux_netlbl_socket_setsockopt`	Controls the `setsockopt()` call.

Table 2.2. Security Server component of Netlabel

Figure 2.2. netlabel cipsov4 command

Thus, when a netlabel DOI definition is added to the kernel, the netlabel subsystem has to do the following:

- Check whether the MLS level specified in the command is less than the maximum legal level.
- Check whether the MLS category specified in the command is less the the maximum MLS category.
- Allocate kernel data structures for each DOI definition that contain the information specified in the `netlabelctl cipsov4 add` command.

Further, Netlabel permits the user to map an LSM domain to a DOI:

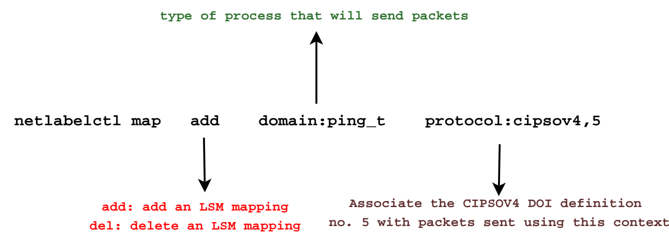

Figure 2.3. netlabel map command

When a netlabelctl map command is issued to the system, the netlabel subsystem must do the following:

- Add the entry to the kernel's list of LSM domain mappings. Note that conversion between LSM and CIPSO levels does not happen at this point.
- Associate the entry with the appropriate DOI definition after some simple sanity checks.

When a socket is created, the Netlabel hooks lead to the creation of the appropriate `netlabel_lsm_secattr` entries in the kernel. Further, the socket's `secid` and the peer's `secid` are also created. The socket's security structure contains these entries. Further there is a conversion from the host's SELinux MLS level and category data for the particular process, and they are filled up in the `netlabel_lsm_secattr`.

When the socket is in use (packets are sent out through the socket), the mappings are checked due to the hooks that invoke netlabel services in the Security Server, and the packet is labeled using the functions mentioned in table 2.3.3.

Reception of packets into a socket leads to the same process in reverse: The socket already has a `netlabel_lsm_secattr` attribute, and therefore, the CIPSO options that have been parsed by the networking subsystem are compared to the security attributes, and the appropriate inbound access checks are executed.

2.4 Xen

Xen is a Virtual Machine Monitor that runs multiple Virtual Machines (known as domains / guests) and provides a virtualized environment to them. Running Virtual Machines isolates them from each other, which has performance and security benefits. Further, Xen was, until recently, a *paravirtualized* system. In a paravirtualized system, the guest is *aware* of the fact that it is in a virtualized environment; it has modifications that enable it to run in such an environment.

Normally, an un-virtualized kernel executes in the highest privilege ring (Ring-0) and applications execute in outer rings (Ring-3). In a virtualized environment, the Ring-0 is occupied by the Hypervisor and the kernel executes in a lower privilege ring. Consequently, without hardware support, it is not possible to run the kernel in Ring-0 in such a scenario.

Paravirtualized guests replace Ring-0 operations with *Hypercalls*, which are requests to the hypervisor to perform these operations for them. The hypervisor executes these operations and schedules the domains to run, making *upcalls* to the domains to signal completion of the operations. The paravirtualized guests associate callbacks with these upcalls that take the actions required upon completion of the Ring-0 operations.

The operation of Xen is described in some detail in [3], only features of interest are discussed here.

2.4.1 Dom-0

The Dom-0 in Xen is a Linux Kernel. It provides the following services to other domains:

- Setting up the *VCPU*, the virtualized CPU for the domain, before startup.
- Allocating initial memory for the domain.
- Populating the domain's page tables with *pseudo-physical machine addresses*.

- Setting up the initial *event-channels* and *grant-tables* for the domain.
- Setting up the *start-info* and *shared-info* pages, that provide the domain with a usable environment in the beginning. (e.g., the start-info page provides such things as timing information, pending upcalls to be handled, page frame numbers for use – which are actually pseudo-physical machine addresses, etc.)
- Access to the real device drivers through the *split-driver* mechanism.

Chapter 3

Related Work

3.1 Security Administration Manager (SAM)

SAM, described in [2] is an implementation of Role-Based Access Control (RBAC). SAM is an administrative tool whose primary objective is to make *policy management* easier in a distributed system; it permits a (possibly distributed) organization to *articulate* an RBAC policy and then *translates it to its equivalent in the Access Control Model used by the underlying system*. Note that enforcement of this "translated" policy is up to the underlying system. SAM does not implement the RBAC model as defined by NIST completely. It is observed that certain RBAC features, e.g., Cardinality, are rarely used by organizations, and many objects, e.g., IPC objects, need not be access-controlled using RBAC. Therefore, they are not implemented by SAM. SAM takes the view that RBAC is a more intuitive way of modeling security policies, and performs the function of translating RBAC specifications to those supported by the underlying access control model (e.g., UNIX access control).

SAM divides enforcement responsibility into several smaller units. SAM defines a generic "master model" that dictates the authorizations associated with the objects, and the roles that can be assumed

by users. Information about the users is obtained directly from the Human Resources database of the employees, thus providing a centralized point of control. The database and the master model that contains the associations of users with roles change in sync. Further, the master model provides the template from which the models of other branches / organizational units / combinations of the two are populated. SAM ensures that changes to the master model propagate down to the models derived from it, that are stored in sub-units. Further, SAM ensures ease of administration by allowing non-technical security administrators to change the assignment of users to roles, while at the same time translating the RBAC model into the underlying Access Control Model of the system via automated scripts in the background. It has to be borne in mind that security administrators need to change only the assignment of users to groups. Groups have predefined roles associated with them, which in turn specify the resources that can be accessed in a particular role and the authorizations that will be given to access the resources. Roles have to be activated, upon which the authorizations become available. Roles are empty, groups are added to roles upon activation.

SAM permits initial definition of roles using exemplars of permission sets given to individual users with certain job functions, trimming down excess rights and the objects that absolutely need to be protected. This is called the “bottom-up” approach. The other, “top-down” approach involves consulting experienced employees to create roles with the exact number of authorizations and adding them to the master model, and then connecting individual user ids to the roles.

3.2 SESAME

Secure European System for Applications in a Multivendor Environment [11] (SESAME) is a security architecture for distributed systems. The main goals of SESAME are openness, flexibility and ease of management. SESAME provides authentication and authorization services for applications distributed across a large-scale system that comprises individual machines connected by an insecure network in-

frastructure. In particular, SESAME tackles the following issues: authentication of subjects, storage of security attributes, portable representation of security attributes, transfer of security attributes between applications, delegation of authorizations and key management in large scale distributed systems. SESAME accommodates heterogeneous applications by using standardized authorization information ("Privilege Attributes") encapsulated in Privilege Attribute Certificates (PACs). Privilege Attributes are translated into local representations of the Operating Systems at the endpoints for enforcement. Portability of authorization information is assured via the PAC concept. The TCB consists of SESAME components (that reside on the endpoints) that ensure that subjects are authenticated prior to access control enforcement; SESAME components are responsible for obtaining *only the required privilege attributes*, which implements the principle of least privilege. Further, the SESAME components also ensure that appropriate cryptographic protection is provided to protect PACs that are sent via networks. SESAME takes the view that privilege attributes of the objects are stored with them, whereas those of subjects are stored at central points per security domain, known as PA-Servers. This makes revocation of access rights easier, and eases policy management by providing a single point of security administration for a domain. This asymmetry of enforcement also means that minimal trust is reposed in the client application – the onus for enforcement is on the target system that contains the server application. To help scalability, the PA-Servers, Key Distribution Services and A-Servers (that perform authentication) might be replicated within domains. For large-scale systems, SESAME encourages the use of a PKI to verify PACs. Flexibility is encouraged via support for distinct identities for audit and naming, groups, roles that specify static sets of authorizations and clearances for secrecy-oriented policies.

Figure 3.1 shows the main components of the *initiator system* and the *target system*, which house the client and the server applications, respectively.

- **Subject Sponsor:** The Subject Sponsor represents the machine initiating access, and it might be authenticatable. It is responsible for providing a trusted path to applications (servers) , establishing

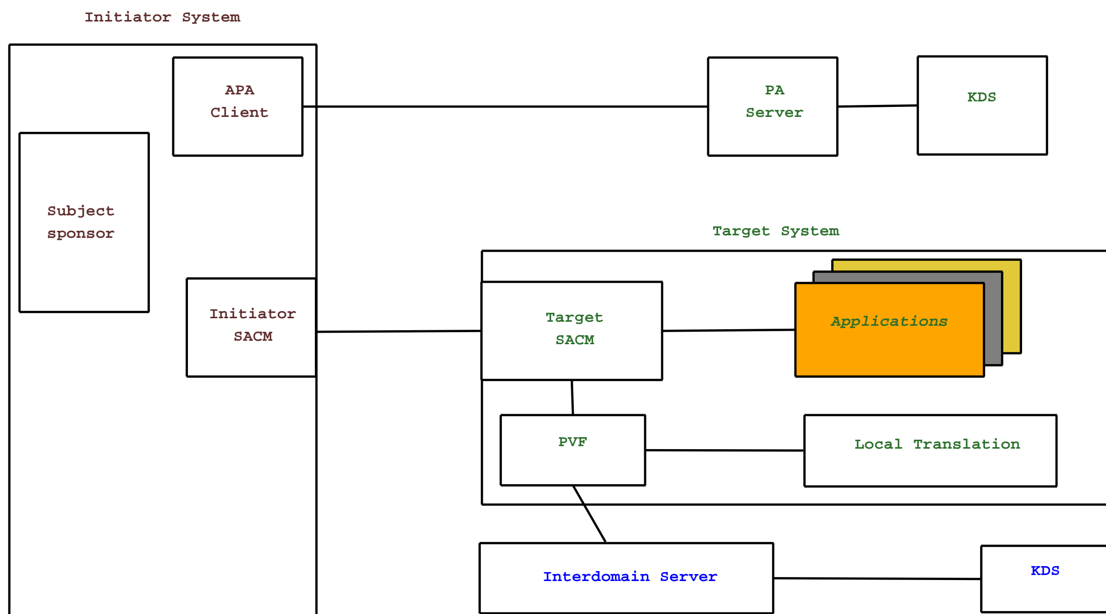

Figure 3.1. SESAME Architecture

keying material, if necessary and authenticating the secondary principal[1] (the user) by contacting the A-Server (Authentication Server, not shown in this figure).

- **Initiator SACM:** The initiator Security Association Context Manager (SACM) is one half of the context management infrastructure that resides on the initiator machine. An SAC is a secure association between the keying material used to protect conversations between the client and the server, together with the Privilege Attribute Certificate (PAC) for the session. To establish the SAC, the SACM has to retrieve the PAC and the keying material for each client-server session; this might involve communicating with the PA-Service.

- **APA-Client:** The Authentication and Privilege Attribute Client (APA-Client) is responsible for supplying the PACs to the initiator SACM mentioned above. Further, it also authenticates the user to the system by obtaining an Authentication Certificate from the A-Server. The APA-Client carries out all these activities in a manner transparent to the application/user.

- **Target SACM:** The target SACM is responsible for extracting the relevant privilege attributes of

[1]the primary principal is the machine, which itself has some privilege attributes associated with it

the client and passing it on to the application (server). This accommodates security-aware applications that might further enforce access control at the language level using the security attributes of subjects and objects.

- **PVF:** The PAC Validation Facility (PVF) is used by the SACM. Each application has a PVF associated with it; before passing on the privilege attributes, the SACM queries the validity of the PAC received using the PVF. The PVF also performs the function of obtaining the keying material needed to encrypt the conversation with the client, if necessary. It might be the case that the PAC to be validated might come from a PA-Server not directly trusted by the PVF. In this case, the Inter-domain server is queried to validate the PAC, and also to translate the Privilege Attributes to a local form (local translation - e.g., generate the equivalent group id in a UNIX system).

- **PA-Server:** The Privilege Attribute Server (PA-Server) serves as the repository of *Privilege Attributes* for all the subjects within a Security Domain. Privilege Attributes are a composite of several pieces of authorization information. These attributes have a standard meaning throughout the distributed system, and are independent of the representations on individual systems (hence the local translation). Privilege Attributes can include identity, group membership information, role information and clearance. A Privilege Attribute Certificate is similar to a Certificate issued by a CA — it contains Privilege Attributes, validity period, certificate id, Application Trust Groups and cryptographic algorithms that have to be used to protect the PAC in transit. The PAC is signed by the issuer, i.e., the PA-Server, using its private key. Thus, a PAC is portable, and can be verified in a manner similar to a certificate issued by a CA.

- **KDS:** The KDS is a Key Distribution Server, which is responsible for generating and maintaining secret keys of principals in the system, in addition to generating session keys to protect conversations between the SACMs.

Another important feature of SESAME is its support for *delegation of access rights*. The PAC is adapted

for this purpose to contain information about *Application Trust Groups* (ATGs). ATGs are groups of applications that are trusted not to mis-represent authorization information, and as such, access rights might be delegated freely between them. The ATGs are defined at the PA-Server, and encoded within PACs. Further, PACs can be constructed by *chaining*, wherein each application inserts its unique PAC id into the PAC that is passed on to the next application. At each stage of enforcement, the PVF defined for the application ensures that the (integrity-protected) PAC does indeed contain the correct chain of unique PAC ids that constitute a "legitimate" chain. The support for delegations goes much beyond the ability to specify chains; for more details please consult [11].

3.3 IBM DCE

Distributed Computing Environment (DCE) is a specification of a Distributed System built using existing operating systems (e.g., UNIX variants). Individual machines integrate DCE software in order to function seamlessly as a distributed platform, while supporting non-distributed applications. The main implementers of DCE are IBM, DEC and HP. DCE leverages TCP/IP, DNS, X.500 and POSIX to provide standard interfaces for communication and naming within and between systems, thus providing a high degree of flexibility. Applications use Remote Procedure Call (RPC) for communication using standard DCE interfaces; RPC is usually authenticated via the DCE security services. The smallest administrative unit that functions as a security domain is known as a *Cell*. Cells reflect the structure of the organization. We describe the Security Service, offered within each cell, that provides authentication and authorization services to applications. Client applications "bind" to servers that register themselves with a directory service (per cell). Intercell binding is achieved by consulting top-level DNS servers or Global Directory Services (based on X.500). The namespace within a cell is organized as a tree-like hierarchy; each name has attributes (specified by the DNS or X.500 standards) associated with it. These are the fundamental unit of access control in DCE. Individual resources within a CDS are grouped into *directories*. Authentication

(which will not be discussed here) is done via Kerberos. As with SESAME, a cell-level *Privilege Service* manages authorization data for subjects and issues PACs. In contrast to SESAME PACs, however, DCE PACs contain only the *name* of the subject and the group memberships of the subject. Similar to SESAME PACs, DCE PACs are also authenticated, but the key difference is that only secret-key cryptography is used[2], and the initial, authenticating, PAC is encrypted using the Authentication Server's secret key. To access a server, the client obtains *another* PAC that is encrypted with a key shared between the server and the Authentication Server. Obviously, the subject can obtain the PAC only after it has authenticated itself to the system; again, the DCE authentication differs from SESAME authentication in that only secret keys are used. The PAC is populated with authorization data that is obtained from the *registry*[3], which is a database defined at the cell level; the registry might be replicated for availability and fast access via a master-slave scheme. Again, similar to SESAME, the registry contains specifications of the keys to be used and possible user-defined authorization attributes (used by ACL managers). Another similarity with SESAME (and TAM, discussed in section 3.4)is that each application is responsible for enforcement; each application has its own *ACL Manager* that gives the security decision after consulting the ACL for the resource. The main information encoded within the ACL is its *type*, groups and names. (names might also be *foreign names* that contain remote subjects that are permitted to access the object). Note that the resources that are protected can be arbitrarily fine-grained, they might be directories, individual files, databases or database entries. A concrete example of an application would be the DFS (Distributed File System), that uses a "Token Manager" layer to obtain and compare PACs in order to authorize access, and converts PAC entries to UNIX ACLs. These UNIX ACLs are then used by the DAC system to further authorize file access. Thus, the ACL manager in this case is composed of both the Token Manager and the underlying DAC system in the UNIX kernel. Extensions to the system call API are necessary in the kernels running on the client and server machines; specifically, there are additional DFS system calls that are layered on top of DCE RPC (also part of the kernels).

[2] version 3.2 of IBM DCE uses public-key cryptography and a PKI infrastructure, falling back to secret-key cryptography if public-key cryptography is not available

[3] there is a cell-level registry service that extracts information from the registry upon request

Although DCE permits distributed access controls with existing operating systems, it suffers from the same trust problems as Kerberos, when it comes to inter-cell activity [30]. For instance, if a client wants to obtain a PAC for a server located in a different cell, then the Authentication Server for that cell has to be one that is trusted by the Authentication Server where the client is located. This means that the two Authentication Servers need to share a secret key, and the Authentication Server in the client's cell must pass on the PAC request to the Authentication Server in the server's cell. Compromise of an Authentication Server in one cell will affect trust relationships will *all* the other cells in the system. Further, Kerberos itself relies excessively on clocks to maintain tickets and ticket-granting-tickets. Although DCE provides robust Distributed Time Services, the compromise of the DTS in one cell has the potential to disrupt authentication in the *entire* system. Initial PACs *have* to be encrypted using secret keys established using Kerberos. By being tied exclusively to Kerberos for authentication and protection of authorization data (PACs), DCE inherits the same problems as those faced by Kerberos: ambiguous inter-domain policies, excessive use of secret-key cryptography, difficulty leading to compromise of long-term keys and excessive reliance on accurate notions of time that are hard to maintain as a Distributed System grows in size. Additionally, it has been pointed out in [30] that DCE does not support audit trails and delegation of authority. Further, as large cells that host all the servers can alleviate the trust problems we have mentioned earlier, an attempt was made to add some 52,000 users to one cell registry, but the system began to fail at 47,000 users, indicating scalability problems. Some of the claims in [30] have been refuted: the DCE specification underwent massive changes between the version reviewed and the version that was current at the time, and the current version of DCE supports audit trails, delegation, but it is conceded that there are limits to the size of cells, mainly due to implementation restrictions on the registry (a database). For a more detailed discussion of the refutations presented, refer to [29].

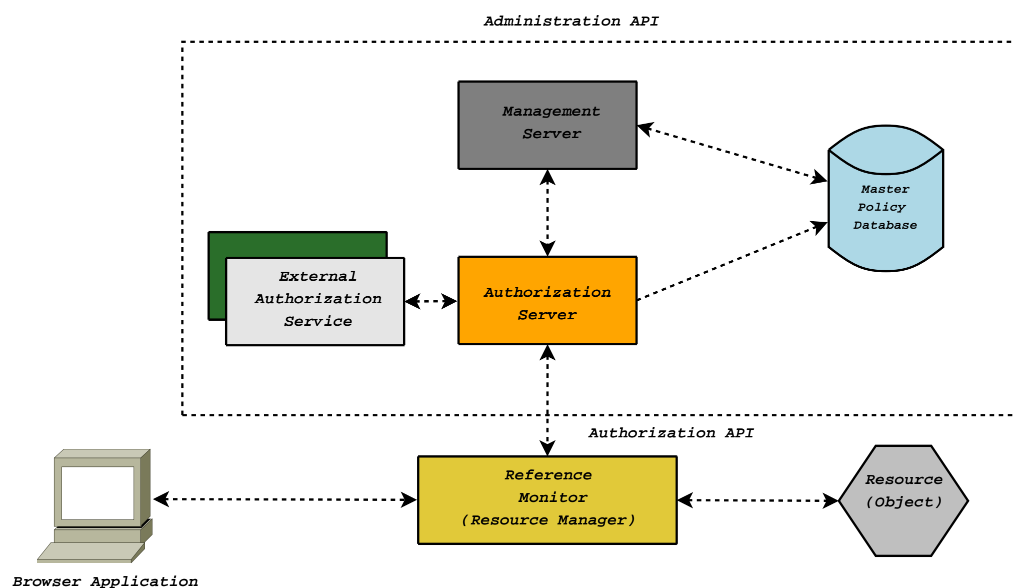

Figure 3.2. Tivoli Access Manager Architecture

3.4 Tivoli Access Manager

Tivoli Access Manager (AM) is a service built by IBM to provide authentication and access management to members of the AM family, which are applications that each have their own access control requirements. AM provides a centralized point of access control for distributed applications to physically dispersed resources. AM consists of an Authorization Server, that hands out access decisions to Reference Monitor embedded in the application (Application Server). The authorization server is accessed via an Authorization API, while it is administered via a command line utility `pdadmin`. As shown in figure 3.2, the Authorization Service consists of the Authorization Policy database, the management server (`pdmgrd` , used to control the policy state) and possible External Authorization Services. For high availability, the policy database can be replicated throughout the secure domain. Note that security attributes of principals and resources are maintained separately by external sources (e.g., an LDAP server) that do authentication.

