METHODS FOR SPECIFYING AND RESOLVING
SECURITY POLICY COMPLIANCE PROBLEMS

A Dissertation in
Computer Science and Engineering
by
Sandra Julieta Rueda Rodríguez

© 2011 Sandra Julieta Rueda Rodríguez

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2011
The dissertation of Sandra Julieta Rueda Rodríguez was reviewed and approved* by the following:

Trent Jaeger  
Associate Professor of Computer Science and Engineering  
Dissertation Advisor, Chair of Committee

Patrick McDaniel  
Associate Professor of Computer Science and Engineering

Adam Smith  
Associate Professor of Computer Science and Engineering

Bhuvan Urgaonkar  
Assistant Professor of Computer Science and Engineering

Stephen G. Simpson  
Professor of Mathematics

Mahmut Kandemir  
Director of Graduate Affairs and Professor of Computer Science and Engineering

*Signatures are on file in the Graduate School.
Abstract

Distributed systems have become sufficiently complex that it is impractical for administrators to configure them manually to prevent security vulnerabilities. These systems consist of multiple interconnected hosts that possibly run virtualized environments and support one or more distributed applications. The administrator’s job is to identify security-sensitive data and configure system components (i.e., programs, operating systems, and virtualization environments) to meet a security goal (i.e., protect data from unauthorized modification or leakage).

To prevent vulnerabilities, mandatory access controls (MAC) have been integrated into applications, operating systems, and virtualized environments. MAC systems guarantee that a system behaves within the boundaries defined by an access control policy. The problem is that although MAC systems are developed to prevent vulnerabilities, configuring several of them to work as a whole is a challenging task for system administrators. Each individual MAC policy is complex, the policies are independently developed, and the security goals that prevent vulnerabilities are usually not explicit.

We develop mostly-automated services to help administrators configure and deploy distributed MAC systems to prevent security vulnerabilities. Our results show that for commonly used deployments it is possible to use available information with little input from administrators to automate tasks that are manual currently. We reduce the burden of configuration on system administrators, thus making the deployment of MAC in distributed systems more practical.
# Table of Contents

List of Figures viii

List of Tables x

Acknowledgments xi

Chapter 1

Introduction 1

1.1 Research Objectives .............................................. 2

1.2 Contributions of this Dissertation .............................. 5

Chapter 2

Background and Related Work 7

2.1 Multilayered Policies .............................................. 7

2.1.1 Virtual Machine Monitor Layer ............................... 8

2.1.2 Operating System Layer .................................... 9

2.1.2.1 SELinux .................................................. 10

2.1.2.2 AppArmor ............................................... 11

2.1.3 Network Layer .................................................. 12

2.1.4 Application Layer ............................................ 13

2.1.4.1 Jif: Java + Information Flow ........................... 13

2.1.5 User space security servers in SELinux ..................... 15

2.1.6 Comprehensive Enforcement of Information Flow Policies ... 16

2.2 Compliance .......................................................... 16

2.2.1 Policy Model ................................................... 16

2.2.2 Problem Definition ........................................... 17

2.2.3 Information Flow Security Models ........................... 19

2.2.3.1 Lattices .................................................. 19

2.2.3.2 Confidentiality Models ................................... 19
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.1</td>
<td>Tamperproof Compliance</td>
<td>67</td>
</tr>
<tr>
<td>4.3.1.1</td>
<td>Build the Tamperproof Goal Policy</td>
<td>67</td>
</tr>
<tr>
<td>4.3.1.2</td>
<td>Build the System Policy</td>
<td>68</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Evaluating logrotate</td>
<td>70</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Evaluating other Trusted Programs</td>
<td>71</td>
</tr>
<tr>
<td>4.4</td>
<td>Discussion</td>
<td>73</td>
</tr>
</tbody>
</table>

**Chapter 5**
Compliant Configuration for Virtual Machines

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>76</td>
</tr>
<tr>
<td>5.2</td>
<td>Problem Definition</td>
<td>77</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Web Applications</td>
<td>77</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Layered Architectures</td>
<td>79</td>
</tr>
<tr>
<td>5.3</td>
<td>End-to-End Enforcement</td>
<td>80</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Multi-Layer Reference Monitor</td>
<td>80</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Mandatory Protection Systems</td>
<td>82</td>
</tr>
<tr>
<td>5.4</td>
<td>Architecture</td>
<td>84</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Components</td>
<td>84</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Protocols</td>
<td>85</td>
</tr>
<tr>
<td>5.4.2.1</td>
<td>Initial Policy Configuration</td>
<td>85</td>
</tr>
<tr>
<td>5.4.2.2</td>
<td>Processing a URL</td>
<td>86</td>
</tr>
<tr>
<td>5.4.2.3</td>
<td>Browser Transitions</td>
<td>87</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Enforcement</td>
<td>88</td>
</tr>
<tr>
<td>5.5</td>
<td>Policy Compliance</td>
<td>89</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Application Compliance</td>
<td>90</td>
</tr>
<tr>
<td>5.5.2</td>
<td>VM Compliance</td>
<td>90</td>
</tr>
<tr>
<td>5.6</td>
<td>Implementation</td>
<td>91</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Implementing an Application Policy</td>
<td>92</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Improving Performance</td>
<td>94</td>
</tr>
<tr>
<td>5.7</td>
<td>Evaluation</td>
<td>94</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Security Evaluation</td>
<td>94</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Performance</td>
<td>95</td>
</tr>
<tr>
<td>5.8</td>
<td>Related Work</td>
<td>97</td>
</tr>
<tr>
<td>5.9</td>
<td>Discussion</td>
<td>98</td>
</tr>
</tbody>
</table>

**Chapter 6**
Distributed Compliant Systems

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>99</td>
</tr>
<tr>
<td>6.2</td>
<td>Problem Definition</td>
<td>102</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Example System</td>
<td>102</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Policy Generation Problem</td>
<td>104</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Related Work</td>
<td>106</td>
</tr>
<tr>
<td>6.3</td>
<td>Solution Methodology</td>
<td>108</td>
</tr>
</tbody>
</table>
6.4  Solution Background ........................................ 109
  6.4.1  Hierarchical Graphs ..................................... 109
  6.4.2  Graph-Cut Model .......................................... 111
  6.4.3  DIFC-Flume Information Flow Policy .................. 112

6.5  Solution Design .................................................. 113
  6.5.1  Specification .............................................. 113
  6.5.2  Building HSM Data Flow Graphs ......................... 114
  6.5.3  Constructing Graph-Cut Models ......................... 117
  6.5.4  Computing Mediation Placements ....................... 121
  6.5.5  Generating Practical Integrity Policy ................. 124
    6.5.5.1  Policy Generation ................................. 124
    6.5.5.2  HSM and DIFC-Flume Equivalence ................ 126
  6.5.6  Implementation ........................................... 129

6.6  Evaluation ........................................................ 131
  6.6.1  Prototype System ......................................... 131
  6.6.2  Experimental Setup ...................................... 132
  6.6.3  Experiments and Results ................................ 134

6.7  Conclusion ....................................................... 139

Chapter 7
Conclusions and Future Work 141
  7.1  Conclusions .................................................. 141
  7.2  Future Work .................................................. 143

Bibliography 145
List of Figures

1.1 The Compliance Problem ......................................................... 3
2.1 Flask Architecture ................................................................. 10
2.2 Pseudo-code for a Trusted Program that Leaks Information ........... 15
2.3 Pseudo-code for a Trusted Program that Releases Information under Well Defined Conditions (Declassification) ........................................ 15
2.4 Information Flow Policy Modeled as a Directed Graph .................. 17
2.5 Example of a Mapping Function for Information Flow Policy Graphs .... 18
2.6 Instance of the Policy Compliance Problem ................................. 23
3.1 SELinux-Jif (OS/Application) Compliance Problem ..................... 53
3.2 Service for Inspection and Execution of Security Typed Applications (SIESTA) architecture .................................................. 55
4.1 Overview of the Compliance Evaluation Process for Trusted Packages before Deployment ................................................................. 61
4.2 Policy Compliance Problems for Trusted Applications .................. 62
4.3 Scheme of the mapping mechanism for the PIDSI approach ............ 64
4.4 Instance of the Compliance problem for logrotate ....................... 65
4.5 Tamperproof Goals for logrotate ................................................ 69
4.6 System Information Flow Graph including logrotate program’s policy module ................................................................. 70
5.1 Architecture of a Canonical Distributed Application ..................... 78
5.2 Example of a Mandatory Protection System ................................. 82
5.3 Example of a Multi-layer MAC Architecture ............................... 84
5.4 Labeling State for the multi-layer VM Architecture ..................... 88
6.1 End-to-end web application system ............................................. 103
6.2 Policy Generation Problem ....................................................... 105
6.3 Proactive Integrity Methodology Tasks ......................................... 108
6.4 Systems are encapsulated and hierarchical ................................... 109
6.5 Outputs: goal policy $\mathcal{L}$, mapping function $\text{map}$, and an Information Flow (IF) Policy. The policy is based on the input components $\text{Component}_1,..,\text{Component}_i$. The IF policy assigns an integrity level and mediation capabilities to every label defined by the initial policy. The level is the integrity a label (i.e., the program represented by that label) can provide, and the mediation capabilities identify lower integrity inputs that the label can handle and sanitize.

6.6 Options to represent information flows between nodes in the HSM model .......................... 114
6.7 Mediation Resolution Algorithm ................................................................. 123
6.8 DIFC-Flume policy generation ................................................................. 126
6.9 HSM to DIFC-Flume and DIFC-Flume to HSM Mapping Functions ........ 128
6.10 HSM model representation ................................................................. 129
6.11 Representation of Interfaces ................................................................. 132
6.12 Representation of Mediators ................................................................. 134
List of Tables

4.1 Compliance Test Case and Results .................................................. 70
4.2 Results of applying the PDISI approach to SELinux Trusted Packages . 72
4.3 System labels used by SELinux trusted packages .............................. 73
4.4 Compliance exceptions and resolutions for the SELinux reference policy 74

5.1 Elements of the mandatory protection system needed at multiple layers for end-to-end secure distributed systems. ................................. 81
5.2 Excerpts of the mandatory policies for a sample application ............ 93
5.3 Browser configuration and (first) page loading times for a new VM, for a pre-loaded VM, and Firefox. ......................................................... 96

6.1 SELinux and iptables Secmark policy excerpts. The SELinux policy specifies the subjects that have access to network communication channels and to what channels, while the network policy specifies the authorized communication channels. ........................................ 116
6.2 Related Network Policies. The first rule specifies the labeling of packets sent by the web server to the database server, while the second rule specifies the labeling of packets as received by the database server from the web server. ................................................................. 130
6.3 Problem Size: number of policy rules, subject labels (S), object labels and interfaces (O+I), and edges (E) for static and runtime cases. ...... 134
6.4 Mediation costs (edges and subjects) by cut problem for the Web Server Case. ................................................................. 135
6.5 Mediation costs (edges and subjects) by cut problem adding a Trusted DNS Case. ................................................................. 135
6.6 Mediation costs (edges and subjects) by cut problem identifying web resources. ................................................................. 135
6.7 DIFC-Flume Dual Capabilities. Subjects are grouped per number of capabilities. ................................................................. 136
6.8 Execution Times, all in seconds. HSM: Construction of the HSM model, GCM: Constructions of the Graph cut model, Cuts: running the MEDIATION RESOLUTION program, DIFC: Computing DIFC-Flume labels and capabilities. ................................................................. 138
Acknowledgments

I want to express my gratitude to all the people that have helped during my years in graduate school. Their support, encouragement, comments, and constructive criticism are invaluable.