Each resource has an associated Access Control List (ACL) and/or Protected Object Policy (POP) "templates". ACL and POP templates store authorization information, and their granularity varies. Individual users, groups, any authenticated principal, and unauthenticated principals can be specified in ACLs /

POPs. This has the advantage that authorizations of a large number of resources can be specified easily by a template at the right granularity, and any change in the template will affect all the objects specified by the template. ACLs and POPs are objects by themselves, and are further subject to access control; this reduces management complexity by allowing delegation of authority. AM employs a combination of Discretionary Access Control and Non-Discretionary Access Control — ACLs themselves are subject to DAC — but all other resources are subject to Mandatory Access Control. ACLs are used in yes/no access decisions, and authorization data can be specified using 18 standard permissions, in addition to 14 permissions that can be defined that are application and object-specific. Thus, ACLs in AM are quite expressive, like SELinux permissions, but they can be used purely in the access function internally, to return a yes/no decision. POPs are used to specify additional constraints on access decisions returned by the Authorization Server. These additional constraints are passed back as Attribute-Value pairs back to the Reference Monitor for further enforcement of the Access Decision. Some examples of POP attributes are audit levels, IP addresses of the user requesting access, time period during which access is granted, "conditional authorizations" (authorizations that will be granted provided some condition is enforced by the application) etc.

As AM is targeted at web applications, it models objects as members of a *hierarchy*, and they are accessed via a URI-like scheme. These Uris form the namespace for the secure domain. AM ACLs differ from traditional ACLs in that they are explicitly attached to each object; objects without explicit ACLs have implicit ACLs *derived via inheritance*, of the closest ancestor. Thus, the namespace for the secure domain is divided into *regions* rooted at a particular ACL template, that forms the basic unit of security administration. This basic unit can be arbitrarily fine-grained, and can also be dynamic, depending upon the URLs generated by web-applications. AM comes with certain standard namespaces are-defined, other namespaces can be added by applications (known as third-party namespaces), making Authorization Data extensible. The ACLs and POPs, which are objects themselves, are kept in a separate `Management` namespace that is subject to DAC.

Thus, AM addresses policy scalability by providing replicated Authorization Server components, and makes management of security policy easier by using ACL inheritance and DAC of ACL and POP templates. These features enable delegation of authority over arbitrarily fine-grained regions of the (possibly large) namespace(s). Further, AM provides a rich vocabulary of standard permissions that is further extensible by individual applications, thus encouraging heterogeneous applications to express different types of permissions interpreted by them. Finally, AM also supports Role-Based Access Control (RBAC), wherein roles are defined statically, but activated dynamically by populating initially empty groups that represent the users authorized for the role. Support for RBAC further simplifies policy management, as it reflects the authorizations associated with job functions in organizations — RBAC is believed to be a more intuitive security model than BLP/traditional Unix DAC.

3.5 Script Execution

Many popular web-based applications make increasing use of the browser capability to run scripts on the local machine. Typically, in DAC-based systems, these scripts (provided by the *writer*) are either restricted to *not* performing I/O or performing *any* I/O, i.e., either access *no files* belonging to the principal executing the script interpreter (called the *reader*) or *all files* belonging to the reader. While one extreme severely affects functionality, the other severely affects security – the script is able to access *any* of the reader's objects, in addition to many system files. An incremental solution, modifying the interpreter so that it has access to only a public directory, is cumbersome. An alternative access-control model [8] that permits the reader to control access completely, along with a prototype that implements this model is described in [9]. The intersection model permits access to only the files shared by both reader and writer. It also permits the reader to start processes with *different* access rights that correspond to the requirements of those different collaborations. Note that the identity of the writer must be known for this scheme to work. In [8], a comparison is made between three approaches: using the Andrew File System (AFS)

security model, using an interpreter that is modified to check authorization data before performing I/O *and* invoking further applications in the course of script execution, and the traditional UNIX DAC.

AFS obtains "tokens" from a Kerberos server that specify the group membership of the writer and compares it to that of the ACL for the authorization decision. The problem with this approach is that tokens are user-based, which means that all applications invoked on the reader-side have only one token. Adding new user-ids is cumbersome, as it requires administrator intervention. If the interpreter is modified, then it is possible to determine file access, but invoking applications and ensuring that they will, in turn, *not* access private files becomes difficult. One possibility is that all the possible "safe" application calls and modes of calling them be cataloged in advance so that only the "safe" application invocations are permitted. But, this approach is too cumbersome, to say the least. The only redeeming feature of this solution is that it accommodates heterogeneous systems – the operating system features are not relied upon for authorization decisions. Finally, the UNIX DAC-based solution requires that the reader call a `setuid` function that determines the *intersection*, i.e., group memberships of both the reader and the writer. Next, the *group ids* of the reader process are reset to those determined by the intersection and the *effective user id* of the process is set to `nobody`. This ensures that only the setuid application be present on all the (UNIX) machines that wish to run scripts, and also that the scripts run with appropriate access rights. Note that *all* the applications that will be invoked during script execution will run with these rights, so the problem of determining *which* applications to permit does not arise.

In [9], a prototype that implements a "kerberized" version of the interpreter is discussed. The objective is to ensure that other kerberized applications can be invoked from the interpreter, and they, in turn, can enforce the desired access rights. Thus, the main problem that this work addresses is delegation of access rights. The interpreter must have a way of representing access rights and also a way to tell other applications these access rights. A type of Kerberos ticket known as a "proxy ticket" is used for this purpose. This ticket has an *authorization data* field that is user-defined, and this field is used to convey the rights to be delegated to the application. Therefore, before transferring control, the interpreter obtains

a proxiable ticket-granting-ticket that can be used to request further proxy tickets. In order for delegation to succeed, first of all, the application must be available to the reader (the catalog of safe calls is used for this purpose), second, the application that is called must be kerberized (the catalog is augmented with lists of kerberized applications and safe parameters). Finally, if the previous two conditions hold, the kerberized application is invoked with the proxiable ticket-granting-ticket. The application passes this proxiable ticket-granting-ticket to obtain the proxy ticket, and if the proxy ticket can be decrypted successfully, then the access rights are extracted to enforce access control. The paper also discusses how *any, un-kerberized* application could enforce delegated rights with the help of the operating system. For a full discussion of solutions, please refer to [9].

3.6 VM Monitoring: XenAccess

Virtual Machine introspection is the ability to access any data within the VM from an external entity, in a manner transparent to the VM. VM Monitoring is a useful security tool, as it permits a monitoring infrastructure that can watch the VM for security policy violations. We have already discussed the need for monitoring the packet-labeling state of a VM in our system. Therefore, VM monitoring solutions are of interest to us.

In [24], the architecture and implementation of one such monitoring system, called *XenAccess* has been described. XenAccess aims to inspect memory and disk activity of an unprivileged guest from the Dom-0 running on Xen. It makes extensive use of the Xen control library `xenctrl` and the disk control library `blktapctrl` to monitor the memory and disk state, respectively, of a VM. XenAccess does not require modification of the guest, neither to the hypervisor. Further, the XenAccess library itself resides in Dom-0, thus providing isolating a monitor built using XenAccess from the VMs being monitored. These features make XenAccess an attractive solution for VM monitoring in Xen.

As already mentioned, Xen isolates guests from one another by providing them with separate page tables

that are subject to hardware protection, and also to protection from the hypervisor. Guests that wish to write to their page tables must make explicit hypercalls. Xen ensures that page table writes are valid and does not permit one guest to modify another guest's page tables. In the case of Dom-0, unlimited, complete access to all the guests' memory is possible, because the `libxenctrl` library is available to applications running inside it. It is possible for Dom-0 to *map* another guest's pages into its local address space using the functionality provided by `libxenctrl`.

Xen provides guests with the illusion of a contiguous memory space by providing them with *Pseudo-physical* Page Frame Numbers, that map to discontiguous *Machine Frame Numbers* (described in more detail in chapter 4). While Xen deals in terms of Machine Frame Numbers, guests are provided with non-writable page tables that carry mappings of the PFNs to MFNs and vice versa. XenAccess makes these tables available to applications in Dom-0 using `libxenctrl`. Thus, at any point of time, any memory area within the guest can be mapped into the Dom-0 and observed by monitoring applications. To provide a higher level of abstraction, XenAccess permits monitoring applications to specify kernel symbols by name, looking them up in the `system.map` file for the particular guest. The `system.map` file is a link map that shows the virtual addresses of symbols within the guest kernel. To improve performance, MFNs corresponding to recently-accessed symbols are cached by XenAccess. Further, XenAccess also provides access to memory regions of specific processes *within the guest*; only the process id is required. Memory regions of processes are accessed by walking the page table of the guest starting from the `pgd` - page global directory of the current process in the guest kernel. In addition to memory introspection capability, disk accesses by a guest can also be monitored by an application using XenAccess.

XenAccess provides a generic, efficient solution to the problem of VM monitoring, enabling a wide variety of VM monitors to be built. As we will describe our implementation in Chapter 4, similarities to XenAccess will become clear, as we also use the `libxenctrl` library to read and write to the guest's memory. Our solution takes advantage of the relaxed trust model of a paravirtualized environment to go a step further and use efficient kernel features *in the guest* to dynamically monitor and *change* the guest's

packet labeling state. This has the advantage that we prevent another layer of indirection — by directly using `libxenctrl` and we leverage the guest kernel interrupt handling mechanisms to efficiently process asynchronous changes in guest packet labeling policy.

3.7 sHype

As we have already discussed, Virtualization is growing in popularity, particularly in data centers where shared resources such as CPU, memory and network bandwidth can be rented out to host applications. As VMs enable common physical infrastructure to be shared efficiently, disparate applications that might otherwise not be run on the same machines are often run on VMs hosted on the same VMM. Although VMM-based systems provide strong isolation between applications, collocating applications that communicate frequently has performance benefits. As a result, information flows cannot be prohibited completely between VMs. [26] recognizes the need for mechanisms that regulate information flows between VMs, allowing VMs to explicitly share resources while preventing undesirable information flows between them. The sHype system, a Mandatory Access Control system built into the Xen hypervisor, offers controlled information sharing at the granularity of VMs, while remaining transparent to them.

Xen offers isolated partitions of real resources such as memory and CPU by presenting them as *virtualized, non-sharable* resources to VMs. Inter-VM communication, known as *inter-domain* communication can occur via memory, while notifications are delivered via *event-channels*. VMs explicitly grant access to memory pages by making entries in their *grant tables*, and publishing this information either in the *Xenstore*, a configuration database or by notifications via event channels. Event channels are two-way communication channels that have to be bound by *each end* individually, by obtaining access to *event channel ports*. Event channel ports are monotonically increasing integer values that can be published in the Xenstore. Dom-0 arbitrates access to device drivers by multiplexing them among VMs and limiting access to driver *backends* that carry out the actual operations on devices. The backends are available via

strictly defined interfaces, such as *device channels* — combinations of shared pages, grant table entries and event channels to simple *front end drivers* that are implemented by unprivileged VMs.

sHype enforces MAC through hooks that interpose on security-critical operations (hypercalls) in the hypervisor. These operations include: allocation of unbound event channels, binding of event channels to ports, allocation of grant tables, modification of grant table entries and creation, migration, startup and saving VMs. By completely mediating these security-critical operations to consult policy in the hypervisor, sHype provides comprehensive reference monitoring that is protected from the VMs using Xen protection mechanisms. sHype supports two types of policies: *Simple Type Enforcement* and *Chinese Wall Policy*. Simple Type Enforcement uses the TE-types of the VMs involved to identify coalition membership, while Chinese Wall Policy uses Chinese-wall types that indicate conflict sets that prohibit VMs from running at the same time. Simple Type Enforcement also serves to label resources such as virtual disks and recently, labeling has been extended to the virtual network interfaces available within VMs.

On the same lines as LSMs, the policy management and access decisions are encapsulated in an *Access Control Module* (ACM) within the hypervisor. The policy is stored in XML format on the Dom-0 filesystem and converted into binary form using `xm` subcommands and fed to the ACM. The policy is simple: it specifies the Simple Types and Chinese-wall types of each VM, in addition to labels for resources like disk and network interfaces. Each policy change leads to a reevaluation of conflict sets and coalition membership for all VMs; when they next use their resources, if their rights have been revoked, the VMs will not be granted access to them. To improve efficiency, decisions are cached by enforcement components within the hypervisor. Further, these caches are populated at the time resources are created (e.g., when event channels are created and bound), to alleviate the access latency associated with repeatedly querying the binary policy.

3.8 Labeled IPSec

IPSec is a suite of protocol standards designed to protect IP packets. IPSec is defined in many RFCs, the most important ones are RFCs 2401 [15](overview), 2409 and 4306 [5, 14](IKE - key exchange) and 2367 [20] (PF_KEY interface). IPSec provides cryptographic protection of packets, key establishment for cryptography and authentication of the endpoints (*peers*), in addition to NAT traversal and other minor features. Most major operating systems, such as Linux, Windows, BSD variants implement IPSec.

The Linux implementation of IPSec is called `XFRM` ("transform"), described in more detail in [12]. IPSec provides security at the IP layer, securing many protocols that run *above* it, e.g., TCP, UDP. Therefore, the IPSec subsystem of the kernel has to be made aware of the following to protect traffic:

- What traffic to protect? The kernel maintains a database, known as the *Security Policy Database* (SPD) to identify traffic to be protected. For each IP packet, one of three decisions might be taken: *discard* it without any further protocol processing, *bypass* the IPSec subsystem and route the packet, or *apply* IPSec processing to the packet. Further, the packets might be specified at any level of granularity — packets that originate from a particular host (end-to-end) or packets that originate from a network (VPN). The SPD is indexed using a *selector* that helps guide the processing decision. The selector might consist of source and destination IP addresses (at the host or network granularity), upper-layer protocol that generated the packet and port numbers. Once an *apply* decision is taken, the next step for the kernel is to find out *how* IPSec processing is to be applied to the packet.

- How to protect traffic? The types of IP packets that have to be protected are protected by applying various security "services" (e.g., Only integrity protection, additional encryption of packet). The specification of services that have to be provided by IPSec to the packets specified by the SPD is encapsulated in a *Security Association* (SA). An SA is *unidirectional*; to process both incoming and outgoing packets that match an SPD entry, *two* SAs have to be present. The kernel keeps track of

all SAs in the *Security Association Database* (SADB).

Even before IPSec processing is applied to a packet, the peer has to be authenticated. Once authentication has occurred, both ends of the connection have to negotiate parameters and pick an agreed upon set of security services they will apply to packets. Next, both ends have to exchange the requisite keying material for encryption and to generate cryptographic hashes that will ensure packet confidentiality and integrity. This negotiation is a complex process and is handled by a userspace daemon, and must occur in accordance with a standardized protocol known as IKE (Internet Key Exchange), described in RFC 4306. IKE runs on UDP port 500 and might also use port 4500. The Linux implementation of this daemon is known as *racoon*. Ultimately, racoon negotiations lead to the creation of SAs that will be fed to the kernel to *apply* IPSec processing.

Labeled IPSec [10] adds security contexts to the XFRM policy data structures in the Linux Kernel. In order to allocate and deallocate these data structures, it adds four hooks, `xfrm_policy_alloc`, `xfrm_policy_free`, `xfrm_state_alloc` and `xfrm_state_free`. The data structure `xfrm_policy` corresponds to the SPD entry, whereas the data structure `xfrm_state` corresponds to an SA. Protocol services above IP are accessed via the socket interface in Linux. Sockets have security contexts that are managed by SELinux, which are in turn derived from that of the process that creates the sockets. Therefore, when a policy has to be retrieved to apply IPSec services to a packet, the policy label (allocated via the `xfrm_policy_alloc` hook) has to match that of the socket, according to a socket-specific policy or a system wide policy. Similarly, once a policy match occurs, the next step is the insertion of the SA into the SADB. Once again, the context of the SA(s) obtained from userspace, that have been labeled (via the `xfrm_state_alloc` hook) has to match that of the socket that originates the packets. Thus, network communication is subjected to MAC in accordance with system wide policy by IPSec. It has to be noted that the labels are not actually transmitted by Labeled IPSec; the mere fact that policy and SA matches have occurred successfully on both the sides of an IPSec connection allow enforcement of Network Mandatory Access Control. The IPSec SA selection itself still conforms to IETF standards, as

only an extra context field is added to the selector.

The Linux Kernel has to be compiled with the Labeled IPSec feature enabled to take advantage of this form of Network Mandatory Access Control. Further, modified userspace tools (collectively known as ipsec-tools [13]) that permit insertion of security contexts into the SPD via `setkey` and a modified `racoon` daemon that uses these contexts for negotiation of SAs has to be used for Labeled IPSec.

3.9 Selopt

Labeled IPSec allows the strongly authenticated endpoints to negotiate an SA, that is guaranteed to be accessible only to sockets with appropriate authorization; the policy creation is also subject to Mandatory Access Control. In a scenario where all the hosts that connect to the network are assumed to be trusted, cryptographic protection of packets is not needed. But, authorization data needs to be conveyed to the endpoints so that Mandatory Access Control can be applied to packets. There are several standards (e.g., CIPSO, RIPSO, FIPS 188) that define how this information is to be transmitted, how it is to be handled by endpoints, and how authorization data expressed in one policy context has to be converted to its equivalent in another policy context. These standards have not made it to the status of IETF standards, but they are used widely by the defense community and are supported in trusted operating systems, e.g., Solaris. All these standards encode security data in the Options field of IP packets.

Selopt [23] was a precursor to the Netlabel implementation in the Linux Kernel. Selopt is based on FIPS 188, and it is intended to convey authorization data. If confidentiality of data in transit is required, some other mechanism, such as IPSec has to be applied. Selopt was distributed as a patch to the Linux kernel, along with userspace tools that communicated with the Kernel via the `Netlink` interface.

The tag formats permitted by FIPS are similar to those for CIPSO, which we have already discussed. The tag format used by Selopt is the “Free Form” security tag. As the name suggests, free form tags are variable-length and may be interpreted by the protocol stack differently in each administrative domain.