To Dr. Trent Jaeger, my academic advisor, for being my mentor in this adventure. His knowledge, technical advice, and feedback in general, guided me all the time. His support, constructive criticism, and faith in my work, encourage me to keep developing the research ideas I had. I also want to thank his wife, Dana, for her sweetness and always warming welcome to their house.

To Dr. Patrick McDaniel whose security class got me interested in the topic of systems security, if not for the first time, in a definitive manner. He also introduced me to Dr. Jaeger, who later became my advisor, for that I am grateful. Dr. McDaniel’s knowledge in the area, feedback on my work, and constant encouragement helped me at every step of my way.

To the members of my committee, Dr. Adam Smith, Dr. Bhuvan Urgaonkar, and Dr. Stephen Simpson, for their technical feedback, and constructive criticism. Their comments improved the content of my dissertation. They also made me realize how important it is to be able to present my work to people from other disciplines and gave me ideas to advance in that direction.

To Dr. Padma Raghavan, my first advisor at Penn State. She listened to my concerns, answered all my questions, and guided me during my first year in graduate school. That year was difficult, the adaptation process was not as easy as I expected, and the stress of being in graduate school was getting to very high levels. Her advice and help were invaluable.

To all the members of the systems and infrastructure security lab, SIIS lab, at Penn State. Their comments, suggestions, and criticism about research ideas were always helpful. They were always willing to lend a hand to solve technical problems that emerged in my research projects. Even better, they always had smart and funny comments that helped me survive everyday life in the lab.

To all my friends. They have made this a nice journey. My friends in Colombia supported me in many ways. They listened to my concerns, kept me informed of the things that were happening over there, and were always willing to help with anything I
needed back home. My friends in State College were always willing to help with daily tasks and share time with me. They even shared their homes when we did not have a place to stay. Additionally, I am fortunate enough to have friends in the SIIS lab. They helped with research discussions and feedback all the time, even late nights and Sunday evenings. It has been an honor, I thank you all for your constant cheering and support, and I hope to see you again.

To my mother, my sister, and my husband. They have been my unconditional partners in this adventure. They have supported me, encouraged me, and reprimanded me when necessary. I don’t know what the future stores for us, but what I do know is that successfully finishing graduate school is the result of our teamwork.
Dedication

To my mother, my sister, and my husband.
Chapter 1

Introduction

Distributed systems have become sufficiently complex that is essentially impractical for administrators to configure them manually to prevent security vulnerabilities. Distributed systems consist of multiple interconnected hosts, each running several programs hosted by an operating system, including several system services, and increasingly a virtualized environment. The administrator’s job is to identify security-sensitive data and configure system components (i.e., programs, operating systems, and virtualization environments) to protect such data from attacks, such as unauthorized modification or leakage. However, each of these components is developed independently, often by different parties, so the administrators must determine how to prevent attacks when composing these component into a system. But while administrators may know where attacks originate, it is difficult to determine how attacks may propagate among all the components in distributed systems.

A web application illustrates the complexity of a distributed system and the variety of security problems that may arise. In a web application, web servers act as gateways for external clients to make requests to web application processes, which often communicate with backend database servers to store and retrieve data. However, adversaries may send malicious requests to a web server that propagates them to the web applications, database, potentially the operating systems and virtualization environment. For example, in the case of SQL injection [118, 143], an attacker uses a web server as a gateway that forwards malicious SQL queries to a database server. An attacker may also use vulnerabilities in a web server [9] to execute remote code provided by the attacker to perform a local exploit to compromise a root process, thus compromising the whole operating system and providing a means to attack the virtualization environment.
To prevent vulnerabilities in distributed systems, mandatory access controls (MAC) have been integrated into operating systems. A MAC operating system specifies a fixed set of labels and assigns them to every subject and object in the system. It also enforces an access control policy defined by security administrators that only they can change. The policy specifies, in terms of the labels, the operations that subjects can perform on objects. Thus, operating systems that implement MAC can enforce a distinct set of authorized operations for each subject that confine processes in a manner that cannot be modified. As an example, to avoid some exploits in the case of the web server, an administrator may assign a particular security label to processes spawned by the web server and restrict them to run only trusted scripts.

However, MAC operating systems alone are not enough to prevent security vulnerabilities, so MAC enforcement mechanisms have also been added to other software layers. Often programs are designed to provide varied functionality and to serve users whose data have different security requirements. They require privileged permissions to meet their functional requirements, but as a result the operating system cannot control what the application does anymore. For example, the X server, a Linux server that allows users to run a graphical interface, needs access to resources from multiple users with various security requirements to display and update data as expected. The OS cannot control how the X server allows other processes to interact. As a result, developers added MAC policy enforcement to X Windows to control user interactions [153]. Additionally, virtual machine technology is now being used to enforce MAC policies [26, 132], because commodity operating systems fail to satisfy reference monitor guarantees [7]. With this technology an additional software layer, a virtual machine monitor (VMM), enforces isolation among virtual machines. VMM MAC enforcement controls each VM access to resources being virtualized to manage interactions among VMs.

With MAC enforcement available at all these layers, deploying distributed systems to prevent attacks should be easier. However, because of the complexity of distributed systems it is difficult for administrators to determine whether attacks are possible, and what to do to stop them.

1.1 Research Objectives

Determining whether a distributed system is prone to attack has been modeled as a compliance problem [90], shown in Figure 1.1. Informally, a compliance problem compares two policies, a test policy that represents the behavior of a system and a goal
Figure 1.1: The Compliance Problem. It receives a system policy that describes the behavior of a given system, a goal policy that describes the acceptable behavior, and determines whether the system policy meets the conditions defined by the goal policy.

A policy that represents the behavior that prevents security vulnerabilities, to determine whether the test policy meets the requirements embodied by the goal policy. A compliant system is guaranteed to behave within the boundaries of the acceptable behavior.

Compliance has been used to evaluate network policies [126, 87, 158], trust management policies [17, 79, 81], and individual access control policies [67, 141, 43], and multiple access control policies, requiring significant manual specification for integrating policies [39, 120, 74]. Our aim is to assist administrators by automating tasks in both building compliance problems from multiple, independently-developed mandatory access control (MAC) policies and evaluating compliance problems.

While researchers have developed methods to generate access control policies from runtime analysis results [123, 135, 109] and information flow models of policies [146], the research goal of repairing policies to meet security goals (i.e., goal policies) is just emerging. Narain et al. have constructed a SMT-based approach [30, 66] for generating network policies from functional and security constraints. For distributed systems’ access control policies, the problem is more complex, as discussed above, because of the variety of software and layered architecture of modern distributed systems. Researchers have proposed using the cuts of the information flow graph for resolution of compliance violations [120, 72], but this proposal has not been applied in practice. The problem has been that neither functional nor security constraints are specified explicitly, and distributed systems have too many components to expect system administrators to produce such a specification manually. By producing compliance problems, we generate security constraints in the form of a goal policy, but we need to assist administrators to assess the impact of different functional requirements. Our aim is to provide methods that produce functional requirements, when possible, or assist administrators in different functional requirements by producing resolutions to compliance errors in independent MAC policies.
in distributed systems.

The thesis of this dissertation is therefore:

*We can construct services that build, evaluate, and resolve compliance problems mostly-automatically for multiple, independent MAC policies in distributed systems.*

In this thesis, we must address several challenges to be able to evaluate compliance for a distributed system.

- It is difficult to construct a test policy for a distributed system automatically. The test policy for a distributed system must express the operations authorized for all the software components in the system (i.e., programs, operating systems, and virtual machine monitors). Also, it must identify the authorized communications among the components, particularly in different software layers. For example, how can a process in one VM communicate through the virtualization environment to access another process on a remote host or other VM? In this thesis, we explore methods to integrate policies written for different components into a single, system-wide data flow graph to express test policies.

- Goal policies are not explicit. While manual specification of goal policies may be reasonable to evaluate single test policies, it is not practical for distributed systems as the size of the goals is proportional to the number of components. However, in some cases, security constraints may be inferred from the context or inferred from a small amount of additional information. In this thesis, we explore methods to generate and use security constraints in goal policies from such information, such as whether a process is expected to be trusted by the component.

- Often access control policies fail to satisfy classic secrecy and integrity goal, such as Biba integrity [15] or Bell-LaPadula secrecy [12], because these goals prohibit two-way interaction with processes of lower integrity or secrecy, respectively. For example, they do not allow trusted programs to read any untrusted input, but often trusted programs are expected to read and process untrusted inputs, as happens with network facing daemons. However, such restrictive goals represent the state-of-the-art for ensuring the prevention of attacks. Thus, administrators need assistance to produce resolutions that approximate such goals, while preserving function. Since functional constraints are not explicitly specified, in this thesis, we also explore methods to infer expected function, where possible, and assist administrators in evaluating different functional specifications.
1.2 Contributions of this Dissertation

To evaluate our thesis we developed and implemented compliance services. Our contributions are:

- We developed a service that evaluates compliance of a program policy and an operating system (OS) policy that are independently developed. The OS policy becomes the goal policy. The service verifies that all flows authorized by the program policy are also authorized by the OS policy [47, 52]. We further explain this service in Chapter 3.

- We developed heuristics to automate the construction of a mapping function to evaluate compliance for trusted programs, enabling automated compliance evaluation [127]. We define a Biba integrity lattice as the goal policy, and construct the test policy by composing the OS policy and the program policy. This approach demonstrates that it is possible to develop heuristics that take information readily available to automate tasks that were previously manual in the compliance evaluation process. We describe our approach in Chapter 4.

- We developed FlowwolF, an architecture that dynamically configures compliant virtual machine systems [129, 49]. It does so by retrieving, generating, and deploying VMM, VM, and network MAC policies to the corresponding MAC enforcers, for web applications. We describe FlowwolF in more detail in Chapter 5.

- We developed a compliance service to construct and resolve compliance problems for distributed systems automatically. The service has heuristics to infer security goals and mapping functions, based on information that is readily available to system administrators in VM systems [128]. To model distributed system data flows, the service leverages the hierarchical graph model developed by Alur et al. [4] for model checking. Such a hierarchical graph enables us to represent flows between different components and exposes the inherent layering [130, 128]. The service resolves errors using a graph-cut approach to compute near-minimal solutions for non-compliant systems [128]. We describe the methods implemented by this service in Chapter 6.

In summary, the purpose of this work is to reduce the complexity of configuration and deployment of MAC systems. To do so, we developed services to build, evaluate, and resolve compliance problems. While developing these services, we addressed several
challenges that have not been addressed before; in particular, constructing compliance problems and finding resolutions automatically. The compliance services we developed implement functions that were not available before but are needed to build and deploy distributed systems that meet security requirements.

Although our compliance services are a first step towards reducing the burden on administrators to develop compliant composite MAC systems, there are still several aspects to consider in order to make MAC systems more practical.

- We did not explore how to use our services to tighten an original set of MAC policies while keeping functionality. Currently functional requirements are not explicit, we want to explore the feasibility of generating them.

- Our approach builds an analytical model from a set of MAC policies previously defined. We want to explore the feasibility of using the model to define security requirements for a new component and generate the corresponding policy while keeping the entire distributed system compliant.

Considering the advantages of MAC systems in enforcing security requirements, our goal is to develop additional approaches and mechanisms that can help in the configuration of components with MAC enforcement, while keeping manual specification to a minimum.
Chapter 2

Background and Related Work

In Chapter 1 we informally introduced the policy compliance problem: it takes two poli-
cies, a test policy that describes the behavior of a system and a goal policy that describes
the acceptable behavior, and evaluates whether the test policy meets the requirements
embodied by the goal. In this chapter we first describe the components of a system with
multiple layers of enforcement. In this kind of environments, in which there are multiple
interconnected hosts, where each host possibly runs a multilayer stack, we want to evalu-
ate compliance of multiple policies (Section 2.1). The second part of the chapter provides
formal background about the compliance problem including evaluation of compliance for
multiple policies (Section 2.2). The compliance services we developed, which we describe
in the remaining chapters of this document, are based on the model of compliance this
chapter presents. The third part describes the evolution of policy enforcement mech-
anisms (Section 2.3) since they are key to our approach. Finally, we present previous
research about compliance evaluation (Section 2.4). The main limitations of previous
approaches are that they leave administrators with the tasks of manually specifying
security goals, and they do not consider the layering in modern systems.

2.1 Multilayered Policies

Modern systems may include MAC enforcement at several software layers: virtual ma-
chine monitor (VMM), virtual machine operating system (VM), and application. In ad-
dition, network policies can also further restrict communications between components.
MAC enforcement at the VMM layer governs communications across VMs (via network
channels or VMM resources), the VM layer governs communication among applications
within a VM, and finally, the application layer governs access over application resources. This section presents technical details about available systems that enable administrators to define and enforce MAC policies at various software layers.

2.1.1 Virtual Machine Monitor Layer

Virtual machines systems are promoted as a basis for building secure systems because they provide isolation, however, in some cases some sharing between VMs is required. For example, virtual machines that support distributed applications need to communicate with untrusted subjects. As a case in point, NetTop [94] and Solaris Containers [77] enforce MLS policies on virtual machines, thus VMs can only communicate with VMs that have the same MLS security level. However, for some applications MLS policies are not flexible enough. For example, a web server may require access to several databases, where each one of them stores data with a particular security level.

Reference monitors at the VMM layer provide a mechanism to enforce mandatory access control policies (MAC) on operations that virtual machines (VMs) execute on VMM resources. These policies control whether one VM can access another VM’s resources or establish communication channels with other VMs. As an example, the Xen Security Modules (XSM) implementation provides a platform to enforce MAC policies at the VMM layer. XSM implements mandatory access controls (MAC) for the Xen hypervisor (VMM). If an operating system is paravirtualized, it traps sensitive operations into the hypervisor, so the hypervisor can enforce controls on these sensitive operations [121]. Otherwise, privileged and sensitive calls are set to trap to the hypervisor directly, removing the need for paravirtualization. The XSM design is derived from Linux Security Modules (LSM). The LSM approach [140] inserts hooks into the Linux kernel, to mediate security-relevant operations, such as access to files. Similarly, XSM provides a set of hooks in the Xen kernel, to mediate access to resources that are security-relevant for Xen, such as event channels (used to send and receive signals) and grant tables (used to share memory). XSM hooks intercept access requests over these resources and query a linked XSM security module to decide whether to allow or stop the request. Current security modules that can be linked at boot time with an XSM system are: Dummy (XSM default), ACM/sHype (IBM) [132], and Flask (NSA) [26]. An XSM/Flask module provides Xen the same kind of functionality that SELinux gives to Linux. SHype [132] enforces a policy that authorizes communication between VMs only if they have the same security label. XSM/Flask policies assign security contexts to VMs and VMM resources and authorizes operations, including resource sharing, between VMs only if the policy
authorizes it. XSM/Flask policies are more flexible than NetTop [94] and SHype [132].

2.1.2 Operating System Layer

Applications usually have some kind of security mechanism to protect their internal data, such as user login and data encryption. Protecting the application itself, i.e., executable and configuration files, requires having mechanisms to handle authorization and access control at the operating system on which the applications run. Since operating systems that implement discretionary access controls (DAC) are not enough, we require operating systems that implement mandatory access controls (MAC) [83]. DAC operating systems are very limited, the enforcing mechanism may be compromised, users may be involved in the assignment of security attributes, and users may be subverted and cannot be contained. Early attempts to realize MAC include Multics [28] and security kernels [96, 92, 19, 95, 154, 6].

MAC operating systems restrict policy specification to only policy administrators [83]. Linux uses the Linux Security Modules (LSM) framework to implement mandatory access controls. The LSM framework defines a reference monitor interface that enables administrators to link the kernel with different reference monitor implementations. The reference monitor concept [7] defines the requirements for access control in a secure operating system [57]: complete mediation, tamper protection, and verifiability. Complete mediation ensures that all security relevant operations are mediated, tamper protection ensures that the reference monitor implementation cannot be modified by untrusted subjects, and verifiability ensures the correctness of the implementation. Since the LSM framework enables the integration of the Linux mainstream with different implementation of the reference monitor, administrators may choose among several implementations the one that best fit their particular requirements [155, 140].

SELinux [111] and AppArmor [109] are well-known security servers currently available that implement LSM for the Linux kernel. However, these two implementations have different approaches to make security decisions. While SELinux uses security contexts (labels) to make security decisions, AppArmor confines applications based on the paths of the application itself and the resources it tries to access. Security analysts have long argued about the advantages and disadvantages of these two approaches. While access control rules based on paths may be easier to define, they are also easier to bypass, by having soft links for instance. In this document we do not argue in favor of any of these implementations but focus on their features, and how we can integrate them into our model.
Figure 2.1: Flask Architecture. Subjects and objects are labeled with contexts that represent security attributes, and the security server makes access control decisions based on such contexts.

### 2.1.2.1 SELinux

Security Enhanced Linux, SELinux [111], is a secure operating system developed by the NSA based on the Flask architecture [142], which itself was developed based on DTOS [96]. DTOS and Flask implement the reference monitor concept [7], and rely on a security server to make access control decisions. Figure 2.1 shows the architecture that SELinux implements. Subjects and objects are labeled with contexts that represent their security attributes and the security server makes access control decisions based on those contexts.

An SELinux security context includes three basic elements: user, role, and type. An SELinux user does not necessarily correspond to a Linux user, it groups users with defined security requirements, such as system administrators (sysadmin), or security administrators (secadmin). A role limits the domains to which a user has access, and the type is a label associated to a domain. For objects, the user is the owner of the object, the role is object_r, a default value, and the type is the label associated to the object. The security contexts below are SELinux contexts. The first context corresponds to processes started by users with no special security attributes, and the second corresponds to objects these users own [139].

```
user_u:user_r:app_t
user_u:object_r:user_home_t
```

The SELinux access control policy defines the roles any SELinux user may assume, the domains a role can access, the types any process can access and how. When a process tries to gain access to a particular object in a given SELinux system, a security decision...
has to be made; the access is allowed or denied depending on the security context of the subject, the security context of the object, and the policy. The following example shows the kind of rules that the SELinux policy language permits:

allow logrotate_t logrotate_tmp_t:file {create read write append};
allow logrotate_t tmp_t:dir {read search};

The first rule allows subjects with label logrotate_t to access objects with label logrotate_tmp_t, and class file, in the indicated modes (create, read, etc.). The second rule allows subjects (processes) running with type logrotate_t to access directories of type tmp_t in the indicated modes.

Currently Tresys provides an standardized version of an SELinux policy, the SELinux reference policy [149]. The reference policy has modules to confine around 300 Linux applications, such as alsa, amanda, anaconda, bootloader, kudzu, and dpkg. The policy also enables some applications to run unconfined. Administrators may use the policy as provided, enable only some of the modules, and add their own modules. Therefore, the set of applications that are confined varies across installations.

2.1.2.2 AppArmor

AppArmor uses profiles, modules that group rules associated to a target application, to confine applications. It only confines applications for which there is a defined profile, other applications run unconfined, thus limited only by DAC permissions.

For an application to have access to a particular file, the profile must explicitly specify the file and the permission, (r for read, and w for write). If a pair file-permission is not specified, then the access is denied. AppArmor can also meditate use of Linux capabilities and requests for network resources. The following example shows the rules that the AppArmor policy language supports. This rule authorizes /usr/bin/foo to read all the libraries in /lib with names as indicated by the pattern.

/usr/bin/foo {
    /lib/lib*.so* r,
}

Since administrators may decide to use only the profiles installed by default, add profiles provided by other developers, or develop their own profiles, the set of applications that are confined varies across installations. The set of applications that are currently installed by default on a standard Ubuntu installation, via the package apparmor-profiles, has: ping, gdm guest-session, dhclient, klogd, syslogd, syslog-ng, evince, firefox,
dovecot, avahi, cups, dnsmasq, identd, mdsnd, nmbd, nscd, smbd, tcpdump, and traceroute.

2.1.3 Network Layer

Labeled networking is possible based on the IP security options (IPSO) of the IP header and protocol [69]. The initial approach was revised and the new approach is known as the commercial IP security option (CIPSO) [57]. Any architecture that aims to enforce end-to-end information flow policies must span multiple machines. For example, a database server and other servers would need support to establish the identity and the security context of its clients in order to make access control decisions, i.e., to enforce an access control policy at the application layer. In general, the goal is to control interactions between applications running in different machines, based on security labels. Current mechanisms that support labeled network communications for Linux include Labeled IPsec [60], NetLabel [110], and iptables with the secmark extension [97].

IPsec is a real time protocol used by two parties to establish a secured communication channel. The protocol requires at least authentication of the involved parties and establishing a session key for cryptographic protection of the conversation. A protected session is known as a security association. An IPsec security association may provide different cryptographic-based services: integrity, encryption, or both [68]. Labeled IPsec provides a mechanism based on IPsec to implement network mandatory access control. It associates packets with the labels of the security association upon which they are being sent [61]. The OS MAC policy controls the processes that may have access to the packets, according to the rules that govern the labels of the process that tries to read the packet, and the packet.