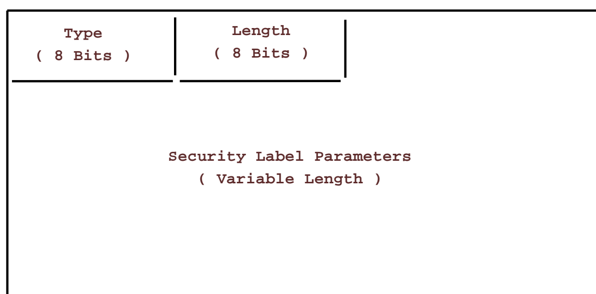

Figure 3.3. Free Form Tag

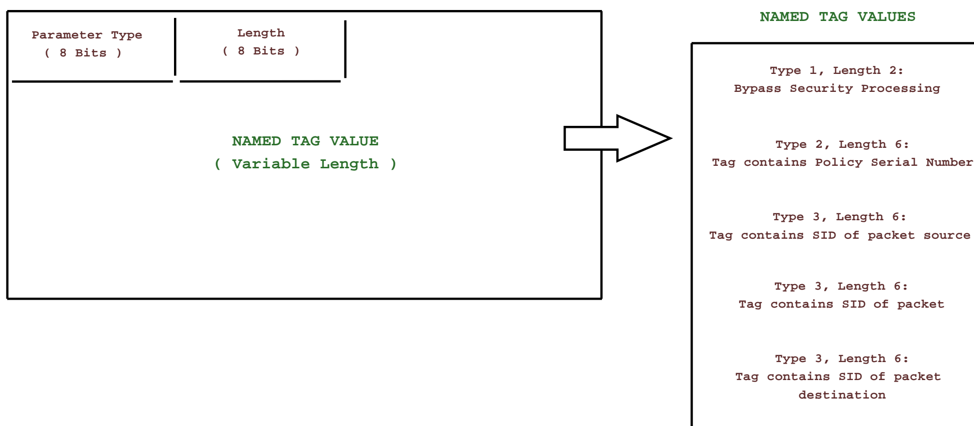

Figure 3.4. Named Tags used by Selopt

Within a domain, though, free form tags have to be processed uniformly. The free form tag is shown in figure 3.3.

The free form tag can further contain security information about the objects carried by the packet, known as “named tags”. Every IP packet *must* contain one type of tag at least, but it might carry more than one tag at the same time. The length of the fields is variable, and is defined by the type of named tag. The types of named tags possible are shown in figure 3.4

The SID field of the named tag is an integer identifier that is used to retrieve authorizations within SELinux. As such, an SID is meaningful only locally. Therefore, in order to retrieve contexts that correspond to network objects(other end), Selopt includes a userspace daemon `scmpd` (Security Context Mapping Daemon). `scmpd` runs on UDP and it has to be protected using IPSec if the network is not believed to be secure. In order for Selopt to work, policies must be equivalent, and user, role and MLS

levels should have the same meaning throughout the administrative domain. `scmpd` communicates with the kernel using netlink. In addition to caching the SIDs of remote objects (called NSIDs) for efficiency, NSIDs that do not have local equivalents yet are queued for processing by Selopt. The state of queues can be inspected and modified using the /proc interface provided by Selopt. Selopt relies upon hooks in the TCP/IP stack to process packets and insert options in them, and also provides an internal Kernel API. But one of the weaknesses of Selopt is that it is not integrated with SELinux policy. This means that Selopt only decodes and encodes labels, but does not check labels for policy compliance. Another problem with Selopt is that it relies upon variable length IP Options (because of the Free Form Tags). IP Options compilation is an expensive process in the Linux Kernel, as it is in the critical path taken by *all* IP packets. Therefore, efficient IP Options processing is a prerequisite for any such labeling scheme to be accepted into the Kernel. Further, both SELinux and the networking code in the Linux Kernel have changed considerably since the introduction of the Selopt patch. Therefore, Selopt was not included in the mainline kernel — it was removed shortly after 2.4.18-lsm2 version of the Linux Kernel was released. In contrast to Selopt, Netlabel implements only the Restrictive Bitmap Type and limits the IP Options processing that occurs for packets — in fact, IP Options are processed only *once* for each packet. Further, Netlabel development has leveraged the new SELinux LSM model and is fully integrated with SELinux. Through SELinux hooks and services in the Security Server, Netlabel allows SELinux contexts to be associated with packets and does access checks based on socket context and packet contexts.

Chapter 4

Implementation

In this chapter, we describe the interaction of the netlabel subsystem in the Linux Kernel with userspace. Next, we describe the design of the `netlabeld` daemon that enables administration of packet labeling on unprivileged domains. We describe the implementation of `netlabeld` and the changes to the Xen control tools that must be made to enable an administrator on Dom-0 to control policy that governs packet labeling on unprivileged domains.

4.1 Netlabel userspace tools

The userspace tool used to interact with the netlabel subsystem is known as `netlabelctl` and is contained in the package `netlabel-tools` on our system. `netlabelctl` is written entirely in C; it is a command-line program that can perform the following operations:

- Create DOIs that specify the labels and categories that can be encoded in IPv4 packets, and send them to the kernel. These DOIs follow the CIPSO draft (partially).
- Delete DOIs from the kernel.

- Associate SELinux domains with DOIs, and send them to the kernel. Packets sent out by a process executing in the specified domain will then be labeled using the DOI specified.
- Delete SELinux domain - to - DOI associations from the kernel.
- Specify whether “unlabeled packets”, i.e., packets that do not contain CIPSO options are allowed to be processed by the TCP/IP stack in the kernel.
- Retrieve the capabilities of the kernel’s netlabel subsystem.
- Retrieve the DOIs in the kernel, their mappings to SELinux domains and the status of unlabeled packet processing in the kernel.

`netlabelctl` interacts with the kernel via the `NETLINK` interface. The Netlink socket mechanism was introduced as a connectionless, socket-based interface between userspace processes and the kernel, with a view to reducing code complexity in the kernel. Netlink sockets support the BSD socket API substantially, and belong to the `AF_NETLINK` domain. Netlink sockets are bidirectional and might belong to protocol families `NETLINK_ROUTE` , `NETLINK_FIREWALL` , `NETLINK_XFRM` etc. Netlink socket functionality has to be explicitly enabled in the kernel configuration; if enabled, the Netlink subsystem registers itself in the protocol table and address family table during system boot. Netlink functionality can be used via the `libnl` library. As with any standardized protocol, netlink messages must conform to a specific format. The most important part of the message is the header, `nlmsghdr` , that contains, among other details, the pid of the sender , certain flags, and the message type. The header can be followed by padding and data. Each protocol family further specifies legal values for the message types. Netlabel uses the generic netlink protocol family and defines formats for messages that it uses to communicate with `netlabelctl`. Thus, `netlabelctl` creates messages with data encoded as netlink attributes, headers specifying the type of the messages and sends them on a netlink socket to the netlabel subsystem in the kernel. Netlabel associates “handlers” with each type of message. Handlers are callback functions that are executed upon receipt of the netlink message from userspace.

4.2 Netlabel Subsystem in Kernel

At present, the netlabel subsystem in the Linux kernel is very simple. During system initialization

- Kernel data structures that represent CIPSO DOIs are initialized.
- In-kernel cache structures for DOIs and LSM mappings are initialized via the slab allocator.
- The netlink message handlers and protocols are initialized.
- By default, unlabeled traffic is permitted.

Further, no DOIs are established at boot time, neither are SELinux domain to DOI mappings.

4.3 Xend and xm Implementation

Xen achieves superior performance by carrying out *data path* operations exclusively, i.e., the hypervisor is responsible only for activities such as network data transfer, updates to page tables etc. *Control plane* activities are performed by user-level processes in Dom-0, via special *control interfaces* to the hypervisor available only to Dom-0. The userspace daemon `Xend` is responsible for control-plane activities such as starting domains, destroying them, controlling memory allocation to domains and setting up virtual devices needed by them. Xend can export services in a number of ways, e.g., XML-RPC, HTTP. In our setup, Xend runs an XML-RPC server and listens to requests on a Unix domain socket. Administrators interact with Xend via a client known as `xm`. xm is written entirely in Python. High-level functions in Xend are written in Python; certain low-level activities e.g., interaction with the Xen Control interfaces occur via calls to `libxenctrl` , `libxenstore` and `libxenguest` , that are shared libraries written in C. Calls to these shared libraries are made through the Python/C interface. VM state is represented in Python objects that are instantiated and destroyed by Xend when commands to create/ start and destroy VMs are issued via xm.

4.3.1 Xend actions for Domain Creation and Startup

Domains are created and started using the `xm create` and `xm start` commands. When these commands are issued, configuration variables are read in, either from a file or from command line arguments to `xm`, and requests, along with the configuration variables, are sent to `Xend`. Upon receipt of these commands, Xend creates a `XendDomainInfo` object that represents the domain, with attributes instantiated based on configuration variables. Next, Xend invokes the methods of the `XendDomainInfo` object. This process is described in brief below. We will look at this process in greater detail when we describe our modifications to `Xend` and `xm`.

- **Domain Construction:** The balloon driver is invoked to obtain space for page tables for the domain. Data structures for the domain's event channels, grant tables, VCPUs and its shared information pages are allocated and initialized.

- **Domain Initialization:** Configures the domain's boot loader. Configuration values indicate if the `pygrub` boot loader is to be used (so that the kernel image is in the domain's filesystem) or if the kernel image is in the Dom-0 filesystem. Timekeeping structures are initialized. For a Linux guest, the `image` attribute of the `XendDomainInfo` object is set to a `LinuxImageHandler` object. Real memory is obtained from the hypervisor for the domain. The `_createChannels()` method is invoked to allocate the event channels required by the domain. If the domain requires devices, then device interfaces are created for the domain through the `XenDomainInfo` object's `_createDevices()` method and are configured (e.g., hotplugging is used in Dom-0). The `xs_introduce_domain()` call is issued via `libxenstore` to create appropriate `xenstore` entries. Finally, the `linux_build` call is issued via `libxenctrl` and the `XendDomainInfo` object that represents the domain is added to the list of domains managed by `Xend`.

4.4 Need for modified userspace tools

If `netlabelctl` is used to control packet labeling for the entire system, it has to be run on *each* guest by the administrator. As `netlabelctl` is a userspace tool, we cannot be sure about its integrity on a guest, unless all the software running on the guest attests to some standard of integrity. Even if such attestation is available, we have to consider the problem of *bootstrapping a guest with known DOI definitions*, because, by default, the Linux kernel is brought up with no DOIs defined. By bootstrapping a guest with proper DOI definitions, we ensure that

- The guest can communicate with others, and also with Dom-0. CIPSO specifications state that if a packet labeled with an unknown DOI is received, the sender must receive an ICMP type-12 (Parameter Problem) message in reply, and the packet must be discarded.
- If and when the guest associates a (proper) DOI definition with an SELinux domain, the process sends out packets with the proper DOIs.
- Confidentiality of messages is preserved. If all the guests are provided with DOIs that specify the "correct" secrecy levels, we can state that the simple-security property and the *-property are satisfied by our Netlabel policy.

`netlabelctl` makes the assumption of a standalone system - as such, it is geared towards controlling the netlabel subsystem on *one* host - the one on which it is running. In our scenario, we have to control the netlabel subsystems on *all the guest kernels*. Further, `netlabelctl` is not aware of the kernel life cycle as a VM. Therefore, `netlabelctl` has to be modified to be aware of the VM life cycle and control the guest kernels' packet labeling state. This suggests the need for a long-lived process on Dom-0 that can bootstrap guests with netlabel policies as they are created / started, dynamically change the guests' packet labeling state (in kernel) and keep the Dom-0's packet-labeling state in sync with that of all guests on the system. We call our daemon `netlabeld` , and it addresses these requirements by interacting with Xend and presents an interface via xm subcommands that can be used to control `netlabeld`.

`Xend` also needs to be modified to support xm subcommands specific to `netlabeld`. Specifically, the control-plane activities that need to occur when a domain has to be bootstrapped with DOI definitions have to be added to `Xend`. Some of these activities include: allocation of "special pages" to domains to hold DOI mappings and allocation of event channels for notification of netlabel subsystem changes in guest kernels.

4.5 Need for Modified Guest Kernel

The default behavior of the netlabel subsystem in the kernel has been described in section 4.2. This is suitable for a standalone system, where it is reasonable assumption that this process can be customized using init scripts that establish DOIs after the kernel has been brought up. But, the integrity of these scripts is questionable in our scenario; for instance, the filesystem on which these scripts reside might not be trusted. Even if the integrity of these scripts can be verified, there is no guarantee that the DOIs will be consistent with those of unprivileged guests and Dom-0 at the time the domain is started on a particular Xen VMM. This suggests the need for a modified guest kernel:

- The guest kernel must be able to obtain the "proper" DOIs it is expected to use from the Dom-0.
- The netlabel subsystem must initialize its packet labeling policy using these DOIs thus obtained.
- Dom-0 and the guest must be able to communicate changes in the packet labeling policy dynamically *as long as the guest is in 'running' state*.
- If and when SELinux contexts are associated with DOIs, these contexts must be made known to Dom-0, so that they can be checked if they are permissible mappings, by Dom-0.
- If DOIs are added to or deleted from the guest kernel, the guest kernel *must inform Dom-0* of these mappings, so that they can be checked if they are in accordance with policy.
- The guest kernel *must use* security contexts derived from netlabel mappings to verify peer labels.

This process ensures that when packets are sent out to or received from other guests on the *same VMM*, the lightweight tunnels are used instead of expensive IPSec tunnels.

4.6 Need for Modified Guest Kernel Interface

When a guest starts up on Xen, it has to be provided with a virtualized environment. Some examples are: configuration variables that specify devices available to the guest, timer values necessary for scheduling and device access, an initial console (so that boot messages from the kernel can be seen) etc. Some of this information is architecture-specific, e.g., initial page table frames, certain elements of the VCPU etc, while others are common to all types of guests, e.g., the event-channel ports for communication with Dom-0, event channel status array, etc. This information is provided to the guest in two areas: `start_info` and `shared_info`.

To ensure that the guest knows where to find DOI mappings and knows the event channel port of the Dom-0 (for notifications) when it starts up, they have to be conveyed to the guest kernel in an area shared between guest and Dom-0. Therefore, the interface offered to the guest kernel has to change. Further, the guest must be provided with accessible shared variables to communicate the status of the netlabel system to Dom-0, and vice versa. Specifically, the `shared_info` area that conveys this information to the guest has to be modified, and the guest kernel, that maps `shared_info` into its virtual memory, has to be aware of this modified interface.

4.7 Modifications to Guest Kernel Interface

The `shared_info` page serves as a common location for shared data structures used by Dom-0 and guests. Each guest that is started on Xen receives this page and has to map it into its virtual address space to function. We use the `shared_info` page 4.7 to share the following variables listed below:

Figure 4.1. `shared_info` structure (`include/xen/interface/xen.h`)

```
struct shared_info {
    struct vcpu_info vcpu_info[MAX_VIRT_CPUS];

    unsigned long evtchn_pending[sizeof(unsigned long) * 8];
    unsigned long evtchn_mask[sizeof(unsigned long) * 8];

    uint32_t wc_version;      /* Version counter: see vcpu_time_info_t. */
    uint32_t wc_sec;          /* Secs  00:00:00 UTC, Jan 1, 1970.  */
    uint32_t wc_nsec;         /* Nsecs 00:00:00 UTC, Jan 1, 1970.  */

    struct arch_shared_info arch;

    xen_pfn_t cipso_mfn;        /* used by dom0 */
    xen_pfn_t lsm_mfn;          /* used by dom0 */
    xen_pfn_t cipso_pfn;        /* used by guest */
    xen_pfn_t lsm_pfn;          /* used by guest */
    uint32_t num_cipso_dois;
    uint32_t cipso_dom0_evtchn; /* Dom0's end of the cipso event channel */
    uint32_t cipso_guest_evtchn; /* Guest's end of the cipso event channel */
    uint32_t lsm_dom0_evtchn;     /* Dom0's end of the security server event channel */
    uint32_t lsm_guest_evtchn;    /* Guest's end of the security server event channel */
    uint8_t cipso_status;         /* cipso status of guest or dom0 */
    uint8_t lsm_status;           /* lsm status of guest or dom0  */
};
#ifndef __XEN__
typedef struct shared_info shared_info_t;
#endif
```

The fields added to `shared_info` will be explained in the sections that follow.

4.8 Netlabeld Implementation

`Netlabeld` is a daemon that runs on Dom-0, responsible for managing packet labeling in unprivileged domains. Netlabeld listens on a Unix domain socket to requests. The daemon does multiplexed I/O over the Unix domain socket and event-channel file descriptors (fds) to serve requests. Netlabeld `selects()` fds available for reading and carries out actions based on the type of the fd: whether it corresponds to the Unix socket, or an open event channel driver interface. The main loop of netlabeld is illustrated in figure 4.2

Each request to the daemon on the Unix socket consists of a header and possible data following it. The

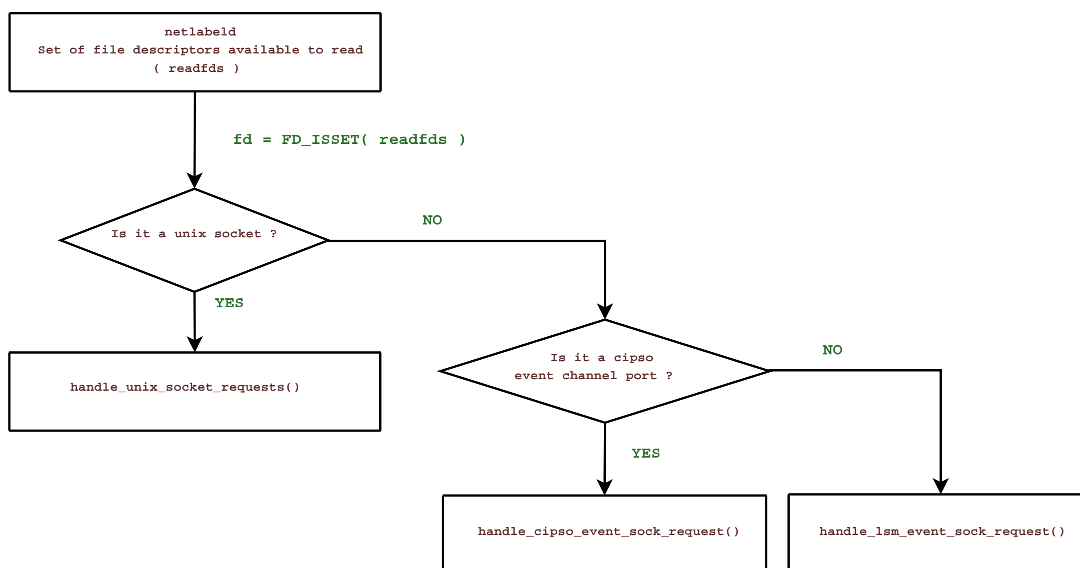

Figure 4.2. netlabeld main loop

header consists of the `type` of message, `domid` of the guest, `length` of the message, followed by data, if present. The message types and their meanings are listed in table 4.8.