NetLabel also provides labeling of network packets. It handles two configurations: the CIPSO protocol and a network peer labeling. With the former, all traffic generated by a given application is assigned the same label (called domain of identification, DOI) and all input traffic labeled with a given DOI is mapped to the same local application domain. With the latter, packets generated by an application may be assigned different labels according to the peer on the other end of the communication. The OS MAC policy controls access to the packets, based on the labels of the process that tries to read the packet and the packet itself.

The secmark extension for iptables enables labeling of network packets with security contexts based on the network data iptables already handles to filter network connections,
such as protocol, source and destination port, and source and target IP addresses. The idea is to use iptables with the semark extension to label packets and SELinux [97] or any other LSM implementation to enforce security policy, based on the labels of the process trying to read or write a packet and the packet.

2.1.4 Application Layer

An operating system (OS) controls the set of OS resources an application has access to. If applications are allowed to access elements at only one security level, then an operating system with proper support mechanisms can enforce end-to-end information flow policies. However, some applications require access to multiple security levels in order to work in the expected way, for example, editors and e-mail clients that may serve users with different security requirements. When an application is given access to multiple security levels, an operating system has no way of controlling the internal behavior of such application. As a consequence, this application might leak information or change integrity values associated to a given set of data.

In these cases we need techniques to evaluate whether a program internally enforces a policy that is consistent with the policy of the system. Available techniques include: type qualifiers [106], taint tracking [156, 107, 151], and policy analysis of user-space reference monitors [153] and security-typed languages [100, 103, 138]. We focus our attention on security-typed languages and user-space reference monitors because they use policies to enforce an expected behavior. In doing so they create an additional layer of enforcement.

2.1.4.1 Jif: Java + Information Flow

Security-typed languages (STLs) enable the development of applications with provable information flow guarantees. They extend the type system of a given language to include security labels. Such labels provide information about the security characteristics of the variables in the program, like secrecy, or integrity. Security-typed language compilers can guarantee the $\ast$ (‘no write down’) and simple security (‘no read up’) properties of the Bell-LaPadula model [12], for secrecy, over all data in a given application. They can also guarantee the dual of these policies for integrity, as described in the Biba strict model [15].

By extending all data type declarations with a label that indicates the security level of the data, and then performing a compositional type analysis, a security-typed language compiler is able to catch all illegal explicit and implicit flows in a given program. As a consequence, all programs that successfully compile are also assured to have the
security property of noninterference [36] between high-secure data and low-secure data, for secrecy.

Jif [103, 101] is currently the most mature available security-typed language. Jif enforces a lattice policy where information in a variable with security label \( \ell_1 \) is allowed to flow to a variable with security label \( \ell_2 \), only if \( \ell_2 \) dominates \( \ell_1 \) in the lattice. Label assignment is based on the Decentralized Label Model (DLM) [102], where every owner controls how his/her information flows, including declassification and endorsement operations. Jif uses labels to indicate allowed information flows, thus, legal and illegal flows may be statically checked. Jif manages and implements multiple concepts to build secure applications: principals, labels, declassification, label polymorphism, automatic label inference, and run-time label checking. We introduced these concepts in the next paragraphs.

- In DLM information is owned and controlled by principals. A principal may represent a uses, groups, or roles. The principals may be organized in a hierarchy that defines a partial order [103, 101].

- Labels are annotations that describe how data may be used; they represent policies. In Jif every variable has a label type that consists of a Java type, such as int, and the Jif label. A label is a list of components, and a component is a pair \(<\text{owner} \rightarrow \text{reader\_list}>\), where \text{reader\_list} defines the list of principals allowed to read the data. Also, in a method declaration in Jif, the return value, the arguments, and the exceptions, may be given their own individual label type. A label \(<\text{owner} \rightarrow \text{reader\_list}>\) describes a secrecy policy, while a label \(<\text{owner} \leftarrow \text{writer\_list}>\) describes an integrity policy [103, 101].

- Declassify and Endorsement are operations that allow users to relabel their data. Declassifiers downgrade information enabling information release, and endorsers upgrade information enabling information use [103, 101].

- Automatic Label Inference uses the labels associated to variables and the operations that a piece of code executes with them, to deduce the labels associated to the resultant data. For instance, in an assignation the left hand side variable must meet the security requirements of the expression on the right hand side [103, 101].

- Run-time label checking enables developers to define labels as string variables that are assigned values at run-time [103, 101].
A Jif label consists of a principal e.g. \{alice:}, \{bob:}, or a conjunction of principals e.g. \{alice:;bob:} where the principals are drawn from a principal hierarchy. The special label {} denotes ‘public’ and is always at the bottom of the hierarchy, such that: \( \forall \ell : \{ \} \sqsubseteq \ell \) (public can flow to \( \ell \)).

Figure 2.2: Example of security-typed program written in Jif pseudo-code, demonstrating a leaky program using operating system resources. Jif catches the leaks at compile-time.

Figure 2.2 gives an example of a Jif program. In this example, the Jif application handles two operating system objects: stdin and a Socket. Once a resource is labeled, the Jif type system will ensure, through static analysis, that the label is never violated throughout its lifetime. This means that all leaks, such as the one in Figure 2.2, and more complicated leaks, will be caught at compile-time.

Some applications need to handle illegal information flows under particular conditions, for example network facing daemons need to read low integrity data, or a client program that needs to send encrypted data via a public channel, as the Internet. Security-typed languages use declassification to relabel data in an exceptional way, contrary to the lattice policy. Jif implements robust declassification, allowing a data item’s label to be downgraded under certain conditions.

Figure 2.3: Example of security-typed program written in Jif pseudo-code. It shows relabeling by declassification.

2.1.5 User space security servers in SELinux

Various trusted applications are allowed to access multiple security labels in a system, however, administrators expect them to internally enforce a defined application security policy. A need for user-space security servers has emerged in SELinux, so this kind of applications can integrate SELinux as a security model. For example, the X Window Server in Linux [153]. A user-space security server [147] would manage application policy and answer access control queries for the application, as illustrated in Figure 2.1 for an
operating system.

Although Jif and user space security servers aim to achieve the same goals, they are different in that Jif guarantees complete mediation while applications based on user space security servers cannot. To guarantee complete mediation, policy developers would need to use program analysis tools [106, 33].

2.1.6 Comprehensive Enforcement of Information Flow Policies

Each one of the enforcing mechanisms we presented works independently, at a different software layer. We aim to provide a framework in which administrators may understand the enforcing behavior of a multi-policy system as a whole. Comprehensive enforcement of information flow policies requires supporting mechanisms for each one of the layers (application, OS, network, etc.). However, since each layer has an independent configuration and runs an independent enforcer, we need methods to analyze the rules they impose as a whole.

For instance, to analyze the behavior of a system that includes an operating system, and a trusted program, as a whole, we need: both MAC policies, OS and application, to assign labels, that define security characteristics, to resources; to propagate labels across layers for send and receive operations; and to check that data will not flow through the application in a way that is contrary to the operating system policy [46].

General solutions to these problems guarantee an integrated infrastructure that enforces information flow policies across layers in a single machine, and across machines in a network. Compliance in this case means that there is no information flow in one system that is not allowed in the other, there is no integrity dependence in one system that is not allowed in the other, and there is no information flow across systems that is not allowed by both systems.

2.2 Compliance

In this section we introduce the representation we use to formally model and reason about multi-policy compliance problems. The model describes how to model MAC policies and security goals, and formally defines the meaning of compliance in this representation.

2.2.1 Policy Model

We model an information flow policy as a directed graph $G = (V, E)$ where $V$ represents a set of subject and object labels and $E$ represents a set of directed edges. An edge
between two vertices indicates flow of information between the vertices. Subjects (e.g., processes and users) and objects (e.g., files and sockets) are assigned labels from the same label set $L$. Given a MAC policy $P$, its information flow representation is a directed graph $G_P = (V_P, E_P)$ that represents the information flows authorized by the policy. $V_P$ represents the security labels defined by the policy, and $E_P$ represents the information flows authorized by the policy. To identify information flows, we follow the principle defined in SLAT [40]. We classify the permissions defined by the policy into two categories; read_like and write_like. A read_like permission allows the requester to read from an object, a write_like permission allows the requester to write to an object. There is an edge from a label $l_1$ to a label $l_2$ if $l_2$ has read_like permission for $l_1$ or $l_1$ has write_like permission for $l_2$. Figure 2.4 presents a policy and its information flow representation.

![Information Flow Policy Representation](https://example.com/figure2_4.png)

Figure 2.4: Information Flow Policy Representation. The vertices and the edges in the graph $G$ represent the labels, and the information flows enabled by the policy rules respectively. There are information flows from input1 to manager, from input2 to manager, and from manager to output.

### 2.2.2 Problem Definition

The policy compliance problem involves a comparison between two information flow policies, more precisely between their graph representations. One policy, called the test policy, is compared to the other policy, called the goal policy, to determine if the test complies with the goal. For the test to comply with the goal, it must not introduce new information flows. That is, all the flows authorized by the test policy must also be authorized by the goal policy. Compliance implies that the test policy obeys the requirements (information flows) embodied in the goal policy [50].

In addition, actual policies are usually independently-developed. As a consequence, they have different namespaces and possibly different granularities. Thus to be able to compare two policies, we need a mapping function.

We say that an information flow policy $A$ is compliant with an information flow
policy $B$ if the set of information flows authorized by $A$ is a subset of the information flows authorized by $B$. Formally:

**Definition 2.2.1** (Policy Compliance). An information flow graph $A = (V_A, E_A)$ is compliant with an information flow graph $B = (V_B, E_B)$, given a partial mapping function $f : V_A \rightarrow V_B$, if for any pair of vertices $u$ and $v$ in $V_a$ if there is a path between $u$ and $v$ in $A$, $u \leftrightarrow_A v$, then there exists a path between their mappings in $B$, $f(u) \leftrightarrow_B f(v)$:

$$\forall u, v \in V_a . [(u \leftrightarrow_A v) \rightarrow (f(u) \leftrightarrow_B f(v))]

Figure 2.5 illustrates Definition 2.2.1. The information flow graph $A$, is compliant with the information flow graph $B$, given the partial mapping function $f$. For any pair of vertices $u, v \in V_a$, if there is a path from $u$ to $v$ in $A$, there is also a path from $f(u)$ to $f(v)$ in $B$.

**Figure 2.5:** The Figure shows a partial mapping function $f$ for vertices in information flow graph A to vertices in information flow graph B.

Observe that the mapping function $f$, as defined in Figure 2.5, only maps elements from $A$ into elements in $B$. To be able to compare a test policy against different policy goals we would need different functions (one per goal policy). From a practical point of view this means that evaluating policy compliance to decide about the installation of third party applications would require a particular mapping per system on which the application is going to be installed. Such an approach does not scale; it is not reasonable to expect administrators to create a different mapping for every one of the systems on which an application is going to be deployed, therefore we require a different approach.

We approach this problem by representing goal policies as information flow security models, which are consistent with the requirements defined in multiple systems, and developing heuristics to compute automatically mapping functions. We describe well known information flow security models in the Section 2.2.3.
2.2.3 Information Flow Security Models

Information flow models define how information is allowed to move in a system. They are usually classified into two categories: confidentiality or where information is allowed to go, and integrity or where information is allowed to come from. This section presents known information flow models, their technical foundations, advantages, and disadvantages. We will later use instances of these models to define security goals.

2.2.3.1 Lattices

Dorothy Denning introduced lattice models as a way of expressing information flow policies [31]. She defines an information flow model $FM$ as a tuple $\langle N, P, SC, \oplus, \rightarrow \rangle$, where $N$ is the set of logical storage objects in the target system, $P$ is the set of active agents that are responsible for all information flows and $SC$ is the set of security classes. The class-combining operator $'\oplus'$ specifies the security class for a result, given the security classes of the operands. The flow relation $'\rightarrow'$ is defined on pairs of security classes and indicates that information from the source security class is allowed to flow to the target security class. Under the semantics of information flow policies, the tuple $\langle SC, \oplus, \rightarrow \rangle$ forms a universally bounded lattice.

Lattices have properties that some information flow systems do not have, such as transitivity and least upper bounds. However, such policies may be embedded into lattices, so information flows may be analyzed under the assumption that any information flow model is a lattice [16].

2.2.3.2 Confidentiality Models

Bell-LaPadula (BLP). The Bell-LaPadula model [12] defines an information flow confidentiality policy. It describes a set of rules that govern system decisions to grant or deny access requests from subjects over objects. The model assigns security labels to subjects and objects in the system. The security labels are ordered in a lattice, a higher security label implies more sensitive information. The model also allows the addition of a set of categories, that describe a kind of information, to each security classification. The categories aim to provide access on a 'need to know' basis. The set of categories a subject may access is an element of the power set of the set of categories. A security label and a category form a security level. In the case of the subjects, the level is called a security clearance.

The model defines two rules; the simple security property and the *-property. The
simple security property, also known as no read-up, states that a subject can read an object only if the subject’s clearance dominates the object security level (the subject must have access to the categories the object belongs to). The \(*\)-property, also known as no write-down, states that a subject can write to an object only if the object’s security level dominates the subject’s clearance (the subject must have access to the categories the object belongs to).

Strictly speaking the model would allow a subject to write to objects with a security label higher in the lattice than the security clearance assigned to the subject. Therefore, systems implement a slightly different version of the rule and allow a subject to write to an object only if the clearance of the subject and the security label of the object are equal. Also, some real system implement declassifiers, mechanisms to allow some applications to scape the rules under specific conditions.

**Multi-level Security Model (MLS).** A multi-level security model is a lattice-based model with multiple sensitivity levels. Subjects are assigned security clearances that represent their trustworthiness, and objects are assigned sensitivities. The security levels are ordered in a hierarchy that indicates the direction in which data is allowed to flow [13]. The BLP model and its variants are MLS instances [57].

### 2.2.3.3 Integrity Models

**Biba Integrity Policies.** Biba [15] defines three different integrity policies: the strict policy, the low water-mark policy, and the ring policy.

The strict Biba policy [15] defines an information flow integrity policy. Like Bell-LaPadula, it describes a set of rules that govern system decisions to grant or deny access requests from subjects over objects. This model is considered the dual of Bell-LaPadula but directed towards integrity rather than confidentiality. The model defines two rules: the simple-integrity axiom and the \(*\)-integrity axiom. The former states that a subject may not read objects with a lower integrity label, the latter states that a subject may not write to objects with a higher integrity label. The strict Biba model is too strict for actual systems since some high integrity programs must read low integrity inputs in order to work, for instance, servers and network-facing daemons.

The Low Water-Mark policy is an integrity model that protects the integrity of subjects and objects by following the Biba access rules. However, the labels in this model are not static. After subjects and objects have their initial integrity labels, the system traces updates in the integrity state of its elements via label updates. Subjects and objects are dynamically bound to the greatest lower bound of the integrity levels
they have seen at run time \[34\]. The main limitation of this model is the same as Biba, in practice high integrity processes need to read low integrity inputs, and implement filtering interfaces to protect themselves thus aiming to keep their integrity.

The Ring policy is an integrity policy specifies the following rules. Subjects may read objects at any integrity level, but they may not write to objects with a higher integrity level. Like the strict policy, the ring policy has static labels. The goal of this policy was to increase flexibility.

**Clark-Wilson.** The Clark-Wilson \[24\] integrity model defines rules to maintain data integrity. The model classifies the objects into constrained and unconstrained; the constrained objects have integrity requirements while the unconstrained do not have such requirements.

The model defines certification and enforcement rules. The certification rules define conditions that the system must satisfy in order to maintain integrity. The enforcement rules limit the subjects that may access constrained objects. The model also considers integrity verification and transformation procedures. Integrity verification procedures test that constrained data conform to the system integrity constraints. Transformation procedures transition the system from a valid state to a different valid state. The certification rules (CR) and enforcement rules (ER) are \[16\]:

- **CR1:** Integrity verification procedures must ensure that all constrained data are in a valid state, **CR2:** A transformation procedure transforms constrained data from a valid state to another valid state, **CR3:** The allowed relations must meet the principle of separation of duty, **CR4:** Transformation procedures must generate append-only logs that enable the reconstruction of the operation, and **CR5:** Any transformation procedure that receives unconstrained data as input may only execute valid transitions or no transitions.

- **ER1:** The system must maintain the list of certified relations, i.e., the list of transformation procedures that are certified in the system and the constrained data on which they can operate, **ER2:** The system must maintain a list of allowed relations, i.e., the list of users that are allowed to run a given transformation procedure (transformation procedures run on behalf of determined users), **ER3:** The system must authenticate each user attempting to run a transformation procedure, and **ER4:** Only the certifier of a transformation procedure may change the list of users and constrained data associated with that transformation procedure.

The Clark-Wilson model requires integrity verification of the inputs and formal ver-
ification of the transformation procedures. The latter is difficult to achieve, formal program verification is still an open problem. However, the former is possible, although it is application specific. CW-Lite [136] is an integrity model that focuses on evaluating that untrusted inputs are verified (CR5). CW-Lite’s insight is that high integrity programs read low integrity inputs from a small number of locations. Application developers may annotate their code to indicate the inputs that implement filtering, and system administrators may use these annotations and their definition of integrity over programs in their systems to detect high integrity programs receiving low integrity data without passing first through a filter. Other practical integrity models, like the usable mandatory integrity protection (UMIP) model [80] and the decentralized information flow model (DIFC) [104, 76], also enable some processes to read lower integrity data under specific conditions.

2.2.4 Policy Compliance against a Security Goal

This section presents an adjusted definition of compliance. We now consider a mapping of a policy graph to a security goal represented as a lattice. This definition gives are more flexibility to evaluate compliance for multiple policies, because developers and administrator may specify standard mapping functions, and we may automatically generate mapping functions in some cases. We will study this possibility in the following chapters.

**Definition 2.2.2 (Information Flow Policy).** An information flow policy graph is a directed graph $G = (V, E)$ where $V$ is the set of vertices, and $E$ the set of edges in the graph. Each $v \in V$ is labeled with a label from $B$, the set of labels assigned to subjects and objects in a MAC policy. An edge $(u, v) \in E$ if either (1) $u$ has write_like access to $v$, or (2) $v$ has read_like access for $u$.

**Definition 2.2.3 (Information Flow Goal).** An information flow goal is a security lattice $L = (L, \sqsubseteq)$ where $L$ is a set of labels and the operator $\sqsubseteq$ specifies a partial order on the labels that defines the relation ‘can flow to’.

An integrity information flow goal as a lattice $L_i$. The function $\text{int}(l)$ returns the integrity level associated to any $l \in L_i$. The lattice $L_i$ has a top $\top$ and a bottom $\bot$ elements, $\text{int}(\top) = \text{high}$, and $\text{int}(\bot) = \text{low}$. high $\sqsubseteq_i$ low indicates that high integrity data can flow to low integrity data, but not the other way around.

A confidentiality information flow goal is a lattice $L_c$. The function $\text{conf}(l)$ returns the confidentiality level associated to any $l \in L_c$. The lattice $L_c$ has a top $\top$ and a bottom $\bot$ elements, $\text{conf}(\top) = \text{low}$, and $\text{conf}(\bot) = \text{high}$. low $\sqsubseteq_c$ high indicates
that low confidentiality data can flow to high confidentiality, but not the other way around.

We now formally define the concept of compliance between a graph $G$ and a security lattice $L$. For $u, v \in V(G)$, we write $u \leftarrow_G v$ if there is a path between vertices $u$ and $v$ in the graph $G$.

**Definition 2.2.4** (Policy Compliance -against a security goal-). An information flow policy graph $G = (V, E)$ is compliant with an information flow goal $L = (L, \sqsubseteq)$ given a partial mapping function $h : V \rightarrow L$, if for all $u, v \in V$, if there is a path $u \leftarrow_G v$, then for $h(u), h(v) \in L$, $h(u)$ can flow to $h(v)$:

$$\forall u, v \in V . [(u \leftarrow_G v) \rightarrow (h(u) \sqsubseteq_L h(v))]$$

Figure 2.6 illustrates Definition 2.2.4. It shows an information flow policy graph $G$ and an integrity information flow goal $L$. The information flow graph complies with the information flow goal under the mapping function $int$ because for all vertices in $G$, if there is information flow from $u$ to $v$ in $G$, then the mapping of $u$ can flow to the mapping of $v$ in $L$. For example, $etc_t \leftarrow_G app_t$ and $int(etc_t) \sqsubseteq_L int(app_tmp_t)$.

![Information Flow Graph (G) and Integrity Security Goal (L)](image)

Figure 2.6: Information flow goal, and information flow graph. The graph is compliant with the goal under the mapping function $int$.

Regarding MAC policies, a positive result of the compliance test implies that the information flow graph for a policy does not permit any operation that violates the information flow goals as encoded in a lattice $L$.

### 2.2.5 Problem Complexity

If we can build and solve an instance of the compliance problem, from the MAC policies available for particular applications and operating system, then we can verify whether an application enforces a system policy [48, 47], and decide, based on actual evidence,
whether to install an application or not. Unfortunately, this task in practice poses a significant challenge because these policies are independently-developed. As a consequence, they have distinct label sets and generating a mapping between such label sets is not easy.