Message Type	Meaning
ADD_INITIAL_DOIS	Request to add DOI definitions for a guest to netlabeld
ADD_INITIAL_DOIS_SUCCESS	Request to add DOI definitions succeeded
ADD_INITIAL_DOIS_FAILURE	Request to add DOI definitions failed
SEND_CIPSO_DOIS	Request to send DOI definitions for a guest in ‘packed’ format
SEND_CIPSO_DOIS_SUCCESS	Request to send DOI definitions succeeded
SEND_CIPSO_DOIS_FAILURE	Request to send DOI definitions failed
DEL_ALL_DOIS	Request to delete all DOI definitions for a domain
DEL_ALL_DOIS_SUCCESS	Request to delete all DOI definitions succeeded
DEL_ALL_DOIS_FAILURE	Request to delete all DOI definitions failed
ADD_DOI_LIBXC_DATA	Request to add guest details from `libxenctrl`
ADD_DOI_LIBXC_DATA_SUCCESS	Request to add guest details succeeded
ADD_DOI_LIBXC_DATA_FAILURE	Request to add guest details failed
LIST_ALL_DOIS	Request to display all DOIs for a guest
LIST_ALL_DOIS_SUCCES	Request to display all DOIs succeeded
LIST_ALL_DOIS_FAILURE	Request to display all DOIs failed

Table 4.1. Message types used by netlabeld

Figure 4.3. cipso_table structure

```
struct cipso_table {
        domid_t domid;
        unsigned int cipso_mfn;
        uint32_t num_cipso_dois;
        uint32_t cipso_dom0_evtchn;
        uint32_t cipso_guest_evtchn;
        int cipso_evtchn_fd;
        int xce_handle;
        struct cipso_list *list;
        struct doidef_list *doidef_listp;
        struct cipso_table *next;
};
```

Figure 4.4. cipso_list structure

```
struct cipso_list {
        nlbl_cv4_doi doi;
        nlbl_cv4_tag_a *tags;
        nlbl_cv4_lvl_a *lvls;
        nlbl_cv4_cat_a *cats;
        nlbl_cv4_mtype *mtype;
        struct cipso_list *next;
};
```

4.8.1 Data Structures used by Netlabeld

Netlabeld maintains linked lists of nodes that represent information about the DOIs associated with each domain. The important data structures are `cipso_table` , `cipso_list` and `doidef_list` .

The `cipso_table` structure (listing 4.3) contains the guest's domid, pointers to the list of DOIs associated with the guest, the list of "kernel" data structures for each DOI. Each guest that has to be bootstrapped with DOI entries has a `cipso_table` entry in Netlabeld.

The `cipso_list` entry (listing 4.4) contains a single DOI definition. The `doi` field contains the DOI number, followed by the tag type of the DOI (`tags`), the CIPSO and local levels of the DOI (`lvls`), the CIPSO and local categories of the DOI (`cats`), and the type of the DOI definition (`mtype`). [1]

[1] Currently only CIPSO_V4_STD , i.e., Standard DOIs are supported.

Figure 4.5. cipso_v4_doi structure

```
/* DOI definition struct */
#define CIPSO_V4_TAG_MAXCNT          5
struct cipso_v4_doi {
        uint32_t doi;
        uint32_t type;
        union {
                struct cipso_v4_std_map_tbl *std;
        } map;
        uint8_t tags[CIPSO_V4_TAG_MAXCNT];

        uint32_t valid;
};
```

Figure 4.6. doidef_list structure

```
struct doidef_list {
        struct doidef_list *next;
        struct cipso_v4_doi *doidef;
};
```

The DOI definitions that are added by netlabeld into a guest are read in the netlabel subsystem initialization code in the guest kernel when it boots up. Therefore, the DOI definitions that are added must correspond to the "kernel" data structures used by the guest kernel. To generate these DOIs , netlabeld maintains a "kernelized" data structure for each DOI associated with a guest, given by `cipso_v4_doi` (listing 4.5). A list of these "kernelized" data structures is maintained in `doidef_list` (listing 4.6).

4.8.2 Adding Initial DOIs

When DOIs have to be added to a guest when it is created / started, Netlabeld receives a request of type `ADD_INITIAL_DOIS` , with the domid of the guest. Upon receipt of this message:

- An initial `cipso_table` data structure is created.
- The message sent will contain the command in a string format, e.g.,

`netlabeld cipsov4 add std doi:2 tags:1 levels:12=13,14=15 categories:255=255`

This command will be converted into an argument list and sent to the `cipsov4` module of netlabeld that manages DOIs.

- For the DOI requested, a `cipso_v4_doi` entry is created and added to the `cipso_list` for the guest.
- The `cipso_list` entry is linked to the `cipso_list` pointer in the `cipso_table` structure for the domain.
- If the DOI is legitimate, then a `CIPSO_ADD_INITIAL_DOIS_SUCCESS` response is sent, otherwise a `CIPSO_ADD_INITIAL_DOIS_FAILURE` response is sent.

Note that each "add" request is sent as a separate request to Netlabeld; each request is processed in the above fashion.

Next, Netlabeld receives a request of type `SEND_CIPSO_DOIS` , along with the domid of the guest. This request means that the "kernelized" DOIs have to be provided. Upon receipt of this message:

- The `cipso_table` entry for the guest is obtained.
- From the `cipso_table`, the list of `cipso_list` entries is traversed to obtain each DOI for the guest.
- For each such `cipso_list` entry that is gotten, a `cipso_v4_doi` entry is created. A list of these "kernelized" DOIs is created for the guest and added to the `doidef_list` entries for the guest in `cipso_table`.
- A "packed" structure that contains the `doidef_list` entries is copied into the response message, and a response of type `CIPSO_SEND_DOIS_SUCCESS` is sent. The message contains the packed set of DOIs following the header, and the number of such entries. If the process fails for some reason , e.g., if no DOIs have been added by issuing a `ADD_INITIAL_DOIS` request beforehand, then a `CIPSO_SEND_DOIS_FAILURE` response is sent.

The packed set of DOIs is in a form that can be used by the (modified) guest kernel's netlabel subsystem initialization code.

Finally, Netlabeld receives a request of type `ADD_DOI_LIBXC_DATA`. This request indicates that the guest has been allocated and certain information about the guest is available. The request is followed by the following data:

- `cipso_mfn`: This is the `machine-frame number` of the page that will hold DOIs for the guest. It also implicitly indicates that the kernelized DOIs supplied in response to `CIPSO_SEND_DOIS` has been copied into this page.
- `num_cipso_dois`: This is the number of DOIs that have been copied into the page.
- `cipso_guest_evtchn`: This is the event-channel port that has been allocated *in the guest* and is available to be bound for *interdomain communication*.
- `lsm_mfn`: This is the `machine-frame number` of the page that will hold SELinux context - DOI mappings for the guest.
- `lsm_guest_evtch`: This is the event-channel port that has been allocated *in the guest* and is available to be bound for *interdomain communication*.

This request indicates that the guest has been started, and will go into "running" state. We describe this how this request is handled below, listings 4.7, 4.8,

4.9 show the functions in detail.

- Netlabeld makes the call `add_cipso_libxc_data()`.
- In `add_cipso_libxc_data`, the data items sent in the message are added to the `cipso_table` entry for the guest.
- `add_cipso_libxc_data` makes the call `setup_event_handling(domid, tab, cipso_evtchn_fd)`

where `tab` is the `cipso_table` entry for the guest.

- In `setup_event_handling`, privileged interfaces to the hypervisor and event channel drivers are opened via calls `xc_interface_open()` and `xc_evtchn_open()` provided by `libxenctrl`.
- Using the hypervisor interface, information about the guest is obtained via `xc_domain_getinfo()`. This call is made in a loop until the guest's state goes to "running".
- When the guest goes to "running" state, using the guest event channel information supplied, a new interdomain event channel endpoint bound to the guest eventchannel is returned. Note that netlabeld has to loop until the guest goes to "running" state before it can bind to `cipso_guest_evtchn` , otherwise the bind call will fail.
- Next, the guest has to be made aware of the event-channel endpoint *it must use for communication*. Therefore, the Dom-0's event-channel endpoint is written in the `shared_info` page accessible to both guest and Dom-0.
- To indicate that the Dom-0 is prepared to communicate with the netlabel subsystem of the guest, the `cipso_status` shared variable in the `shared_info` frame is updated to `DOM0_UP` state.
- A "test" event is sent to the guest via the event channel using the `xc_evtchn_notify()` call.
- Finally, a file descriptor corresponding to the open, privileged interface to the event channel driver is returned in `cipso_evtchn_fd` via the `xc_evtchn_fd()` call.
- When the call to `add_cipso_libxc_data()` succeeds, netlabeld adds the file descriptor described above to the set of fds that are `select()`ed by netlabeld, using the function `FD_SET( cipso_evtchn_fd, &readfds)`.
- If there is an error in any step during this process, the guest is destroyed using the `xc_domain_destroy()` function provided by `libxenctrl`, and response `ADD_DOI_LIBXC_DATA_FAILURE` message is sent. If there are no errors, then a `ADD_DOI_LIBXC_DATA_SUCCESS` response is sent.

Figure 4.7. add_cipso_libxc_data function

```c
/*
 * add_cipso_libxc_data:
 * libxc reports the cipso machine frame number,
 * number of DOIs given to domU initially
 * and the guest's cipso event channel port.
 * cipso_evtchn_fd is an output parameter used by main.
 * These items are added to cipso table for the domain.
 */

int add_cipso_libxc_data(domid_t domid, unsigned long cipso_mfn, uint32_t num_cipso_dois,
                         uint32_t cipso_guest_evtchn, int *cipso_evtchn_fd)
{
        int retval = -1;
        struct cipso_table *tab = get_cipso_table(domid);

        if( tab == NULL ){
                retval = -1;
                goto out;
        }
        log_message("%s: Got table.\n",__FUNCTION__);
        tab->cipso_mfn = cipso_mfn;
        tab->num_cipso_dois = num_cipso_dois;
        tab->cipso_guest_evtchn = cipso_guest_evtchn;
        log_message("%s: Setting up event handling.\n",__FUNCTION__);
        if( setup_event_handling(domid,tab,cipso_evtchn_fd) != 0 ){
                tab->cipso_mfn = 0;
                tab->num_cipso_dois = 0;
                tab->cipso_guest_evtchn = 0;
                retval = -1;
                goto out;
        }
        log_message("%s: Event handling set up successfully.\n",__FUNCTION__);
        retval = 0;
        return retval;

out:
        return retval;
}
```

Figure 4.8. setup_event_handling function

```
/*
 * setup_event_handling:
 * creates an fd for eventchannel, and sends a 'test' event to domain.
 * returns 0 on success, -1 on failure.
 * adds the guest evtchn to
 * cipso_tab.
 */
static int setup_event_handling(domid_t domid, struct cipso_table *tab, int *cipso_evtchn_fd)
{

        int retval = -1;
        int xc_handle;
        xc_dominfo_t dominfo;

        /* xc_domain_getinfo asks for uint32_t */
        uint32_t guest_id = domid;


        /* sanity checks */
        if( domid == 0 || tab == NULL ){
                log_message("Crazy params.\n");
                goto out;
        }
        log_message("%s\n",__FUNCTION__);

        /* get a handle on the evtchn interface */
        retval = tab->xce_handle = xc_evtchn_open();

        if( retval == -1 ){
                log_message("Unable to open evtchn interface!\n");
                goto out;
        }
        log_message("%s: Opened evtchn interface.\n",__FUNCTION__);
        /* get a handle on hypervisor interface */
        retval = xc_handle = xc_interface_open();
        if( retval == -1 ){
                log_message("Unable to open hypervisor interface!\n");
                goto out;
        }
        log_message("%s: Opened hypervisor interface.\n",__FUNCTION__);
        /*
           loop until domain is 'running'
           when 'running' allocate a Dom0 evtchn
           and bind it to domU's event channel
        */
again:
        retval = xc_domain_getinfo(xc_handle, guest_id, 1,&dominfo);

        if( retval == -1 ){
                log_message("Unable to get domain info.\n");
                goto out;
        }
        if( dominfo.running ){
```

```
            /*
               domain is running, now write that we are 'up' - DOM0_UP
            */
            log_message("Guest running.\n");

            log_message("%s: Value of guest_evtchn: %d\n",
                        __FUNCTION__,tab->cipso_guest_evtchn);


            if( xc_evtchn_unmask(tab->xce_handle,tab->cipso_dom0_evtchn) != -1){
                    log_message("Unmasked evtchn.\n");
            }
            else{
                    log_message("Evtchn unmasking failed!\n");
            }

            tab->cipso_dom0_evtchn =
                    xc_evtchn_bind_interdomain(tab->xce_handle,guest_id,tab->cipso_guest_evtchn);

            if( tab->cipso_dom0_evtchn == -1 ){
                    log_message("%s: cipso_dom0_evtchn is -1\n",__FUNCTION__);
            }
            else{
                    log_message("%s: Value of cipso_dom0_evtchn: %d\n",
                                __FUNCTION__,tab->cipso_dom0_evtchn);
            }

            /*
              Now put it in shared info frame so that guest can find it
              and bind to us.
            */
            write_dom0_evtchn_shinfo(tab->domid,tab,dominfo.shared_info_frame);


            if( set_cipso_status(tab,DOM0_UP) != 0 ){
                    log_message("Could not write status.\n");
                    retval = -1;
                    goto out;
            }
            else{
                    xc_evtchn_notify(tab->xce_handle,tab->cipso_dom0_evtchn);
            }
    }
    else{
            goto again;
    }

    /* get the fd that can be used by main */
    *cipso_evtchn_fd = xc_evtchn_fd(tab->xce_handle);
    tab->cipso_evtchn_fd = *cipso_evtchn_fd;
    return 0;
out:
    /* kill domain */
    xc_domain_destroy(xc_handle, guest_id);
    return retval;
}
```

Figure 4.9. write_dom0_evtchn_shinfo function

```
/*
 * write_dom0_evtchn_shinfo:
 * checks whether guest's shared frame info matches ours.
 * return 0 on success, -1 on failure
 */

static int write_dom0_evtchn_shinfo(uint32_t guest_id, struct cipso_table *tab,
                                    xen_pfn_t shinfo_frame)
{
        int retval = -1;
        int xc_handle;
        shared_info_t *sharedp;

        /* sanity checks */
        if( tab == NULL ){
                log_message("Crazy parms.\n");
                goto out;
        }

        /* open hypervisor interface */
        retval = xc_handle = xc_interface_open();
        if( retval == -1 ){
                log_message("Unable to open hypervisor interface!\n");
                goto out;
        }

        /* mmap the shared page into our mem */
        sharedp = xc_map_foreign_range(
                xc_handle,guest_id,XC_PAGE_SIZE,
                PROT_READ|PROT_WRITE,
                shinfo_frame);

        if( sharedp == NULL ){
                log_message("mmap failed.\n");
                retval = -1;
                goto out;
        }
        if( tab->cipso_mfn == sharedp->cipso_mfn && tab->num_cipso_dois == sharedp->num_cipso_dois &&
            sharedp->cipso_status == GUEST_UP ){
                /*
                   everything is ok, update the shared_info
                   with our event channel port
                */
                sharedp->cipso_dom0_evtchn = tab->cipso_dom0_evtchn;
                sharedp->cipso_guest_evtchn = tab->cipso_guest_evtchn;
                retval = 0;
        }

        /* Always remember to munmap!! */
        retval = munmap(sharedp,XC_PAGE_SIZE);
        if( retval != 0 ){
                log_message("Unable to unmap!!!\n");
                goto out;
        }
```

```
        /* Debug prints */                                                    1
                                                                              2
        log_message("%s: Value of cipso_mfn: %x\n",                           3
                    __FUNCTION__,tab->cipso_mfn);                             4
        log_message("%s: Value of num_cipso_dois: %d\n",                      5
                    __FUNCTION__,tab->num_cipso_dois);                        6
                                                                              7
        /* end debug prints */                                                8
                                                                              9
        return retval;          /* success */                                 10
                                                                              11
out:                                                                          12
        return retval;                                                        13
}                                                                             14
                                                                              15
                                                                              16
```

Figure 4.10. Modified version of `xc_linux_build` function (`tools/libxc/xc_dom_compat_linux.c`)

```
int xc_linux_build(int xc_handle, uint32_t domid,                    1
                   unsigned int mem_mb,                              2
                   const char *image_name,                           3
                   const char *initrd_name,                          4
                   const char *cmdline,                              5
                   const char *features,                             6
                   unsigned long flags,                              7
                   unsigned int store_evtchn,                        8
                   unsigned long *store_mfn,                         9
                   unsigned int console_evtchn,                      10
                   unsigned long *console_mfn,                       11
                   unsigned int cipso_evtchn,                        12
                   unsigned int lsm_evtchn)                          13
{                                                                    14
    struct xc_dom_image *dom;                                        15
    int rc;                                                          16
                                                                     17
    xc_dom_loginit();                                                18
    dom = xc_dom_allocate(cmdline, features);                        19
    if ( (rc = xc_dom_kernel_file(dom, image_name)) != 0 )           20
        goto out;                                                    21
    if ( initrd_name && strlen(initrd_name) &&                       22
         ((rc = xc_dom_ramdisk_file(dom, initrd_name)) != 0) )       23
        goto out;                                                    24
                                                                     25
    rc = xc_linux_build_internal(dom, xc_handle, domid,              26
                                 mem_mb, flags,                      27
                                 store_evtchn, store_mfn,            28
                                 console_evtchn, console_mfn,        29
                                 cipso_evtchn, lsm_evtchn);          30
                                                                     31
 out:                                                                32
    xc_dom_release(dom);                                             33
    return rc;                                                       34
}                                                                    35
                                                                     36
```

4.9 Modifications to `libxenctrl`

Xend uses `libxenctrl` to interact with low-level interfaces to the hypervisor when starting up domains. As mentioned earlier, the `linux_build` call is issued. The interface to `libxenctrl` is defined in the python module `xen.lowlevel.xc`, found in the `tools/python/xen/lowlevel/xc/` directory in the xen source tree. The file `xc.c` contains the Python/C API code to some functions in `libxenctrl`. The modified version of the `xc_linux_build` function is shown in figure 4.9. Basically, this function is passed the values of the event channel ports already allocated, for further use later.

The `xc_dom_image` structure holds the relevant information about a guest being built by Xend. Additions

Figure 4.11. Additions to `xc_dom_image` structure (`tools/libxc/xc_dom.h`)

```
1
2
3   struct xc_dom_image {
4       xen_pfn_t cipso_pfn;        /* used by guest */
5       xen_pfn_t cipso_mfn;        /* used by  Dom-0 */
6       uint32_t num_cipso_dois;
7       /* The event channel used for cipso activity */
8       unsigned int cipso_guest_evtchn;
9
10
11      /* mfn for the DOI<->LSM mappings page */
12      xen_pfn_t lsm_pfn;          /* used by guest */
13      xen_pfn_t lsm_mfn;          /* used by Dom-0 */
14      uint32_t num_lsm_mappings;
15      /* The event channel used for SS activity */
16      unsigned int lsm_guest_evtchn;
17
```

to the `xc_dom_image` structure are shown in figure 4.9.

`xc_dom_image` structure also contains the architecture-dependent "hooks", which are functions that depend on the type of guest, whether it is a 32-bit guest, 64-bit guest, and so on. We consider a 32-bit guest, whose architecture-specific hooks are listed in table 4.9 . `xc_linux_build_internal` calls these "hooks" when initializing memory for the guest.