Obtaining a mapping function from developers is not practical as such mappings are complex to specify, because of the number of labels, and a particular mapping would be needed for each particular deployment. The automated generation of a compliant mapping function for two policies like the ones Figure 2.5 illustrates would be an instance of the subgraph-isomorphism problem, which is known to be NP-complete. Furthermore, in our case, even if we find one mapping, we do not know how to prove, without any additional information, that such mapping represents the actual relationship between two policies.

By using an information flow security model; like Biba, BLP, or their practical versions to define security goals instead of trying to establish a direct relationship between an application and an operating system policies we approach the problem in a different way. These information flow goals enable us to: (1) define guidelines to map common resources. For example, there are resources that are critical for the integrity of the Linux kernel across different distributions and deployments and for which we can generate a mapping function, and (2) map different application policies independently.

### 2.3 Evolution of Enforcement

Policy enforcement mechanisms are key to our approach as they confine an application’s behavior to those operations specified by the policy. A reference monitor is the basic concept behind any access enforcement mechanism. A reference monitor has three components: interface, authorization module, and policy store. The interface defines the points where access decisions should be made and the arguments needed to make such decisions, the authorization module translates the arguments the interface sends into a query to the policy store, and the policy store answers whether the access request is to be granted or denied. A reference monitor must meet three properties: complete mediation, be tamperproof, and be verifiable [7, 57].

Multics pioneered many of the concepts implemented by modern operating systems, including the reference monitor concept. The Multics reference monitor was implemented by the supervisor, a process that implements the most trusted functionality in the system, similar to a kernel in modern operating systems, but it included more tasks. Since each
instruction in Multics implied access to system resources, the supervisor would make access control decisions per instruction. Eventually, availability of hardware extensions enable hardware to make access control decisions, as it happens with modern operating systems, where processors protect their operating systems and resources by using two levels, supervisor and user, thus some instructions may be executed only by processes that run in supervisor mode. Multics used more than two levels, called rings, to define the resources a process was allowed to access, as well as access control lists (ACLs), and multi-level policy (MLS). All three policies have to authorize an access request for it to be allowed [57].

Modern general-purpose operating systems that implement discretionary access controls, DAC OSs, do not meet the three properties associated to the reference monitor concept, namely complete mediation, tamper-protection, and verifiability. They were developed with different goals in mind. The goal of UNIX developers was to provide a multi-user platform, they adapted some ideas from the Multics project, but simplified them so UNIX was smaller, performed better, and was easier to administer. The goal of the Windows systems was to provide a single-user platform disconnected from the network [57]. In the case of Linux, the addition of the Linux Security Modules (LSM) hooks have promoted research projects to evaluate complete mediation and we have some compelling results [164, 59, 35]. In the case of Windows the security community cannot do much since the code is not available. Tamper-protection cannot be guaranteed in a DAC OS since untrusted users are allowed to change the protection state of the system (the operations that subjects can perform on objects). As a consequence the safety problem is undecidable for DAC operating systems [43]. Additionally, multiple vulnerabilities have been found that enable remote malicious users to locally run processes as root. In UNIX network facing daemons are a main target, as they need both to run with root privileges and receive untrusted inputs to meet their functional requirements. In Windows regular users have administrative permissions, by default, so they can run tasks that require such permissions, like installing an application [57]. Finally, we cannot guarantee implementation correctness, as we do not have the technology to performs this task on code this big. Security kernels were an alternative to general-purpose operating systems. They emphasized verifiable security with reasonable performance [57]. Their design involved three main components, hardware, a minimal kernel, and a small set of trusted services that would also need to be verified. Security kernels are not as flexible as UNIX/Linux and Windows as their verifiability depends on their size. Thus, they cannot support the amount of applications and drivers that these other systems can.
Modern operating systems that implement mandatory access controls, MAC OSs, aim to meet the reference monitor properties. The most known, the Linux Security Modules (LSM) implementation provides complete mediation, tamper-protection, and partially verifiability. As previously mentioned, after the addition of the LSM hooks, security researchers used source code analysis tools to evaluate complete mediation [164, 59, 35]. Regarding tamper-protection, the reference monitor is compiled as part of the kernel, and runs likewise. In addition, only trusted processes are allowed to change the protection state of the system. Finally, although the attempts at proving correctness of the implementation are limited, given the available technology, multiple frameworks are available for administrators to prove the correctness of a given policy [134, 159, 40, 50].

More recently, researchers have made efforts to address the security problem at additional software layers. Protection and security are no longer a concern only at the operating system layer; trusted services are granted privileged permissions that the operating systems cannot control. we count with tools and techniques, as security-typed languages, user level security servers, and source code analysis tools to proof complete mediation at the application layer. Applications still depend on the underlying operating system to guarantee tamper-protection. Proving applications’ correctness is a task limited in the same way that operating systems; by the size of the code and our current technology.

2.4 Related Work

The problem of policy compliance is not new, it has been addressed multiple times in different areas of security research, for example, trust management, policy reconciliation, network policies, and access control models. Section 2.4.1 summarizes the differences between the corresponding definitions of policy compliance. One of the consequences of these differences is that the complexity of the problem varies. The compliance problem is NP-hard in trust management, exponential for network policies, and undecidable for some access control models. In every case researchers look for constraints that change the complexity of the problem while keeping it useful for a wide range of applications.

We focus on policy compliance for a particular area of systems security: end-to-end mandatory access control policies. Previous research projects that evaluate compliance of mandatory access control policies either focus on single policies or evaluate multiple policies that must correspond to the same software layer. Sections 2.4.2 and 2.4.3 present relevant works in these areas. Our work is different from these projects in that we aim
to evaluate compliance for end-to-end systems that enforce mandatory access control policies at multiple hosts and possibly multiple software layers per host. Also, we aim to develop methods to compute automatically security goals and the corresponding mapping function as usually the test policy and the goal have different name spaces.

2.4.1 Compliance in Systems Security

In this section we present the compliance problem as defined in different areas in systems security, namely, trust management, policy reconciliation, network policies, evaluation of access control models, and contextual integrity.

2.4.1.1 Trust Management

Blaze et al. [17] used policy compliance in trust management to evaluate whether the credentials presented by a principal when requesting an action conform to an application’s local policy. The inputs to the compliance checker are a request, a policy, and a set of credentials. The compliance checker returns yes or not, depending on whether the credentials constitute a proof that the request complies with the policy. They show that this notion of proof leads to a compliance-checking problem that is undecidable and is NP-hard even if restricted in several ways. They identify constraints that result in a case of the problem that is solvable in polynomial time and is widely applicable: they bound the functions that the principals can use to define credentials and require them to be monotonic. For example, the length of the set of credentials inferred by the compliance checker is bounded.

Li et al. [79] define D1LP, a logic language to specify authorization with delegation in large-scale, open, distributed systems. For these systems the authorization problem becomes a problem of compliance: whether a set of credentials meets a set of requirements. The authors impose constraints on the syntax of D1LP to make the compliance problem tractable. For example, delegates appearing in delegation rules must be a conjunction of principals, they cannot be a disjunction. D1LP can express a principal issuing a credential, a k-out-of-n threshold structure to authorize a request, a structure that expresses the conjunction or disjunction of requirements, delegation, speaks-for statements, and queries to determine if a set of requirements is met.

Li et al. [81] study security properties, namely, safety and availability for RT (Role base Trust Management) a family of trust management languages. They found that contrary to the classical safety problem, primary security properties for these languages are decidable in polynomial time. The policy language they consider allows the definition
of roles and assignment of principals to the roles, as well as delegation up to two levels. They describe the semantics of a set of policy statements by translating each policy statement into a Datalog clause. Datalog is a restricted form of logic programming with variables, predicates, and constants but without function symbols. They found that it is possible to compute a role’s lower bound (principals that are members of the role in every reachable state) and upper bound (principals that could become a member of the role in some reachable state) in linear time.

Notice that the compliance problem in the most general case for trust management frameworks is NP-hard, mainly because of delegation. However, researchers aim to identify constraints for their models so that they can evaluate compliance in polynomial time.

### 2.4.1.2 Policy Reconciliation

Policy compliance problems may resemble policy reconciliation problems, but they are different. Given two goal policies A and B that define a set of requirements, a reconciliation algorithm looks for a policy instance I to satisfy all the stated requirements [90]. Policy compliance in this scenario means: given a policy instance A and a policy B that defines a set of requirements, does A meet the requirements defined by B?, Is any part of A in conflict with B? Although both of these problems are similar in that they test policy properties, they differ in their inputs and expected outputs. While in the case of reconciliation, an instance that satisfies the requirements will be calculated, in the case of compliance, goals and policy instances are given and the instance is evaluated against the goal.

McDaniel et al. [90] used policy compliance to evaluate whether a policy instance that defines the values of a network communication session is compliant with defined goals on provisioning and authorization for communication sessions. Compliance evaluation in this case is linear in the size of the goal.

### 2.4.1.3 Network Policies

Ritchey et al. [126] use model checking techniques to analyze network vulnerabilities. Their model requires manual specification of exploits, hosts and their initial state, connectivity, states that represent a compromised system, and entry point of the attacker. To perform an analysis a search is conducted from the entry point of the attacker for a host the attacker can communicate with and has the requisites for any of the exploits to succeed, these are added to the entry points of the attacker and the search continues.
Eventually the search will get to the point where a system is compromised or no more exploits can be employed.

FANG [87] aim to test the configuration of a set of firewalls before they are deployed. Fang reads all the configuration files and builds an internal representation of the implied policy and network topology. Given a query it simulates the behavior of various firewalls and computes what packets would reach a destination from a source. The user must manually specify the network topology. Fang aims to answer queries of the form (source-hosts, target-hosts, service). For example, (*,web-servers,http-services) to compute the set of hosts that reach the http services. In the worst case, the complexity of the algorithm is exponential in the size of the zones of the graph. In practice the network topology is a tree most of the time so the search would be a depth-first-search which is linear in the size of the graph.

FIREMAN [158] performs symbolic model checking of firewall configuration for all packets and along all possible paths. The analysis is both sound and complete because the finite nature of the firewall configurations.

Firewall rules are written in platform-specific, low-level languages, so it is difficult to determine whether these rules implement high level security policies. Moreover when these components are installed at multiple network components in an organization. FIREMAN looks for two types of errors: policy violations and inconsistencies. Policy violations happen when packets that are supposed to arrive are blocked and packets that are supposed to be blocked are allowed. Inconsistencies happen at three levels: intra-firewall, inter-firewall, and cross-path.

Intra-firewall inconsistencies are shadowing, generalization, and correlation. Inter-firewall inconsistencies are shadowed accept rules. To the downstream users this may manifest as a connectivity problem. Cross-path inconsistencies are multiple data paths from the Internet to the same protected network. FIREMAN can also look for inefficient rules, that is redundant and verbose ones. Fireman executes in two steps: parser to translate to the intermediate representation, and control-flow analysis. FIREMAN uses cache techniques to reuse analysis and determine errors in linear time.