Architecture-dependent Hook	Value
guest_type	xen-3.0-x86_32p
page_shift	PAGE_SHIFT_X86
sizeof_pfn	4
alloc_magic_pages	alloc_magic_pages
count_pgtables	count_pgtables_x86_32_pae
setup_pgtables	setup_pgtables_x86_32_pae
start_info	start_info_x86_32
shared_info	shared_info_x86_32
vcpu	vcpu_x86_32

Table 4.2. Architecture-dependent Hooks - x86 guest with PAE enabled shown (Modified Hooks in Red)

4.9.1 Changes to `alloc_magic_pages`

In the function `alloc_magic_pages` , the "special" pages are allocated to the guest. The modified code is listed in 4.9.1. Similar to the allocation of the `shared_info` and other pages, we "allocate" a "cipso

Figure 4.12. Modified function `alloc_magic_pages (tools/libxc/xc_dom_x86.c)`

```
/*
 * netlabeld modification to alloc_magic_pages:
 * Allocate a special page: 'cipso page' that will
 * contain packed cipso data structures.
 * CIPSO subsystem init in DomU kernel will create
 * CIPSO DOIs using data in cipso page.
 * Allocate a special page: 'lsm page' that will contain
 * DOI<->LSM mappings
 */
static int alloc_magic_pages(struct xc_dom_image *dom)
{
    size_t p2m_size = dom->total_pages * dom->arch_hooks->sizeof_pfn;

    /* allocate phys2mach table */
    if ( xc_dom_alloc_segment(dom, &dom->p2m_seg, "phys2mach", 0, p2m_size) )
        return -1;
    dom->p2m_guest = xc_dom_seg_to_ptr(dom, &dom->p2m_seg);

    /* allocate special pages */
    dom->start_info_pfn = xc_dom_alloc_page(dom, "start info");
    dom->xenstore_pfn = xc_dom_alloc_page(dom, "xenstore");
    dom->console_pfn = xc_dom_alloc_page(dom, "console");
    if ( xc_dom_feature_translated(dom) ){
        dom->shared_info_pfn = xc_dom_alloc_page(dom, "shared info");
    }
    /* Allocate the page containing cipso mappings to the domain */
    dom->cipso_pfn = xc_dom_alloc_page(dom, "cipso page");
    /* Allocate the page containing DOI<->LSM mappings to the domain */
    dom->lsm_pfn = xc_dom_alloc_page(dom, "lsm page");

    dom->alloc_bootstack = 1;
    return 0;
}
```

page" and an "lsm page" which will hold the DOIs and the DOI-to-SELinux context mappings for the guest respectively. `xc_dom_alloc_page` keeps track of virtual addresses within the (not yet running) guest and returns the latest Page Frame Numbers (within the guest) of the pages requested, reserving them. After this step, the next hook allocates page tables and boot stack for the guest.

The layout of the pages allocated is shown in the figure 4.13.

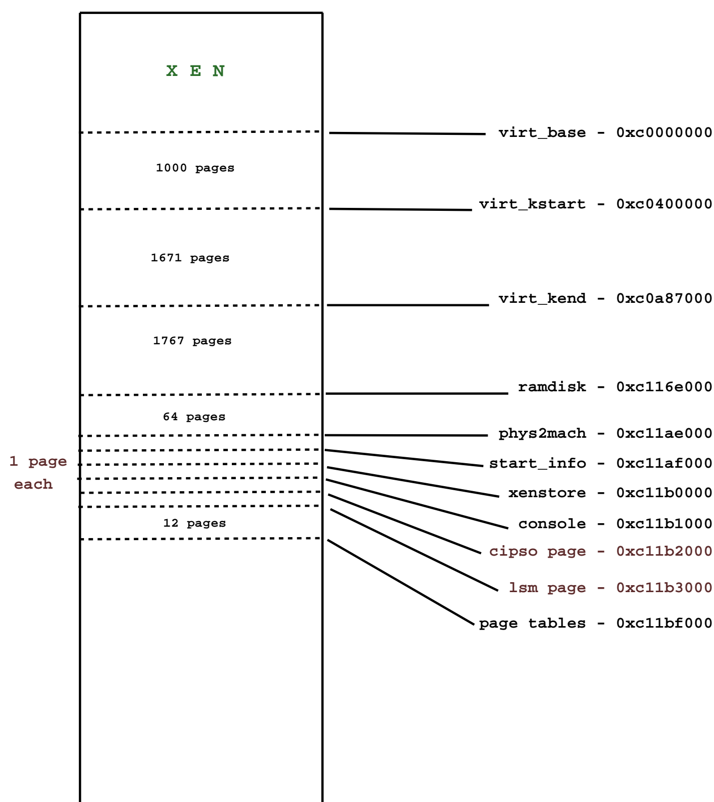

Figure 4.13. Pages allocated to guest (4kB pages)

4.9.2 Guest Mapping Updates

Xen deals exclusively in terms of `Machine Frame Numbers`. Machine frame numbers correspond to the real memory available on the machine. Linux kernels number PFNs contiguously; no "gaps" in PFNs are permitted. Therefore, Linux guests have to be provided with contiguous Page Frame Numbers which, in reality, correspond to *discontiguous* MFNs. These *apparently contiguous* page frames that guests receive are known as "pseudo-physical page frames". Xen maintains `Physical-to-machine (p2m)` tables that contain mappings from PFNs to MFNs. `p2m` tables also have the benefit that they allow both 32-bit and 64-bit guests to be run. Paravirtualized guests' page table entries are populated with `p2m` table entries directly when they are allocated, in the `arch_setup_meminit` hook.

For guests to use them correctly, the "special pages" that were reserved for them during allocation should be available in their page tables. Therefore, in the `arch_setup_bootlate` function, the special pages

are added at specific locations within the guest's pseudo-physical address space. We have modified the `arch_setup_bootlate` function to add our CIPSO and LSM pages into the guest's pseudo-physical address range, as shown in listing 4.9.2. The function `xc_memory_op` , with the arguments shown in the listing, issues the `XENMEM_add_to_physmap` hypercall.

Figure 4.14. Modified `arch_setup_bootlate` function. (`tools/libxc/xc_dom_x86.c`)

```
int arch_setup_bootlate(struct xc_dom_image *dom)
{
    static const struct {
        char *guest;
        unsigned long pgd_type;
    } types[] = {
        { "xen-3.0-x86_32",  MMUEXT_PIN_L2_TABLE},
        { "xen-3.0-x86_32p", MMUEXT_PIN_L3_TABLE},
        { "xen-3.0-x86_64",  MMUEXT_PIN_L4_TABLE},
    };
    unsigned long pgd_type = 0;
    shared_info_t *shared_info;
    xen_pfn_t shinfo;
    int i, rc;

    for ( i = 0; i < sizeof(types) / sizeof(types[0]); i++ )
        if ( !strcmp(types[i].guest, dom->guest_type) )
            pgd_type = types[i].pgd_type;

    if ( !xc_dom_feature_translated(dom) )
    {
        /* paravirtualized guest */
        xc_dom_unmap_one(dom, dom->pgtables_seg.pfn);
        rc = pin_table(dom->guest_xc, pgd_type,
                       xc_dom_p2m_host(dom, dom->pgtables_seg.pfn),
                       dom->guest_domid);
        if ( rc != 0 )
        {
            xc_dom_panic(XC_INTERNAL_ERROR,
                         "%s: pin_table failed (pfn 0x%" PRIpfn ", rc=%d)\n",
                         __FUNCTION__, dom->pgtables_seg.pfn, rc);
            return rc;
        }
        shinfo = dom->shared_info_mfn;
    }
    else
    {
        /* paravirtualized guest with auto-translation */
        struct xen_add_to_physmap xatp;
        int i;

        /* Map shared info frame into guest physmap. */
        xatp.domid = dom->guest_domid;
        xatp.space = XENMAPSPACE_shared_info;
        xatp.idx = 1;
        xatp.gpfn = dom->shared_info_pfn;
        rc = xc_memory_op(dom->guest_xc, XENMEM_add_to_physmap, &xatp);
        if ( rc != 0 )
        {
            xc_dom_panic(XC_INTERNAL_ERROR, "%s: mapping shared_info failed "
                         "(pfn=0x%" PRIpfn ", rc=%d)\n",
                         __FUNCTION__, xatp.gpfn, rc);
            return rc;
        }
```

```
    /* Map cipso mappings frame into guest physmap. */
    xatp.domid = dom->guest_domid;
    xatp.space = XENMAPSPACE_shared_info;
    xatp.idx = 0;
    xatp.gpfn = dom->cipso_pfn;

    /* dom->guest_xc: handle,
       XENMEM_add_to_physmap: command,
       &xatp: argument pointer
    */
    rc = xc_memory_op(dom->guest_xc, XENMEM_add_to_physmap, &xatp);
    if ( rc != 0 )
    {
        xc_dom_panic(XC_INTERNAL_ERROR, "%s: mapping cipso_pfn failed "
                     "(pfn=0x%" PRIpfn ", rc=%d)\n",
                     __FUNCTION__, xatp.gpfn, rc);
        return rc;
    }

    /* Map DOI<->LSM mappings frame into guest physmap. */
    xatp.domid = dom->guest_domid;
    xatp.space = XENMAPSPACE_shared_info;
    xatp.idx = 0;
    xatp.gpfn = dom->lsm_pfn;

    /* dom->guest_xc: handle,
       XENMEM_add_to_physmap: command,
       &xatp: argument pointer
    */
    rc = xc_memory_op(dom->guest_xc, XENMEM_add_to_physmap, &xatp);
    if ( rc != 0 )
    {
        xc_dom_panic(XC_INTERNAL_ERROR, "%s: mapping lsm_pfn failed "
                     "(pfn=0x%" PRIpfn ", rc=%d)\n",
                     __FUNCTION__, xatp.gpfn, rc);
        return rc;
    }


    /* Map grant table frames into guest physmap. */
    for ( i = 0; ; i++ )
    {
        xatp.domid = dom->guest_domid;
        xatp.space = XENMAPSPACE_grant_table;
        xatp.idx = i;
        xatp.gpfn = dom->total_pages + i;
        rc = xc_memory_op(dom->guest_xc, XENMEM_add_to_physmap, &xatp);
        if ( rc != 0 )
        {
            if ( (i > 0) && (errno == EINVAL) )
            {
                xc_dom_printf("%s: %d grant tables mapped\n", __FUNCTION__,
                              i);
                break;
            }
            xc_dom_panic(XC_INTERNAL_ERROR,
                         "%s: mapping grant tables failed " "(pfn=0x%"
                         PRIpfn ", rc=%d)\n", __FUNCTION__, xatp.gpfn, rc);
            return rc;
        }
    }
    shinfo = dom->shared_info_pfn;
}
```

```
    /* setup shared_info page */                                                    1
    xc_dom_printf("%s: shared_info: pfn 0x%" PRIpfn ", mfn 0x%" PRIpfn "\n",        2
                  __FUNCTION__, dom->shared_info_pfn, dom->shared_info_mfn);        3
    shared_info = xc_map_foreign_range(dom->guest_xc, dom->guest_domid,             4
                                       PAGE_SIZE_X86,                               5
                                       PROT_READ | PROT_WRITE,                      6
                                       shinfo);                                     7
    if ( shared_info == NULL )                                                      8
        return -1;                                                                  9
    dom->arch_hooks->shared_info(dom, shared_info);                                 10
    munmap(shared_info, PAGE_SIZE_X86);                                             11
                                                                                    12
    return 0;                                                                       13
}                                                                                   14
                                                                                    15
                                                                                    16
```

4.9.3 Copying CIPSO Mappings into the Guest's Memory

The final architecture-dependent hook that is invoked during guest startup is `shared_info`. For a 32-bit guest, this maps to the function `shared_info_x86_32`. As the name suggests, this function initializes the variables in the `shared_info` structure that will be used by the guest and Dom-0. We modify this function to do do the following:

- We initialize the event-channel port variables in `shared_info` to those that we have already allocated within the guest.
- We copy the initial DOIs for the guest into the CIPSO page we have allocated earlier, by calling the function `setup_cipso_page`. To do this, we send a `SEND_CIPSO_DOIS` request to Netlabeld. We have already discussed how Netlabeld handles this request in section 4.8.2. If Netlabeld sends a `SEND_CIPSO_DOIS_SUCCESS` response, the response contains the "packed" DOI definitions that will be read in by the guest. We obtain a pointer to the CIPSO page using the `xc_dom_pfn_to_ptr` and copy the packed DOI definitions received into the page. The modified `shared_info_x86_32` function and the `setup_cipso_page` function are listed in 4.9.3 and 4.9.3 respectively.

Figure 4.15. Modified `shared_info_x86_32` function. (`tools/libxc/xc_dom_x86.c`)

```
static int shared_info_x86_32(struct xc_dom_image *dom, void *ptr)
{
    shared_info_x86_32_t *shared_info = ptr;
    int i;

    memset(shared_info, 0, sizeof(*shared_info));
    for ( i = 0; i < MAX_VIRT_CPUS; i++ )
        shared_info->vcpu_info[i].evtchn_upcall_mask = 1;

    /* Doing the cipso page setup for the guest here */
    if( setup_cipso_page(dom,&(shared_info->num_cipso_dois)) != 0 ){
        return -1;
    }
    /*
       Put eventchannel port values in shared_info so that both
       Dom-0 and guest can know where to find it
       cipso_guest_evtchn has been allocated already in
       the python code via xend
       netlabeld will put the value of cipso_dom0_evtchn in
       shared_info later.
    */
    shared_info->cipso_guest_evtchn = dom->cipso_guest_evtchn;
    shared_info->lsm_guest_evtchn = dom->lsm_guest_evtchn;

    dom->cipso_mfn = xc_dom_p2m_guest(dom,dom->cipso_pfn);
    dom->lsm_mfn = xc_dom_p2m_guest(dom,dom->lsm_pfn);
    shared_info->cipso_pfn = dom->cipso_pfn;
    shared_info->lsm_pfn = dom->lsm_pfn;
    shared_info->cipso_mfn = dom->cipso_mfn;
    shared_info->lsm_mfn = dom->lsm_mfn;
    dom->num_cipso_dois = shared_info->num_cipso_dois;

    /* printing out values for debug */
    xc_dom_printf("%s: REMOTE DOMAIN: %d\n",__FUNCTION__,dom->guest_domid);
    xc_dom_printf("%s: CIPSO MFN: %lx\n",__FUNCTION__,dom->cipso_mfn);
    xc_dom_printf("%s: CIPSO GUEST EVENTCHANNEL: %d\n",
                  __FUNCTION__,shared_info->cipso_guest_evtchn);
    xc_dom_printf("%s: LSM MFN: %lx\n",__FUNCTION__,dom->lsm_mfn);
    xc_dom_printf("%s: LSM GUEST EVENTCHANNEL: %d\n",
                  __FUNCTION__,shared_info->lsm_guest_evtchn);
    /* end debug prints */

    return 0;
}
```

4.9.4 Sending Guest Details to Netlabeld

The final step of the domain-building process is to “release” the domain. This is achieved via the function `xc_dom_release`. The guest pages that have been mapped into Dom-0 are unmapped, and the internal

Figure 4.16. setup_cipso_page function. (tools/libxc/xc_dom_x86.c)

```
/*
 * setup_cipso_page: talks to netlabeld.
 * Issues request to send packed cipso data structures.
 * Copies the cipso data structures into 'cipso page'
 *
 */

static int setup_cipso_page(struct xc_dom_image *dom, uint32_t *num_cipso_dois)
{
    uint32_t msgtype,msglen;
    struct sockaddr_un netlabeld_sock;
    int msg_sock;
    int len;
    char recvbuf[1024];
    void *cipso_mapping_data;
    void *send_str;
    void *original_str;
    char *dataptr;
    void *destptr;
    uint32_t mapping_data_len;
    ssize_t send_len, recv_len;
    uint32_t num_dois;

    /* using this just to get rid of warnings */
    int asprintf_res = 0;

    struct netlabel_sockmsg *msg;

    /* connect to netlabeld */

    msg_sock = socket(AF_UNIX,SOCK_STREAM,0);

    if( msg_sock == -1 ) return -1;

    netlabeld_sock.sun_family = AF_UNIX;
    strcpy(netlabeld_sock.sun_path, NETLABELD_SOCKPATH);

    len = strlen(netlabeld_sock.sun_path)+sizeof(netlabeld_sock.sun_family);

    if( connect(msg_sock, (struct sockaddr *)&netlabeld_sock, len) == -1 ){
        xc_dom_printf("%s:Cannot connect to netlabeld\n",__FUNCTION__);
        return -1;
    }

    /* connected now */
    msg = malloc(sizeof(struct netlabel_sockmsg));

    if( msg == NULL ){
        xc_dom_printf("%s: Cannot allocate netlabel_sockmsg.\n",__FUNCTION__);
        return -ENOMEM;
    }

```

```
/*
   First send request to netlabeld
   msgtype:SEND_CIPSO_DOIS
   msglen: 0
*/

msgtype = SEND_CIPSO_DOIS;
msglen=0;
msg->msg = NULL;
asprintf_res = asprintf(&msg->type,"%u",msgtype);
asprintf_res = asprintf(&msg->domid,"%u",dom->guest_domid);
asprintf_res = asprintf(&msg->len,"%u",msglen);

send_len = strlen(msg->type)+1+
    strlen(msg->domid)+1+
    strlen(msg->len)+1;
/*
   prepare buffer for sending
   Preserve start of msg in original_str
   mempcpy moves pointer.
*/
original_str = send_str = malloc(send_len);
send_str = mempcpy(send_str,msg->type,strlen(msg->type)+1);
send_str = mempcpy(send_str,msg->domid,strlen(msg->domid)+1);
send_str = mempcpy(send_str,msg->len,strlen(msg->len)+1);
/* XXX: What is this doing here? */
memcpy(send_str,&msg,msglen);

if( (send_len = send(msg_sock,original_str,send_len,0)) == -1 ){
    xc_dom_printf("%s: Send error.\n",__FUNCTION__);
    return -1;
}
/*
   message sent. receive the response from netlabeld
*/
memset(recvbuf,0,1024);
recv_len = recv(msg_sock,recvbuf,1024,0);
close(msg_sock);
if( recv_len == 0 ){
    return -1;
}

dataptr = recvbuf;
/*
   Check whether response is 'success'
   HACK: domid (second field of message) actually refers to
   the number of dois sent in the page.
*/
memcpy(&msgtype,(void *)dataptr,sizeof(uint32_t));
if( msgtype != SEND_CIPSO_DOIS_SUCCESS ){
    xc_dom_printf("%s: Request to send CIPSO DOIs did not succeed.\n",
                  __FUNCTION__);
    return -1;
}
dataptr+=sizeof(uint32_t);
memcpy(&num_dois,(void *)dataptr,sizeof(uint32_t));
/* num_cipso_dois will be used to unpack DOIs */
*num_cipso_dois =  num_dois;
/* mapping_data_len: Size of packed cipso structures */
dataptr+= sizeof(uint32_t);
memcpy(&mapping_data_len,(void *)dataptr,sizeof(uint32_t));

xc_dom_printf("%s: num_cipso_dois: %d\t mapping_data_len: %d.\n",
              __FUNCTION__,*num_cipso_dois, mapping_data_len);

```

```
    cipso_mapping_data = malloc(mapping_data_len);                                   1
    if( cipso_mapping_data == NULL ){                                                2
        xc_dom_printf("%s: Cannot allocate cipso_mapping_data.\n",__FUNCTION__);     3
        return -ENOMEM;                                                              4
    }                                                                                5
    /*                                                                               6
       Now move dataptr past len to msg                                              7
       Copy packed structures into cipso_mapping_data                                8
    */                                                                               9
    dataptr+= sizeof(uint32_t);                                                      10
    memcpy(cipso_mapping_data,dataptr,mapping_data_len);                             11
    /*                                                                               12
       We now have the mappings in place.                                            13
       Get a usable pointer using xc_dom_pfn_to_ptr.                                 14
       Copy the cipso DOIs into the area.                                            15
    */                                                                               16
    destptr = xc_dom_pfn_to_ptr(dom, dom->cipso_pfn, 1);                             17
    if( destptr != NULL ){                                                           18
        memcpy(destptr,cipso_mapping_data,mapping_data_len);                         19
    }                                                                                20
    else{                                                                            21
        xc_dom_printf("%s: xc_dom_pfn_to_ptr returned NULL!\n Cannot copy CIPSO DOIS.\n",  22
                      __FUNCTION__);                                                 23
        return -1;                                                                   24
    }                                                                                25
                                                                                     26
    return 0;                                                                        27
}                                                                                    28
                                                                                     29
                                                                                     30
```

data structures related to the domain (`xc_dom_image` etc) are freed. We modify this function to send relevant data about the guest to Netlabeld. `xc_dom_release` makes a call to `send_nlbl_dom_data`, which sends the guest details to Netlabeld. The listings 4.9.4 and 4.9.4 show the relevant functions.

Figure 4.17. Modified `xc_dom_release` function. (`tools/libxc/xc_dom_core.c`)

```
/*
 * xc_dom_release:
 * Netlabeld modification:
 * Sends DomU data to netlabeld
 * via call to send_nlbl_dom_data(dom)
 */
void xc_dom_release(struct xc_dom_image *dom)
{
    xc_dom_printf("%s: called\n", __FUNCTION__);
    if ( dom->phys_pages )
        xc_dom_unmap_all(dom);
    if( send_nlbl_dom_data(dom) != 0 ){
        xc_dom_panic(XC_INTERNAL_ERROR,
                     "%s: UNABLE TO SEND DOMU DATA TO NETLABELD\n", __FUNCTION__);
    }
    xc_dom_free_all(dom);
    free(dom);
}
```

Figure 4.18. send_nlbl_dom_data function. (tools/libxc/xc_dom_core.c)

```
/*
 * send_nlbl_dom_data:
 * Talks to netlabeld.
 * Sends Data about DomU's cipso_mfn,
 * cipso_guest_evtchn, num_cipso_dois
 * to netlabeld. Used to bind eventchannels in Dom0
 * to those of guest after it is in 'running' state.
 */

static int send_nlbl_dom_data(struct xc_dom_image *dom)
{

    uint32_t msgtype,msglen;
    struct sockaddr_un netlabeld_sock;
    int msg_sock;
    int len;
    ssize_t send_len;
    int asprintf_res = 0;
    struct netlabel_sockmsg *msg;
    char *original_str;
    char *send_str;
    void *tmp;

    /* Connect to netlabeld */
    msg_sock = socket(AF_UNIX,SOCK_STREAM,0);

    if( msg_sock == -1 ){
        return -1;
    }

    netlabeld_sock.sun_family = AF_UNIX;

    strcpy(netlabeld_sock.sun_path, NETLABELD_SOCKPATH);

    len = strlen(netlabeld_sock.sun_path)+sizeof(netlabeld_sock.sun_family);

    if( connect(msg_sock, (struct sockaddr *)&netlabeld_sock, len) == -1 ){
        xc_dom_printf("%s:Cannot connect to netlabeld\n",__FUNCTION__);
        return -1;
    }

    /* connected now */
    msg = (struct netlabel_sockmsg *)malloc(sizeof(struct netlabel_sockmsg));

    if( msg == NULL ){
        xc_dom_printf("%s: Cannot allocate netlabel_sockmsg.\n",__FUNCTION__);
        return -ENOMEM;
    }

    /*
       First send request to netlabeld
       msgtype: ADD_DOI_LIBXC_DATA
       domid: guest's domid
       msglen: three data items sent.
       asprintf used because the above three items are sent as strings.
    */

    msgtype = ADD_DOI_LIBXC_DATA;
    msglen = sizeof(dom->cipso_mfn)+sizeof(dom->num_cipso_dois)+
        sizeof(dom->cipso_guest_evtchn)+sizeof(dom->lsm_guest_evtchn)+
        sizeof(dom->lsm_mfn);
    asprintf_res = asprintf(&msg->type,"%u",msgtype);
    asprintf_res = asprintf(&msg->domid,"%u",dom->guest_domid);
    asprintf_res = asprintf(&msg->len,"%u",msglen);

```

```
    send_len = strlen(msg->type)+1+
         strlen(msg->domid)+1+
         strlen(msg->len)+1+
         msglen;

    /* copy over the mfn, number of dois, and dom0's evtchn into msg */
    tmp = msg->msg = malloc(msglen);
    if( msg->msg == NULL ){
        return -1;
    }
    /* mempcpy: 'join' data */
    tmp = mempcpy(tmp,&dom->cipso_mfn,sizeof(dom->cipso_mfn));
    tmp = mempcpy(tmp,&dom->num_cipso_dois,sizeof(dom->num_cipso_dois));
    tmp = mempcpy(tmp,&dom->cipso_guest_evtchn,sizeof(dom->cipso_guest_evtchn));
    tmp = mempcpy(tmp,&dom->lsm_mfn,sizeof(dom->lsm_mfn));
    tmp = mempcpy(tmp,&dom->lsm_guest_evtchn,sizeof(dom->lsm_guest_evtchn));

    /*
       prepare buffer for sending
       Preserve start location in original_str.
       mempcpy moves pointer.
     */
    original_str = send_str = malloc(send_len);
    send_str = mempcpy(send_str,msg->type,strlen(msg->type)+1);
    send_str = mempcpy(send_str,msg->domid,strlen(msg->domid)+1);
    send_str = mempcpy(send_str,msg->len,strlen(msg->len)+1);
    send_str = mempcpy(send_str,msg->msg,msglen);

    if( (send_len = send(msg_sock,original_str,send_len,0)) == -1 ){
        xc_dom_printf("%s: Send error.\n",__FUNCTION__);
        goto out;
    }

    return 0;
out:
    if( msg->msg ) free(msg->msg);
    if( original_str ) free(original_str);
    return -1;
}
```

Chapter 5

Results

5.1 Experiment Setup

We carry out our experiments on two hosts `xabi` and `oxygen` both connected via a switch and ethernet, as shown in figure 5.1. The hosts are one hop away from each other. Both machines host two guests each, a Dom-0 and an unprivileged guest on Xen. To test local communication scenarios, we run two unprivileged guests along with a Dom-0 on `xabi`. Both `xabi` and `oxygen` are Dell Precision 380 machines with dual-core Intel 64-bit processor , 2.80 GHz. The output of `/proc/cpuinfo` is shown below:

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 15
model           : 6
model name      : Intel(R) Pentium(R) D CPU 2.80GHz
stepping        : 2
cpu MHz         : 2793.234
cache size      : 2048 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 6
wp              : yes
flags           : fpu tsc msr pae mce cx8 apic mtrr mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
lm constant_tsc pni monitor ds_cpl vmx cid cx16 xtpr lahf_lm
bogomips        : 5592.91
clflush size    : 64