Compliance evaluation for network policies is exponential largely due to the need to explore all communication paths, which in some cases may involve evaluating a network node multiple times. However, researchers expect response times to be lower in practice, considering the most common network topologies and the use of optimizations such as caching intermediate results and reusing them when possible.
2.4.1.4 Access Control Models

Jones et al. [67] defined the take-grant access control model that defines a type of systems for which the safety problem is decidable in linear time. The model represents a system as a directed graph, and defines rules that specify legal changes to the graph: take, a subject can take rights from another subject; grant, a subject can give away permissions; create, a subject may create new objects; and remove, a subject can remove its rights over an object. The authors show that there is an algorithm for deciding if a subject p can eventually access an object q in linear time in the size of the graph.

Solworth et al. [141] defined another access control model where the safety problem is decidable. An object has a label that defines the privilege that various users need to perform operations on the object. There are two disjoint sets of labels: group labels <U,G> and ordinary object labels <U,N>. Each label is mapped to three groups: r(l) for read, w(l) for write, and x(l) for execute. Relabel privileges are defined as a sequence of relabel rules: rl(t,t'). New users, groups, objects, and labels can be added as the system runs.

The authors defines the following constraints to guarantee decidability:

1. It is decidable whether, starting at a given configuration, existing user U₀ can ever become a member of existing group g₀. "We need not consider the addition of groups, because, while group membership changes over time, the group assigned to a particular permission for a particular label is fixed." Only the group of administrators may relabel the group (add members to the group), so it is possible to check if any of those users may add the user to the target group, or to other groups and in a sequence of relabeling operations to the target group during the final step.

2. Let <U,N> be an ordinary object label, and let U₀ be a user such that U₀ does not belong r(<U,N>). It is decidable whether U₀ can gain r permission for label <U,N> from a given configuration. Let g = r(<U,N>) be the group with r permission. Note that the group assigned a particular permission can never change. Thus the question is reduced to whether U can become a member of group g, that was addressed in (1).

(HRU) Consider a given configuration with object o with label <U,N> such that user U₀ does not belong to r(<U,N>). There is an algorithm to decide whether U₀ can gain r access to o in any configuration reachable from the given configuration. There are two possible types of transitions: (a) changes one or more group tag sets (but leaves o’s
label unchanged) this is equivalent to (1). (b) transition in which o is relabeled. If the current state has initial label l, they use the relabel rules to determine for every other initial label l', the group assigned to rl(l,l'). If that group is nonempty in the current state then add a successor state with label l' (so the search space grows but it is finite).

Notice that in every case authors identify constraints that make the compliance problem in their models solvable in polynomial time. Our analysis of mandatory access controls is similar in that we consider the access control matrix to be fixed, as only trusted administrators can change it. This constraint creates a problem that, contrary to the general case of the safety problem [43], is decidable in linear time.

2.4.1.5 Contextual Integrity

Contextual integrity is a framework for understanding privacy expectations and their implications. It is a philosophical account of privacy in terms of the transfer of personal information. Barth et al. [11] propose a logical framework to express privacy expectations, such as the ones defined by current legislation: the HIPAA regulates the transmission of health information, the children’s online privacy protection act (COPPA), and the Gramm-Leach-Bliley Act (GLBA) that contains privacy provisions to limit how financial institutions can handle non-public personal information.

The model consists of communicating agents that assume various roles in specific contexts and send each other messages about other agents. The evolution of the model depends on the messages they exchange and the computation rules that enable agents to infer attributes. Agents interactions create traces. The conditions of transmission are expressed in linear temporal logic (LTL) formulas interpreted over the traces. A privacy policy regulates what flows of information are permitted between agents in various roles. A policy is a conjunction of contexts, requiring the conditions of each context to be respected. A privacy policy is consistent with a purpose if there is a trace such that the policy and purpose are met. The compliance problem in this framework is: given the sequence of past communications, does the policy permit a contemplated communication and if so, what future requirements are incurred? The authors define weak and strong compliance. Weak compliance ensures that each action taken by agents locally satisfies the privacy policy. It could incur unsatisfiable future requirements. Strong compliance ensures that the agent can meet future requirements, however this problem can only be decided in PSPACE (it involves checking for satisfiability).
2.4.2 Analysis of Individual Policies

The previous sections introduces works that have addressed the problem of compliance for various type of policies. We are interested in evaluation of MAC policies. This sections introduces works that evaluate single MAC policies, all of these works expect administrators to define manually security goals and resolve non-compliant systems.

The SLAT [40], PAL [134], APOL [148], and SELAC [159] projects provide tools to analyze particular properties on single-system MAC policies. They convert MAC operating system policies, such as SELinux [111] and AppArmor, into information flow graphs $G = (V, E)$, where each vertex $v \in V$ represents a policy label, and each edge $(u, v) \in E$ represents a read-like permission for $v$ over $u$, or a write-like permission for $u$ over $v$. These tools mainly evaluate whether it is possible to reach a given label, from a starting one, via a path in the graph. Although, they enable administrators to define manually more queries on the information flow graph, however, that requires administrators to develop a reasonable understanding of the underlying model. The main limitation of these approaches is that they require administrators to determine the security properties they want to evaluate and write them in the language used by the tool, understand the semantics of the results, and define possible solutions to resolve detected errors.

Another project to evaluate compliance is Gokyo [58], it introduces the concept of access control space, the set of all permissions assigned to a subject (or role), and identifies three subspaces for every space: permissible, prohibited, and unknown. Gokyo guides administrators in their policy analysis by identifying conflicts among these subspaces. The idea is that subspaces enable aggregation, so a policy model should be simpler. Gokyo’s developers used the model to analyze the integrity of the TCB for the SELinux example policy [63], a predecessor of the current SELinux reference policy. In their analysis, the authors encoded the goals and the mapping into the tool, and manually solved the conflicts.

Symbion [56] and Security-by-Contract [32] use a different approach. They represent system policies and goals with automata. They both use automata theory to evaluate whether a policy is compliant with a goal. The main limitation of these tools is similar to the previous approaches, administrators are responsible for setting up the problem, that is, defining the security goals and the policy-goal mapping function required to evaluate compliance. Administrators are also responsible for determining possible solutions for non-compliant policies.

It would be possible to analyze composite policies with the previously introduced
tools, if all policies were first translated into the same representation and integrated into a single entity, as proposed by Koch, Mancini, and Parisi-Presicce [73]. They developed a formalism, based on a graph representation of access control policies, to reason about the evolution and integration of various policy access control models into a single representation. Since this work introduces a formal model and does not provide an implementation, administrators would be responsible for manually following the rules to translate their policies to the model. In addition, the approach does not consider automated generation of goals or resolutions for non-compliant systems.

2.4.3 Analysis of Composite Policies

The projects we presented in Section 2.4.2 support analysis of simple policies. In this section, we present different approaches to analyze composite policies. These approaches do not consider multi-layer policies, they expect all policies to be at the same software layer. A consequence of this expectation is that all policies can express access control rules associated to objects that belong to the same software layer, that is, they have the same granularity.

The flexible authorization manager framework, FAM [65], and the system to manage multipolicy access control models, MACS [14], are frameworks for administrators to specify different access control policies within a single system. These frameworks receive requests to determine whether a user $u$ may have access to an object $o$. There are no conflicts across policies in these systems because objects may be governed by only one access control policy. These approaches are designed to specify multi-policy access control models rather than to analyze properties of the models.

Web Access Control Policy Analysis [75] and Policy ratification [2] use formal models to evaluate specific properties; namely, safety, coverage, and conflicts, on composite policies. The former uses description logics (DL), a decidable fragment of first order logics, to represent multiple XACML policies, and evaluate safety properties, where safety means that a particular subject cannot get access to a particular object. The latter uses boolean formulas to represent policies, and goals, and provides a framework to evaluate whether a set of policies are in conflict, that is, given a request they make decisions that cannot be achieved at the same time. Since the satisfiability problem is NP-complete, the authors developed heuristics, and make assumptions that bound the complexity of the problems they deal with. Although these works analyze multiple policies, they require all policies to correspond to the same layer and belong to the same name space. The former condition means that the granularity of the objects in the
policies is the same, the latter means that no policy-goal mapping function is required.

The framework to compare security enhanced operating systems [20] presents an approach to compare security policies and determine which one is more effective. Determining what policy is more effective helps administrators decide what security policy to deploy. This framework addresses a problem that is different from the one we are interested in. It is not concerned with the potential interactions between the policies, and that is one of the key characteristic of our problem.

PolicyGlobe [114], and SPAN [39] provide tools to evaluate reachability for composite systems. While PolicyGlobe uses network policies to compute a connectivity matrix, SPAN needs administrators to define constraints that the system uses to associate different policies in their binary decision diagram representation. These tools are designed to simply check whether an access control request would be granted or denied. Thus, they cannot help administrators define security requirements, interpret the meaning of the results, or provide solutions to resolve problems.

Finally, the layered approach to access control in virtualized environments by Payne et al. [119] and Dstar [163] present architectures to enforce policy in distributed environments at runtime, they focus on enforcement not analysis of the policies. The former proposes an architecture that decomposes policies in VM-based environments into multiple layers: the VMM, the OS, and the application layer. The advantage of the layering approach is that it reduces the complexity of the policies, since the MAC at the hypervisor layer controls information flows between VMs, and MAC at the guest OS controls information flows between applications, instead of a global MAC enforcer controlling information flows across all layers. Although the paper acknowledges the problem of mapping security policies across layers, they assume that there is a service that correctly provides the mapping at runtime, and leave the subject as an open question. The latter, Dstar, provides decentralized information flow controls (DIFC) for distributed applications. The architecture works with independent enforcements mechanisms and policies at every machine, furthermore, every application within a machine defines its own policy. Package exchange across machines relies on the exporters, servers that run at every machine relabeling every input and output package as defined by a predetermined policy. These two approaches assume that there is a predefined service that provides a consistent mapping of security policy across systems and do not address the problem of evaluating consistency (compliance) of the system.
2.4.3.1 On the Composability of Security Properties

Taking a different approach at the problem of composability, some frameworks evaluate whether a system composed of two or more subsystems that meet a specific information flow property meets the same property. It is convenient to have security models that are composable because most systems are actually a set of smaller subsystems that work together with a common goal. Also, a well known design principle is to break up a complex system into smaller components because they are easier to understand [133].

However, McCullough proved [89] that some information flow properties are not composable by default. He showed that it is possible to connect two systems that are proved secure for the MLS non-interference security policy and create a composed system that does not meet the same property. However, he also found that some properties are composable under specific conditions; he introduced restrictiveness or hook-up security, a system is restrictive if it responds the same to a low-level input whether or not a high level input was made immediately before, and proved that two restrictive systems connected legally result in a composite system that is also restrictive.