processor       : 1
vendor_id       : GenuineIntel
cpu family      : 15
```

```
model           : 6
model name      : Intel(R) Pentium(R) D CPU 2.80GHz
stepping        : 2
cpu MHz         : 2793.234
cache size      : 2048 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 6
wp              : yes
flags           : fpu tsc msr pae mce cx8 apic mtrr mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe
lm constant_tsc pni monitor ds_cpl vmx cid cx16 xtpr lahf_lm
bogomips        : 5592.91
clflush size    : 64
```

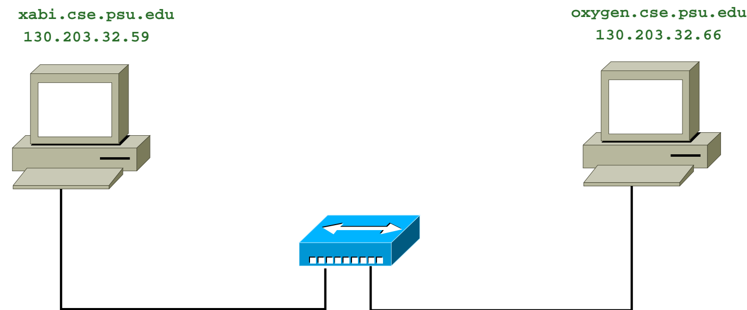

Figure 5.1. Network Topology

We run Xen version 3.1.0 obtained from the XenSource website, and the VMs are derived from Fedora Core 7, they are patched versions of the mainline Linux Kernel version 2.6.20. The base RPM package in Fedora Core 7 is `kernel-xen-2.6.20-2931.fc7xen`. The Xen source code is patched with our modifications described in our previous chapter and is compiled with the ACM feature and PAE (Physical Address Extension) enabled, while the Linux code is patched with our modifications described in the previous chapter and compiled with support for Labeled IPSec, netlabel and PAE enabled. Further, the Xen network configuration is set to `network-route` and `vif-route` which enable traffic from/to guests to be routed via Dom-0. The SELinux policy configuration in all the guests is the default targeted MLS policy, run in permissive mode. This configuration defines 16 sensitivity levels (`s0-s15`) and 1024 categories (`c0-c1023`). Also, offloading of higher-level protocol functions to the card is disabled via the commands `ethtool -K eth0 tx off; ethtool -K eth0 rx off`, to prevent retransmissions due to faulty checksums inserted by the (buggy) Dom-0 network driver. The command is run on the virtual network interfaces (`vifs`) of the unprivileged guests also. The Dom-0 has ACM simple types that are a super set of all the simple types of the guests on the system, and the primary policy enforced is Simple

Type Enforcement.

For benchmarking, we run the microbenchmark `lmbench` [21], obtained from

`http://sourceforge.net/projects/lmbench`. The version of `lmbench` used is 3.0-a9. `lmbench` is widely used; it has been used in previous efforts such as LSM [31] and is used routinely by the Linux developer community to measure kernel performance.

5.2 Netlabel processing overhead

We have discussed netlabel processing of packets in previous chapters. We conducted experiments to measure the impact of labeling individual packets using netlabel. The experiments test labeling impact in two scenarios, guests communicating on the same VMM and communicating across VMMs, over a network. Figures 5.2 and 5.3 depict two collocated domains communicating with and without labeled packets, respectively. Figures 5.4 and 5.5 depict two domains located on physically separate hosts communicating with and without labeled packets, respectively. The benchmark application is confined via SELinux by running it using the `runcon` command; all domains are configured to send labeled packets by default in figures 5.3 and 5.5. Note that in the labeled case, packets undergo IP Options processing at three points: at the unprivileged guests that send and receive labeled packets, and at the Dom-0 that *forwards* packets.

We measure TCP connection latency, latency to send TCP messages, UDP messages and bandwidth over TCP connections. For latency measurements for TCP and UDP sockets, we vary the message size (m) parameter to the benchmark, as this parameter decides the number of packets sent through the sockets and the associated netlabel processing; the same holds for TCP bandwidth measurements too.

We observed that the TCP connection latency is not impacted much due to netlabel processing. In the case of connection latency, labeled connections are slower than unlabeled connections by 3% (local communication) -8% (remote communication). For TCP latency, the cost difference is about 10% (remote

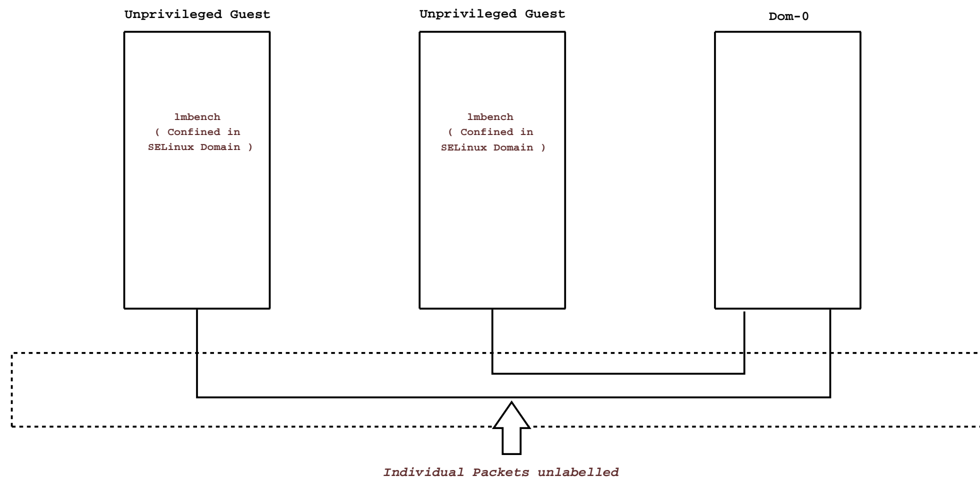

Figure 5.2. Unlabeled local communication

communication) - 11% (local communication). For UDP latency, the cost difference is about 1% (remote communication and local communication). For TCP bandwidth, the cost difference is about 1% (remote communication) - 11% (local communication). In some cases (figures 5.8 and 5.9), there is a large difference (about 35% for very large TCP segment size), but this is more the exception than the norm. Even Though this is a performance anomaly, this has to be explained further. We believe that this large gap in performance for large TCP segment sizes (32K and 64K) is because of the corresponding increase in the number of packets that are generated.

We conclude from this experiment that the overhead of netlabel processing is small, compared to unlabeled packet processing, for the extra functionality offered by labeled packets.

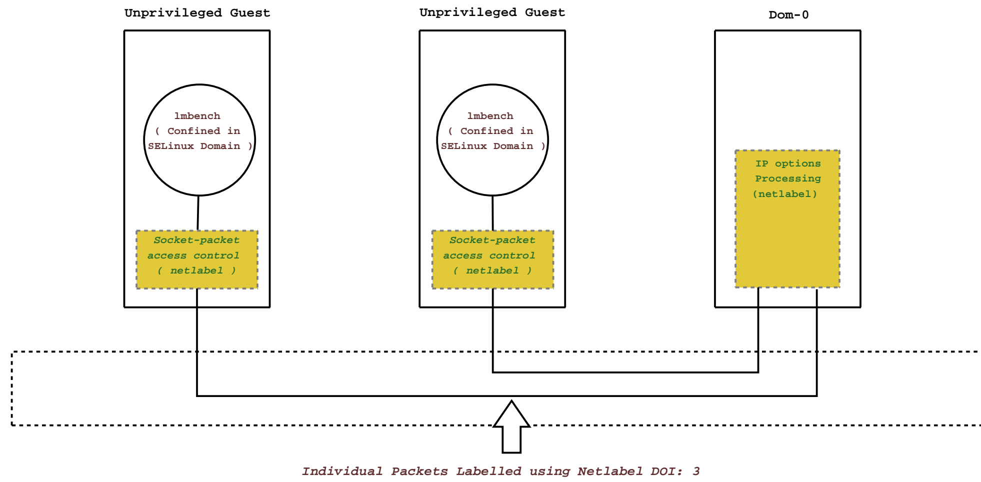

Figure 5.3. Local communication with Netlabel

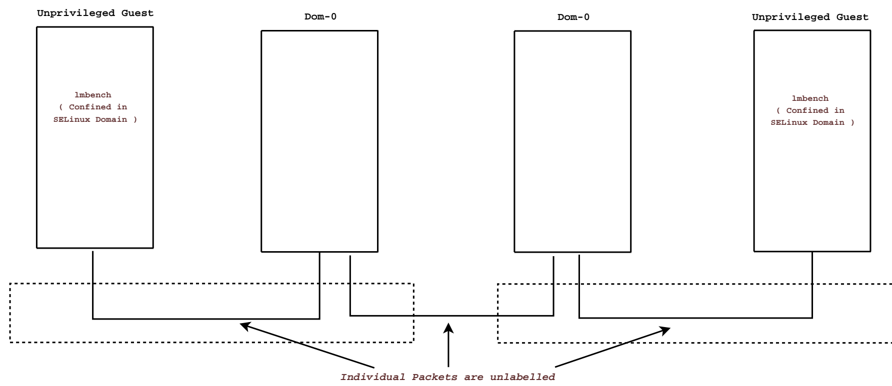

Figure 5.4. Unlabeled remote communication

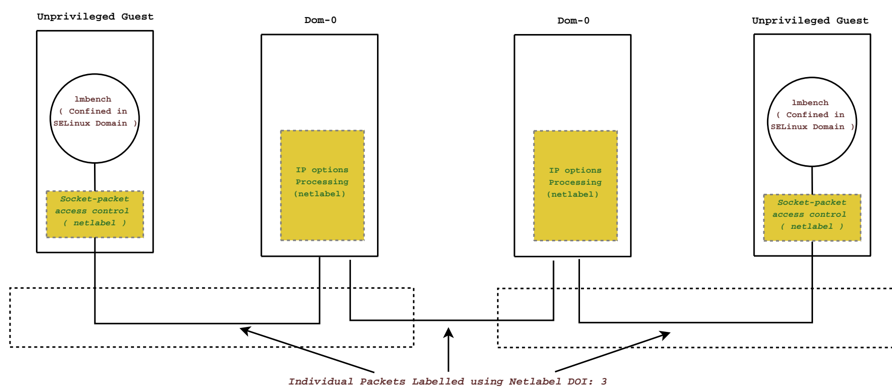

Figure 5.5. Remote communication with Netlabel

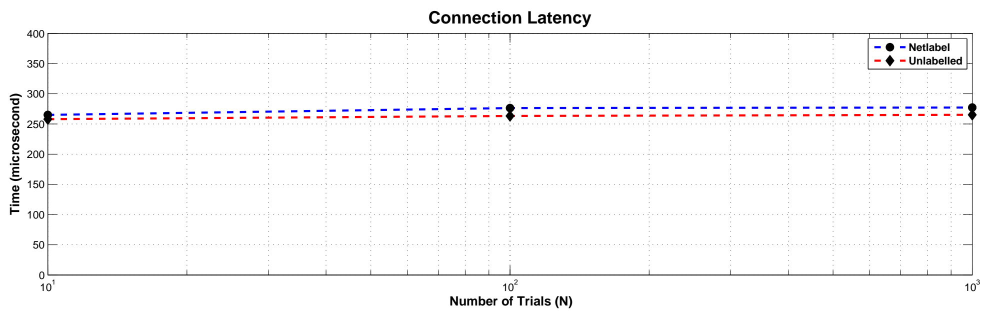

Figure 5.6. TCP connection latency, local communication

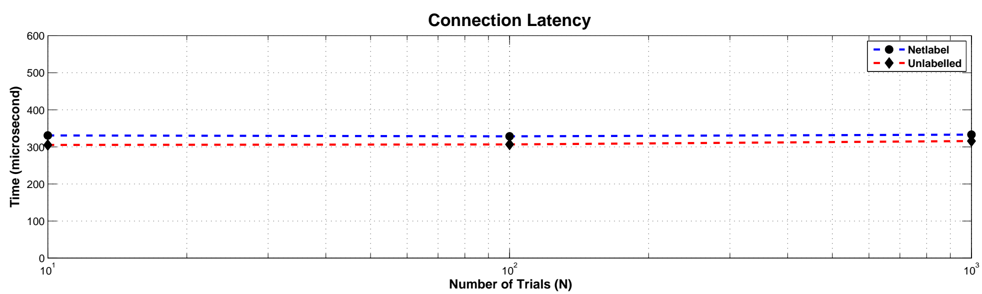

Figure 5.7. TCP connection latency, remote communication

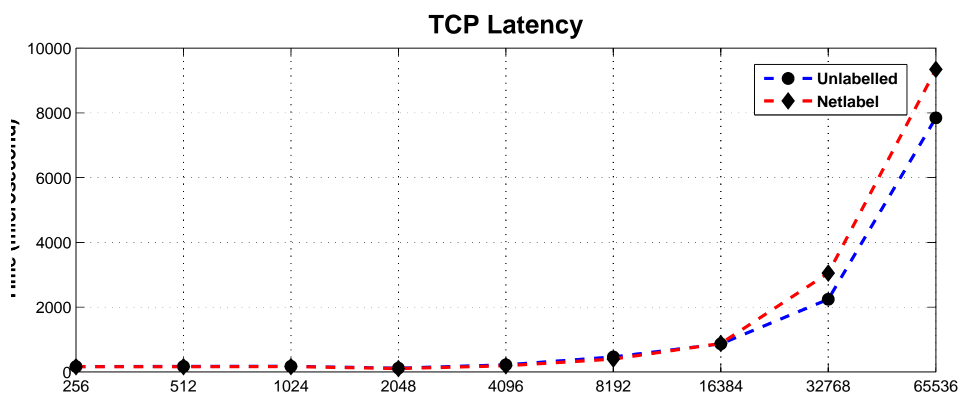

Figure 5.8. TCP latency, local communication

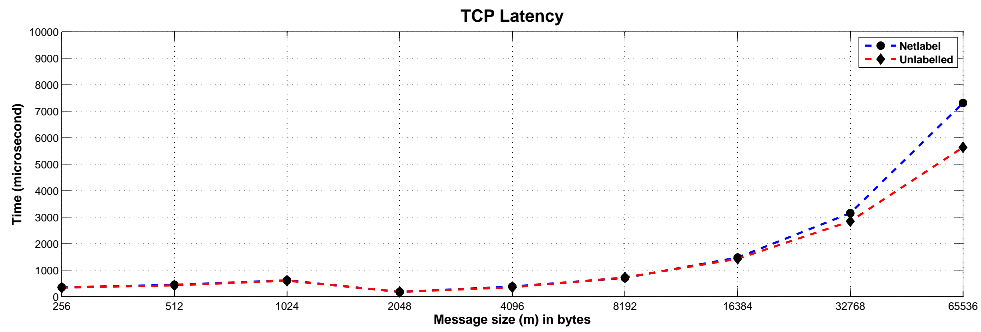

Figure 5.9. TCP latency, remote communication

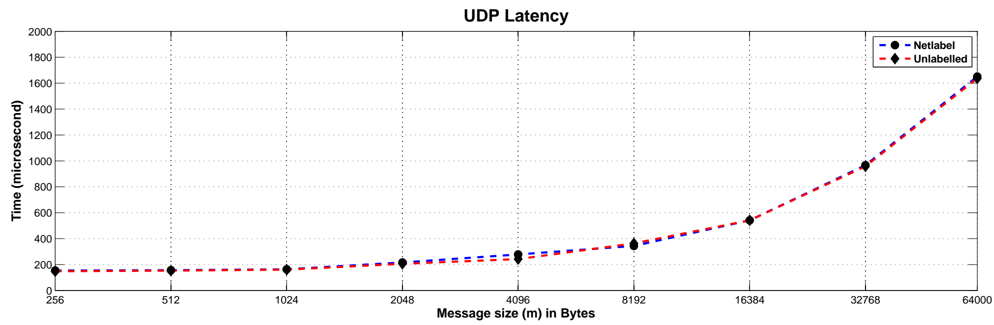

Figure 5.10. UDP latency, local communication

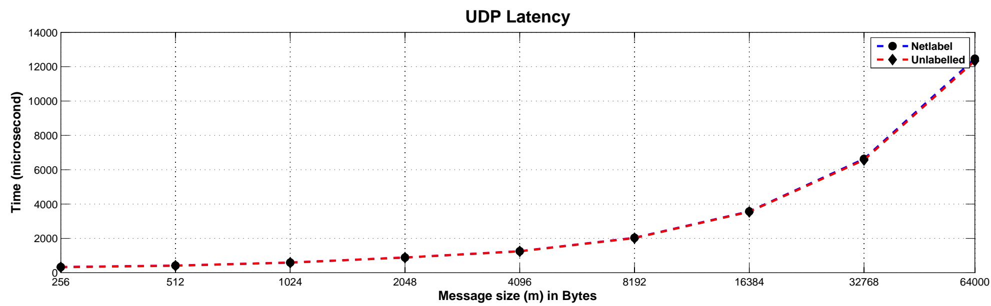

Figure 5.11. UDP latency, remote communication

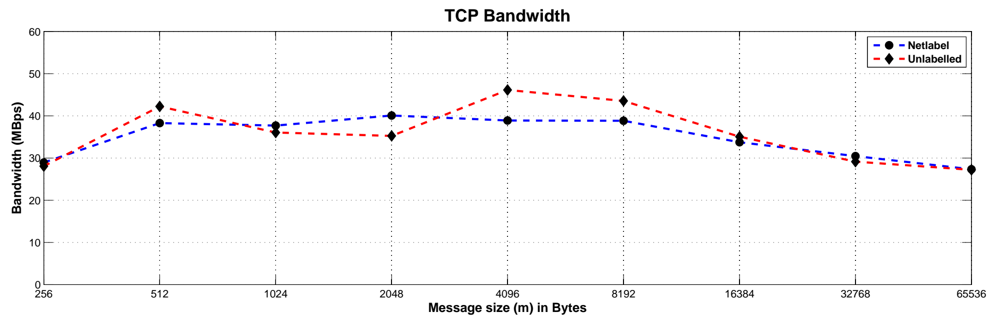

Figure 5.12. TCP Bandwidth, local communication

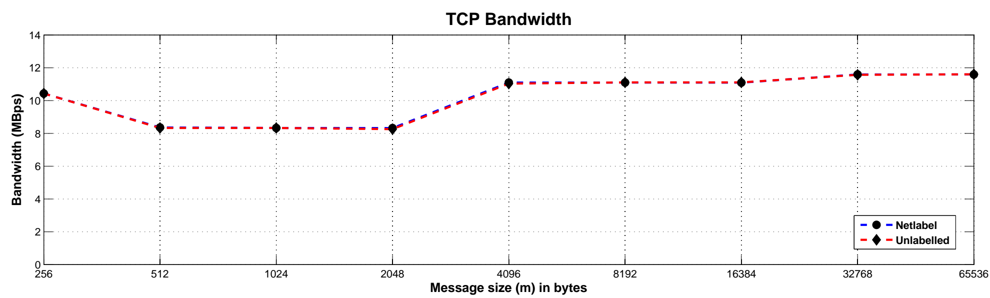

Figure 5.13. TCP Bandwidth, remote communication

5.3 Using Netlabel to Carry Authorization Data

Our contention is that using Netlabel to carry authorization data, while relying on protection mechanisms in Xen to protect confidentiality of this data, between collocated guests improves performance (compared to using costly IPSec mechanisms to protect confidentiality of authorization data). To check this hypothesis, we carried out measurements in three usage scenarios. Figure 5.14 depicts two domains located on physically distinct machines, that transmit authorization data via Labeled IPSec. To achieve this, labeled IPSec tunnels are used between the unprivileged guest and the Dom-0 on each machine. The Dom-0's also use labeled IPSec tunnels to transmit authorization data between each other. Figure 5.15 depicts Netlabel being used to transmit authorization data between the unprivileged guest and Dom-0, and a cryptographically protected "Vanilla" IPSec tunnel being used to transmit authorization data between the Dom-0's. Figure 5.16 depicts Netlabel being used to transmit authorization data between the unprivileged guest and Dom-0, while a single labeled IPSec tunnel is used to transfer authorization data between the Dom-0's. The individual packets that are protected by the labeled IPSec tunnel carry authorization data in the form of IP Options in them.

For TCP connection latency, using cryptographic protection via IPSec and the associated processing overhead costs 60% more than our proposal – using Netlabel to carry authorization data between collocated domains. For TCP latency, IPSec throughout costs anywhere from 81%-165% more than our proposed method. For UDP latency, using IPSec throughout costs 83%-324% more than our proposed method. TCP bandwidth is also affected adversely: using IPSec, only about 20% of bandwidth possible using our proposal is achieved.

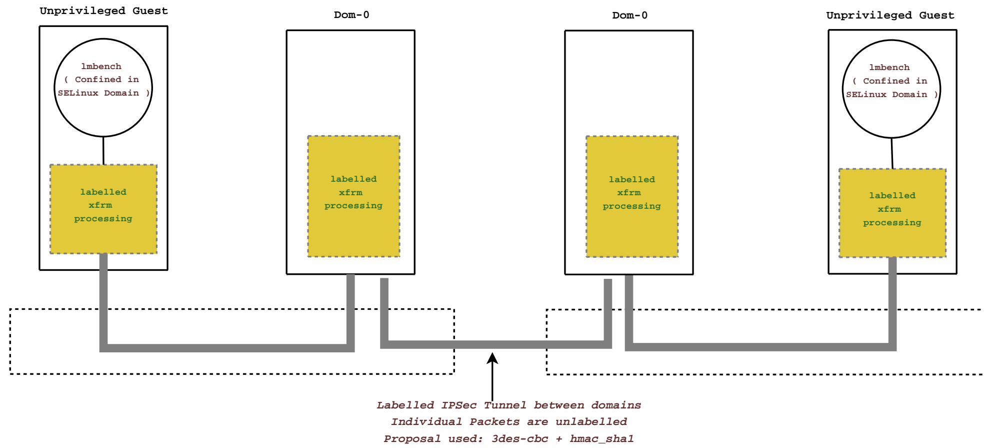

Figure 5.14. Remote communication with Labeled IPSec,tunnel mode throughout

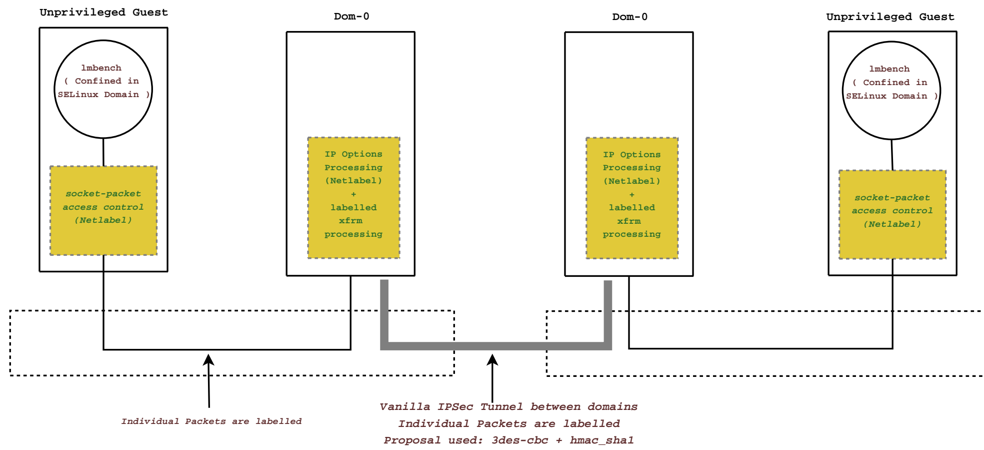

Figure 5.15. Remote communication with Vanilla IPSec,tunnel mode between Dom-0's

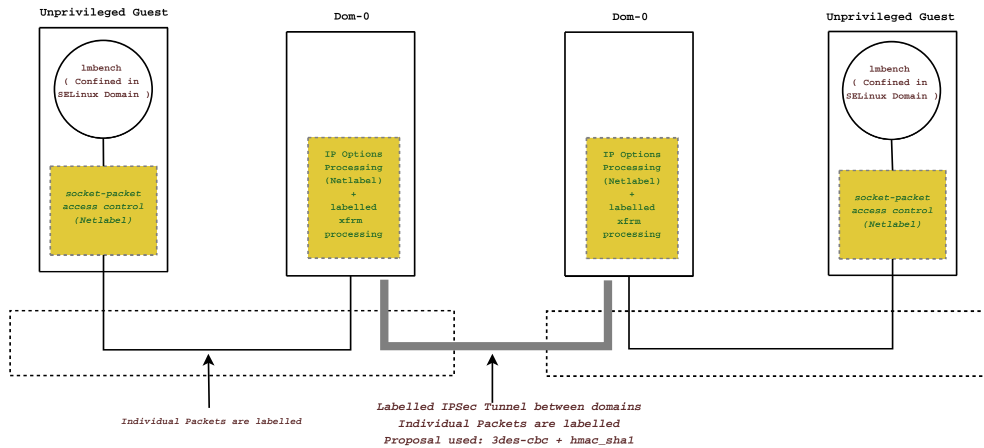

Figure 5.16. Remote communication with Labeled IPSec,tunnel mode between Dom-0's

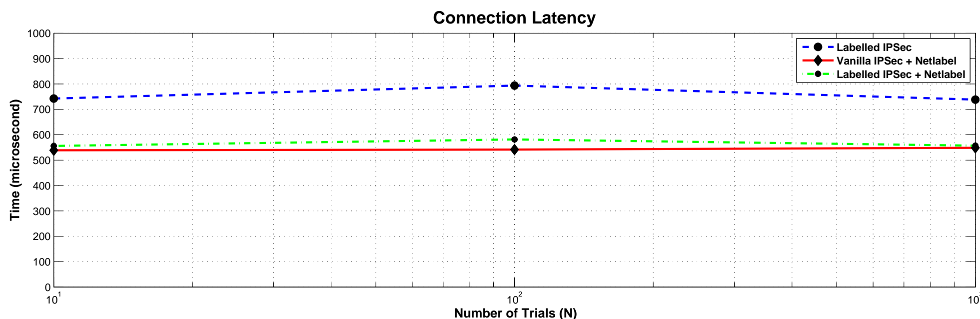

Figure 5.17. Connection latency, remote communication,tunnel mode

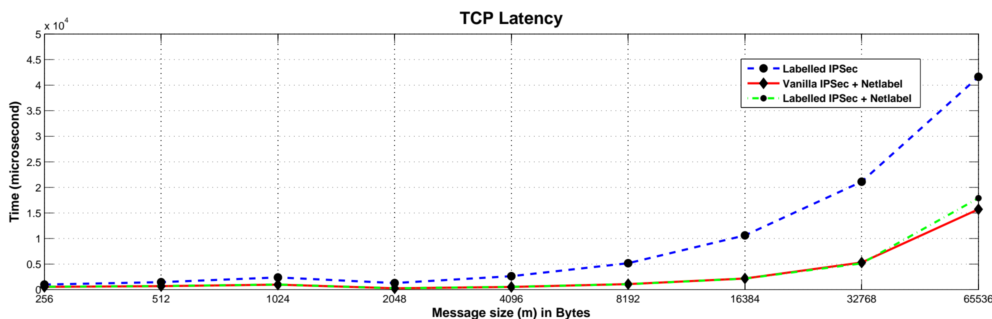

Figure 5.18. TCP latency, remote communication,tunnel mode

5.4 Using IPSec in transport mode to carry authorizations

If both the guests that communicate have IPSec processing capability, then IPSec can be used in "transport" mode to convey authorization data and protect its confidentiality. The difference between this method and those shown in the previous section is that the Dom-0 is unable to inspect packets sent by the guests, it merely forwards packets. This reduces costs slightly by avoiding IPSec processing on the part

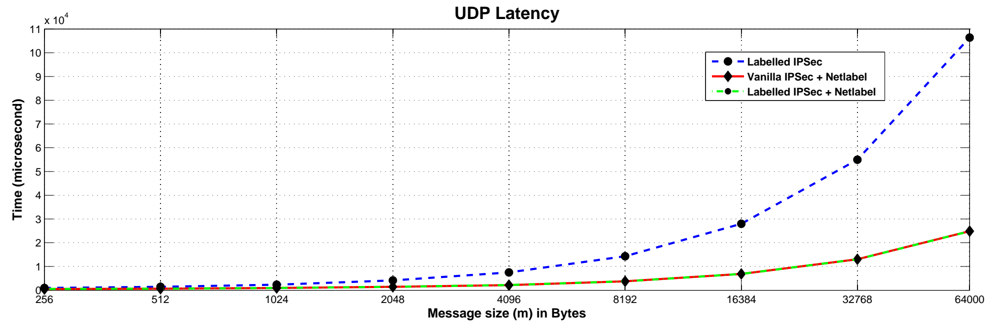

Figure 5.19. UDP latency, remote communication,tunnel mode

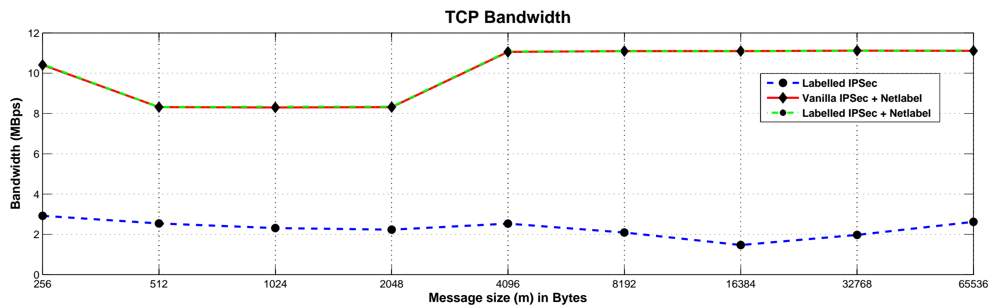

Figure 5.20. TCP bandwidth, remote communication,tunnel mode

of Dom-0. The experiment scenarios are depicted in figures 5.21 - 5.24.

We find that compared to using tunnel mode processing throughout, end-to-end IPSec connections are cheaper, but they still do not outperform netlabel performance for transmitting authorization data among collocated domains (in fact, they cost almost double in all cases). If end-to-end IPSec connections are used to transfer authorization data between domains located on *different* hosts, the difference between the performance seen using our proposal is minimal. But, note that we have not measured the CPU usage

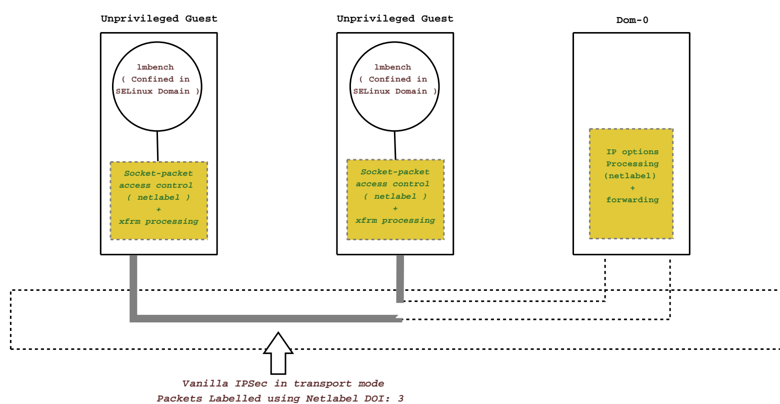

Figure 5.21. End-to-end local communication, vanilla IPSec+Netlabel

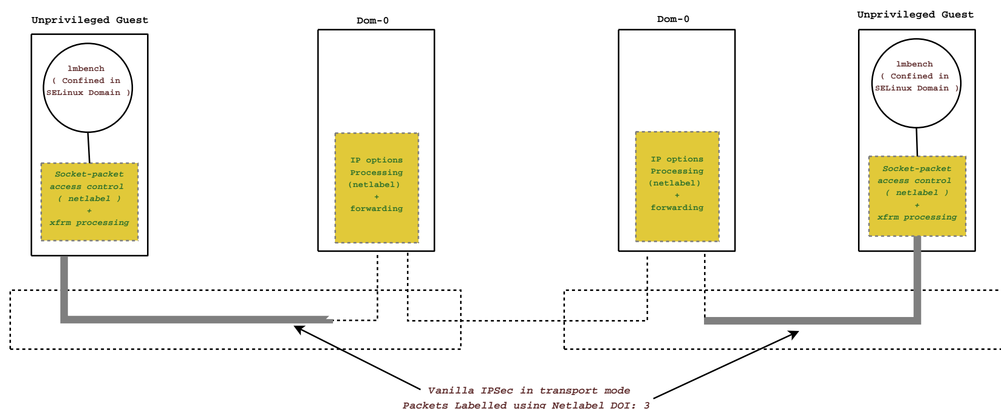

Figure 5.22. End-to-end remote communication, vanilla IPSec+Netlabel

by domains in these scenarios. We believe that although end-to-end connections can be used to transmit authorization data among guests on different hosts (at a marginally worst impact on latency), the CPU costs will still be higher for IPSec processing.

5.5 Conclusions and Further Work

The experiments discussed in the previous section clearly demonstrate the following:

1. Netlabel offers the same functionality as Labeled IPSec as far as conveying authorization data is concerned. As can be expected, there is a certain amount of overhead to do this, compared to a kernel that does not offer this functionality. But, the performance measurements clearly show that

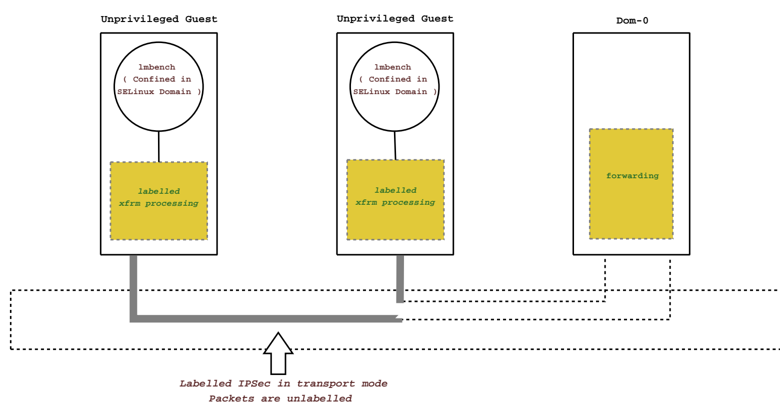

Figure 5.23. End-to-end local communication, Labeled IPSec

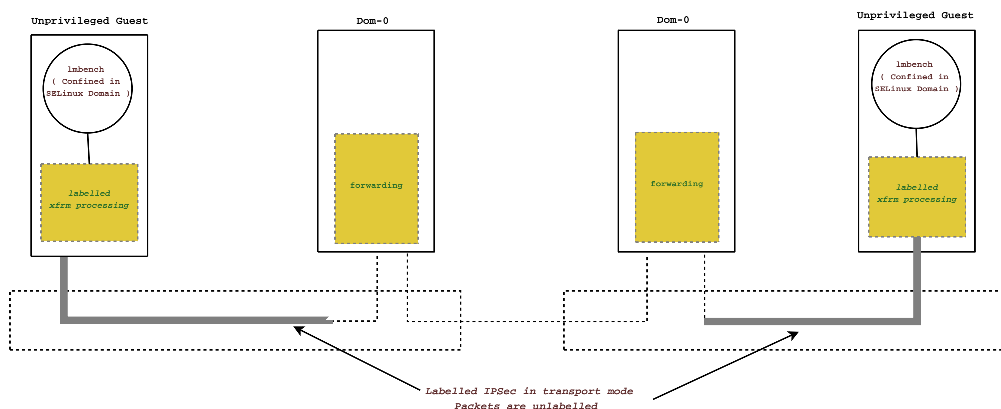

Figure 5.24. End-to-end remote communication, Labeled IPSec

the overhead is minimal, for the additional functionality.

2. Labeled IPSec conveys authorization data, but it displays very poor performance characteristics, when compared to Netlabel. Almost always, IPSec processing involves significant effort spent in policy matching, SA matching and generation, and cryptographic processing on the packets. The performance penalty comes mainly from IPSec processing; labeled IPSec, by itself, does not add much more penalties than IPSec. Labeled IPSec is best used in a scenario where cryptography is the only form of protection that can be relied upon to protect authorization data in packets. In the presence of other mechanisms that protect packet data, Labeled IPSec usage should be eschewed in favor of Netlabel processing.

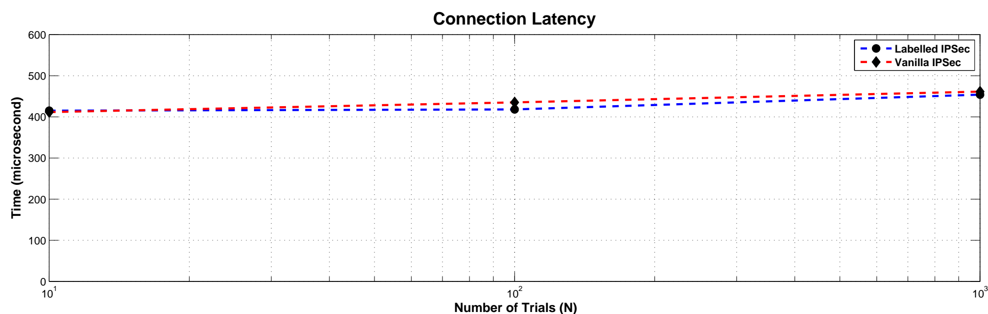

Figure 5.25. Connection Latency for end-to-end IPSec Connection, local communication

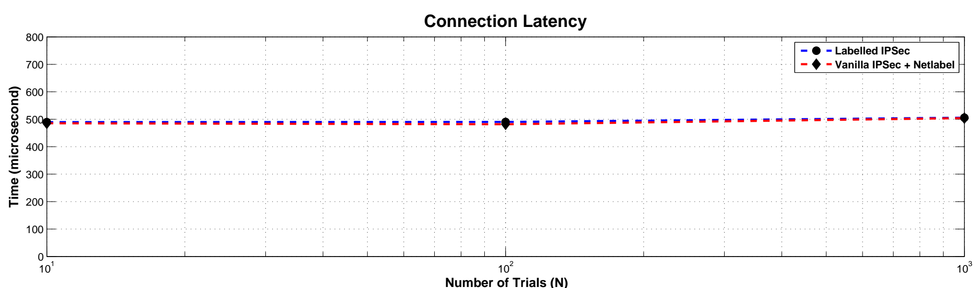

Figure 5.26. Connection latency for end-to-end IPSec Connection, remote communication

We have built a prototype that provides mechanisms which enable the packet labeling state of unprivileged guests to be monitored, and changed dynamically to reflect changes in authorization policy. Further, we leverage existing Xen protection mechanisms, and involve the TCB comprised of guest kernels, the hypervisor and the Dom-0, along with its control tools to lighten the performance penalties associated with conveying authorization data. We have also presented a quantitative picture of the key performance measures. We capture key performance measures in several communication scenarios, and show that our proposed method of transmitting authorization data has clear performance benefits compared to labeled IPSec in all scenarios. But, we realize that our effort is merely a start; further work is needed to make efficient Network Access Control a reality in a virtualized environment. We outline key areas of future work that are required in this direction. We hope that our work will provide a good starting point in the area of Network Access Control in distributed environment.

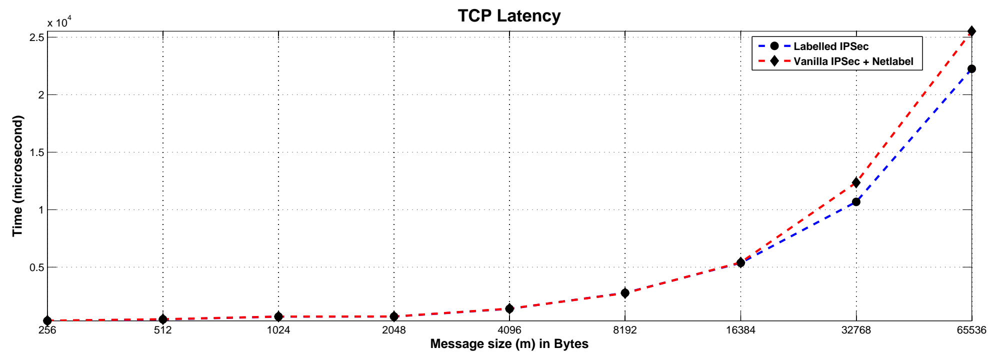

Figure 5.27. TCP Latency for end-to-end IPSec Connection, local communication

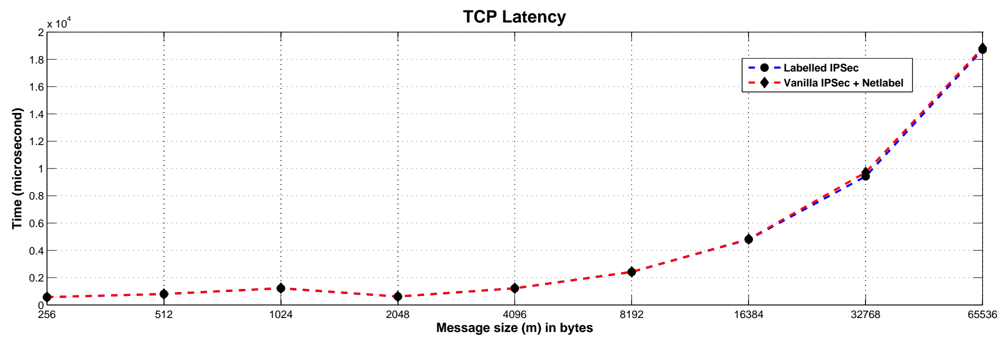

Figure 5.28. TCP latency for end-to-end IPSec Connection, remote communication

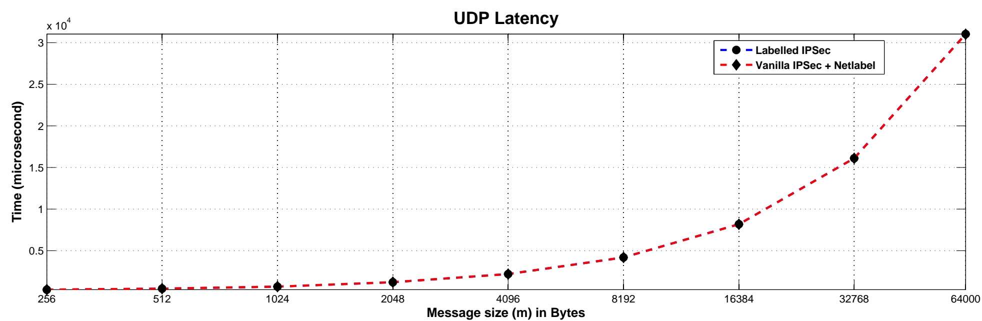

Figure 5.29. UDP Latency for end-to-end IPSec Connection, local communication

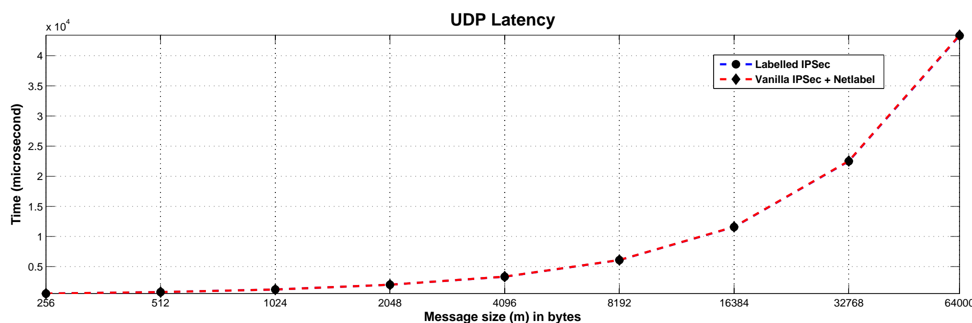

Figure 5.30. UDP latency for end-to-end IPSec Connection, remote communication

1. **Performance Measurements:** We have captured the effect of labeling on key communication metrics: bandwidth and latency, for various types of transport-level services provided by the protocol stack in the kernel. But, our performance measurements have several shortcomings. First, they simulate labeled communication in somewhat restricted scenarios – e.g., single application on each machine, machines that are adjacent to each other in the network. Second, we *do not* capture a key metric, CPU load on each machine. This metric is very important, because it throws more light on the impact of labeled communication on performance. We believe that CPU load measurements will further bolster the case for using Netlabel mechanisms, as IPSec processing stresses the CPU much more than Netlabel, because of the extensive use of cryptography and integrity protection mechanisms. We have discussed the TCP performance anomaly at larger segment sizes in the previous section. We believe that this anomaly is an avenue for further investigation, as it might uncover inefficiencies in Netlabel processing within the Kernel. Therefore, more detailed performance measurements and analysis are necessary to fully understand the impact of transmitting authorization data using Netlabel.

2. **Support for security-aware applications:** Security-aware applications that further aim to regulate and enforce information flows are key to Network Access Control. The fundamental requirement for such applications would be the ability to extract SELinux contexts associated with packets, that

are used by the Reference Monitoring system in the kernel. By making this context information available to application, finer-grained access control can be extended to the network. At present, the netlabel infrastructure within the kernel uses context information in access decisions, but does not make it available via the SELinux interface to applications. The netlabel LSM infrastructure should be modified to accommodate this feature.

3. **Standardized interface between netlabeld and Xen control tools:** At present, our prototype uses a restricted UNIX socket interface, together with a somewhat arbitrary interface between the Xen control tools and netlabeld. This interface has to be standardized, so that a variety of applications can be used to push policy definitions regarding packet labeling to guests. At present, we only permit the default DOI, i.e., the DOI conveyed in all packets irrespective of the application that generates them to be specified via netlabeld. Further work would involve sophisticated SELinux policy analysis tools that investigate information flows in SELinux binary policies in guests and assign mappings of SELinux contexts to DOIs to guests, via netlabeld.

4. **Support for better userspace tools:** New LSMs, such as SMACK, are being written, and are gaining popularity. Although netlabel includes a userspace toolset that allows SELinux LSM contexts to be mapped to DOIs, only SELinux is supported; this support must be extended to all LSMs that become available. Further, netlabeld should provide a way to map *any* type of LSM context information to DOIs within guests.

5. **Rethink the LSM Model:** Although LSM is a descendant of Flask, which is a service in a microkernel, support for network objects remains poor. In particular, the SELinux API has to be improved so that services that can do translations back and forth between network-level and local security attributes of network objects are easier to write. This will have two advantages: first, applications can utilize this API to improve granularity of access control and the type of policies that can be enforced; second, unmodified Linux installations can function as components of a virtualized, distributed system in a seamless fashion. The first step in this direction would be to extend the

kernel mechanisms underlying the SELinux API. The Security Server on one guest should be able to establish Xen communication channels with that of another guest, permitting exchange of authorization data in a manner transparent to applications. This will go a long way towards encouraging secure applications.

6. **Rethink abstraction levels for Xen Communication:** At present, the communication mechanisms offered by Xen are quite primitive: they are limited to device channels — which are useful only to device drivers — and Xenstore entries, which can convey information in restricted text strings (thus limiting its usage to configuration information). A communication primitive that offers a higher level interface while abstracting out the nitty-gritty of the hypercalls that have to be made is an absolute necessity. If such a primitive, that is integrated with sHype, is introduced along with easy-to-use interfaces within the operating system, many modes of secure communication that leverage high-performance features of Xen can be used for interdomain communication.

Bibliography

[1] J. P. Anderson. Computer security technology planning study. Technical Report ESD–TR–73–51, Hanscom AFB, October 1972.

[2] Roland Awischus. Role based access control with the security administration manager (SAM). In *ACM Workshop on Role-Based Access Control*, pages 61–68, 1997.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[4] Robert P. Goldberg. Architectural principles for virtual computer systems. Technical report, Harvard University, Cambridge MA, Division of Engineering and Applied Physics, 1973.

[5] D. Harkins and D. Carrel. RFC 2409: The Internet Key Exchange (IKE), November 1998. Available from: `ftp://ftp.internic.net/rfc/rfc2409.txt,ftp://ftp.math.utah.edu/pub/rfc/rfc2409.txt`.

[6] Neil Horman. *Understanding and Programming with Netlink Sockets*, December 6, 2004. Available from: `http://people.redhat.com/nhorman/papers/netlink.pdf`.

[7] IETF CIPSO Working Group. *COMMERCIAL IP SECURITY OPTION (CIPSO 2.2)*, 16 July, 1992. Available from: `http://www.mjmwired.net/kernel/Documentation/netlabel/draft-ietf-cipso-%ipsecurity-01.txt`.

[8] T. Jaeger and A. Prakash. Support for file system security requirements of computational E-mail systems. In *2nd ACM Conference on Computer and Communications Security*. ACM Press, 1994.

[9] T. Jaeger and A. Prakash. Implementation of a discretionary access control model for script-based systems. In *8th IEEE Computer Security Foundations Workshop*, 1995.

[10] Trent Jaeger, Kevin Butler, David H. King, Serge Hallyn, Joy Latten, and Xiaolan Zhang. Leveraging IPsec for mandatory access control across systems. In *Proceedings of the Second International Conference on Security and Privacy in Communication Networks*, August 2006.

[11] P. Kaijser, T. Parker, and D. Pinkas. SESAME: The solution to security for open distributed systems. *Computer Communications*, 17(7):501–518, jul 1994.

[12] Mitsuru Kanda, Kazunori Miyazawa, and Hiroshi Esaki. USAGI IPv6 IPsec development for linux. In *SAINT Workshops*, pages 159–164. IEEE Computer Society, 2004. Available from: `http://csdl.computer.org/comp/proceedings/saint-w/2004/2050/00/20500159%abs.htm`.

[13] Ipsec-tools. Available from: `http://ipsec-tools.sourceforge.net/`.

[14] C Kaufman. RFC 4306:Internet Key Exchange (IKEv2) Protocol, 2005. Available from: www.ietf.org/rfc/rfc4306.txt.

[15] S. Kent and R. Atkinson. RFC 2401: Security architecture for the internet protocol, November 1998. Available from: http://www.unix-ag.uni-kl.de/~massar/vpnc/docs/rfc2401.txt.

[16] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, June 25-30, 2001, Boston, Massachusetts, USA*, pages 29–42. USENIX, 2001. Available from: http://www.usenix.org/publications/library/proceedings/usenix01/freenix%01/loscocco.html.

[17] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *21st National Information Systems Security Conference*, pages 303–314, Arlington, VA, oct 1998.

[18] Stuart E. Madnick. Time-sharing systems: Virtual machine concept vs. conventional approach. In *Modern Data 2, 3*, pages 34–36, 1969.

[19] Stuart E. Madnick and John J. Donovan. Application and analysis of the virtual machine approach to information system security and isolation. In *Proceedings of the workshop on virtual computer systems*, pages 210–224, New York, NY, USA, 1973. ACM.

[20] D. McDonald, C. Metz, and B. Phan. RFC 2367: PF_KEY Key Management API, Version 2, July 1998. Available from: http://www.faqs.org/rfcs/rfc2367.html.

[21] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In USENIX, editor, *Proceedings of the USENIX 1996 annual technical conference: January 22–26, 1996, San Diego, California, USA*, pages 279–294. USENIX, 1996.

[22] Paul Moore. *netlabel_tools documentation*. Available from: http://netlabel.svn.sourceforge.net/viewvc/netlabel/netlabel_tools/head%/docs/.

[23] James Morris. *selopt overview*. Available from: http://www.intercode.com.au/jmorris/selopt/old/selopt-overview.txt.

[24] Bryan D. Payne, Martim Carbone, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, December 2007.

[25] Bryan D. Payne, Reiner Sailer, Ramón Cáceres, Ron Perez, and Wenke Lee. A layered approach to simplified access control in virtualized systems. *SIGOPS Oper. Syst. Rev.*, 41(4):12–19, 2007.

[26] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramon Caceres, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn. Building a MAC-based security architecture for the xen open-source hypervisor. In *Proceedings of the 2005 Annual Computer Security Applications Conference*, pages 276–285, December 2005.

[27] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. RFC 3549: Linux netlink as an IP services protocol, July 2003. Available from: http://www.faqs.org/rfcs/rfc3549.html.

[28] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The flask security architecture: system support for diverse security policies. In *SSYM'99: Proceedings of the 8th conference on USENIX Security Symposium*, pages 11–11, Berkeley, CA, USA, 1999. USENIX Association.

[29] Walter Tuvell. Response to “problems with dce security services”. *SIGCOMM Comput. Commun. Rev.*, 26(2):64–73, 1996.

[30] Gregory White and Udo Pooch. Problems with dce security services. *SIGCOMM Comput. Commun. Rev.*, 25(5):5–12, 1995.

[31] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In *Proceedings of the 11th USENIX Security Symposium*. USENIX, aug 2002. Available from: `http://online.securityfocus.com/data/library/lsm-usenix.pdf`.