Alpern and Schneider [3] defined a framework to analyze possibilistic security properties, and Abadi and Lamport [1] extended that framework. They both model a property and a system behavior as trace sets, a property holds for a system if and only if the system traces are a subset of the property traces. The latter extends the former by considering the environment, and determining the rules that guarantee that given a set of subsystems that meet a property, then their composition also meets the same property. McLean [93] states that not every security property of interest is a property on traces, and proposes a framework that uses selective interleaving functions and some composition constructs, such as product and cascading, to evaluate whether a property is composable or not. Mantel [85] further refines the idea of a framework to analyze possibilistic security properties of composite systems.

These frameworks are analytical and determine the conditions under which composability of information flow policies is possible. We simply compose and then check if the composite system meet defined security goals. Leveraging the previous frameworks to analyze our own is part of our future work.

2.4.4 Summary

The policy compliance problem has been addressed in multiple areas of systems security, including trust management for distributed systems, policy reconciliation, network
policy design, access control models, and more recently contextual integrity. Although the policy compliance problem is the same at a high level, that is, to evaluate whether a test policy meets the requirements defined by a goal policy, the precise specification of the problem varies in every area of research and as a consequence its complexity. The compliance problem is NP-hard for trust management frameworks because of delegation [17], exponential for network policies because of the need to explore all network paths which in some cases may involve evaluating a network node multiple times [87], and undecidable for protection systems [43]. However, researchers look for constraints in every case that reduce the complexity of the problem while keeping it useful for a wide range of applications. Our research focuses on end-to-end MAC policies. We also define constraints that bound the complexity of our compliance problem. We consider the access control matrix to be fixed as only trusted administrators can change it. This constraint creates a problem that, contrary to the general case of the safety problem for protection systems [43] is decidable in polynomial time.

Previous research projects that evaluate compliance of mandatory access control policies either focus on single policies or evaluate multiple policies that must correspond to the same software layer. Approaches that evaluate single MAC policies expect administrators to define manually security goals and mapping between the policy and the goals. While this approach is reasonable for single systems, it does not scale for end-to-end MAC systems that involve multiple policies. Approaches that evaluate composite policies expect all policies to correspond to the same software layer. Our goal is to evaluate compliance for end-to-end MAC systems that enforce multiple MAC policies not only at the same software layer but possibly at multiple software layers.

2.4.5 Error Resolution

If we detect errors in the policies for a given system, we will need to compute a solution. Intuitively an error is a conflict between the actual policy and a goal, where we may represent the goal via security models, properties, assertions, or constraints. Various researches address this problem and propose diverse mechanisms to compute solutions.

XACML [112] and the multi-policy management frameworks proposed in [73] and [65] have built-in operators to resolve conflicts. Operators such as permit-overrides, deny-overrides, first applicable, and priority for one policy, define the rules that have precedence. This approach does not apply to our model as the infrastructure we are representing does not have this kind of built-in operators.

An alternative approach is to use constraint solvers to compute a solution. Although
the SAT problem is known to be NP-complete, modern SAT solvers can solve millions of constraints in seconds. SMT solvers (Satisfiability Module Theories) work similarly. An SMT instance is a formula in first order logic where some functions and predicate symbols have interpretations in a background theory such as integers, real numbers, or lists. Examples of research works that use SAT or SMT solvers to resolve conflicts include ConfigAssure [105], and the framework for analysis and repairing of network policies while considering additional requirements [53].

A different approach is to look for optimal solutions, given particular criteria. The next paragraphs describe works that use this approach. Jaeger et al. [64] look for a solution with the maximal impact. That is, given a set of conflicts, look for the solution that will resolve the higher number of such conflicts, where all conflicts have the same priority.

Finally, another option is to look for a minimum-cost solution that will solve all problems, as proposed in Efficient Minimum-Cost Network Hardening Via Exploit Dependency Graph [108] and Automating Security Mediation Placement [72]. The former, models network attacks as exploit dependency graphs, graphs that represents the dependencies among initial conditions and exploits (the exploits that are reachable from the initial conditions, and the initial conditions that are relevant -backward-reachable- from the exploits). It translates an exploit dependency graph into a term in canonical conjunctive normal form, a conjunction of maxterms where a boolean variable represents a network condition, and each maxterm represents a safe assignment of initial conditions. Finally, it picks the maxterm that implies minimum cost hardening actions.

The latter uses an information flow graph to represent all flows that are enabled by a security-typed program, and generates a minimum cut of the graph that is equivalent to the points at which mediators should be placed for the program to satisfy the associated type system. We aim to use a solution that can represent different costs, so we can analyze how to generate solutions for non-compliant systems from different points of view, and select an adequate solution.
Some programs require access to multiple secrecy levels to provide the functionality they were designed for; these programs are known as trusted programs. Since the operating system can only control processes access at the granularity of inputs and outputs, it cannot control how trusted programs internally handle the data they have access to. These programs could leak information, for example send secret information over public network channels. We want to evaluate whether the policy a trusted program enforces is compliant with the policy that the underlying operating system enforces, and as a consequence whether a program may be trusted or not. We developed a Service for Inspection and Execution of Security Typed Applications, SIESTA [47] to address this requirement. SIESTA is a system service that evaluates compliance at run time.

SIESTA runs in user space, and evaluates compliance of security-typed applications with the MLS policy of the operating system on which they both, SIESTA and the application, run. Based on the information SIESTA returns, the operating system may decide to run only applications that meet the system policy. Our SIESTA prototype evaluates compliance of an SELinux MLS policy and a Jif application with externally declared policies, but the approach is not limited to these systems, it may be extended by adding parsers to translate other policies to the same representation.

To develop SIESTA, we first developed an analytical model to represent and reason about the SELinux MLS policy model, we then determined the requirements to evaluate compliance of a Jif application against the SELinux MLS policy [50, 52], and finally designed and implemented the service. In the next sections we describe these steps.
3.1 SE Linux MLS Policy Model

Understanding the semantics of the SE Linux MLS policy is useful for the following purposes:

- Evaluating whether all operations corresponding to system resources are constrained by the policy, thus all types of access are mediated.

- Determining whether the policy faithfully implements the simple security condition and \( \ast \)-property (both concepts introduced in 2.2.3.3).

- In distributed systems, a service may need to determine whether two MLS policies are compliant [62]. In cases where a MAC-based OS needs to trust an application to handle multiple levels of data, it is important for the OS to determine whether the application’s information flow policy complies with its own [51].

SE Linux implements three different security models [139]: the Type Enforcement (TE) model, the Role based access control model (RBAC), and the Multi Level Security model (MLS). Trusted Computer Systems (TCS) [41] recently developed the current implementation of the MLS model. The MLS model is mostly orthogonal to the TE model, so their interaction is limited. This model associates an MLS level to every subject and object in the system. For every security-sensitive operation, a set of MLS constraints is evaluated based on the MLS levels assigned to the subject and the object, and the access mode associated to the operation. The SE Linux distribution provides a standard MLS policy that seeks to implement a confidentiality policy according to definitions by Bell and LaPadula [12]. It aims to meet the simple and \( \ast \)-properties.

While TE policies aim to enforce the principle of least privilege [133], multi-level security policies aim to control how information is allowed to flow between subjects in a system. An MLS policy assigns subjects a clearance that indicates the sensitivity of the information they are allowed to access and objects a sensitivity depending on the information they store, and restricts how information may flow between designated sensitivities. As an example, consider a military application with four sensitivities ordered from least to most sensitive: Unclassified (UC), Confidential (CO), Secret (S), and Top Secret (TS). Note that in this example, the sensitivities form a total ordering in which each sensitivity is higher, lower, or equal to another one. This is not always the case, i.e., they may be organized in a partial order and some sensitivities may be incomparable.

Typically, MLS defines information flow policies based on two properties: the simple security condition and the \( \ast \)-property. The simple security condition also known as ‘no
read up' requires a subject S to dominate an object O to have read rights. That is, the subject’s clearance must dominate the object’s sensitivity. The *-property also know as ‘no write down’ requires the object’s sensitivity to dominate the subject’s clearance for the subject to have write rights. To allow finer granularity in the model, the MLS model was expanded with categories. Categories group information of the same kind, thus access may be granted to subjects on a need-to-know basis. Categories provide a way to allow access to certain types of data, while staying within the confines of the sensitivity restrictions. A subject must then have a superset of the object’s categories to dominate the object.

In the next sections, we present syntax and semantics for SELinux MLS policy rules. We begin with an informal description of SELinux policy rules, followed by our model, and finally the formal semantics for the rules.

3.1.1 Labels

SELinux may work with the MLS extension enabled or disabled. In an SELinux system with the MLS extension enabled, a label (called security context in the SELinux community) has four parts: user, role and type (which are also used for the RBAC and TE models), and an MLS range defined by low and high MLS levels. Each level has a sensitivity level and an optional set of category compartments. The sensitivity represents MLS clearance on subjects and sensitivity classification on objects, while the categories represent a set of non-hierarchical compartments to which the subject may have access. For example, staff_u:staff_r:staff_t:s0-s2:c0.c15 is a possible SELinux security context.

In this security context, colon (:) separates the parts in the context. The last part, s0-s2:c0.c15, has the MLS range (s0-s2) and the categories c0.c15 of the context. Most objects in an SELinux system have the same value for their low and high levels (they are single-level). However, there are some exceptions like multi-level directories. On the other hand, it is common for subjects to have different low and high levels. The low level is the current security clearance of the subject and the high level represents the upper bound in security clearance for the same subject. In the example, staff_u:staff_r:staff_t:s0-s2:c0.c15, s0 is the low level and s2 is the high level. In addition, at s2, the user has access to compartments c0 through c15 (including all the compartments that lie in between).

Although an SELinux policy may implement the TE, RBAC, and MLS models, we focus on the MLS part of the policy here. The MLS policy language provides the following
statements for a user to define a specific MLS policy: sensitivity, category, level, dominance, mlsconstrain and mlsvalidatetrans. In the next section we describe these statements.

3.1.2 Syntax

In this section we present the SELinux MLS statements. First we introduce the notation we will use.

\[s: \text{Security context for a subject}\]
\[o: \text{Security context for an object}\]
\[c: \text{Class}\]
\[p: \text{Mode in which an object may be accessed}\]
\[C: \text{Set of classes}\]
\[P: \text{Set of modes}\]
\[u: \text{user}\]
\[r: \text{role}\]
\[t: \text{type}\]
\[sl: \text{Sensitivity level}\]
\[ca: \text{Category}\]
\[exp: \text{Boolean expression}\]
\[Policy: \text{SELinux policy}\]
\[stmt: \text{Policy rule (including TE and RBAC)}\]

**sensitivity**: Sensitivities in an MLS model represent security clearance for subjects or security classification for objects. The set of sensitivity statements define the set of valid sensitivities in a particular SELinux system.

SELinux syntax:

```plaintext
sensitivity id [ alias id_set ];
```

**category**: Categories expand an MLS model by making it possible to represent different families of data associated with each sensitivity. For example, categories allow us to make a distinction between Top Secret Nuclear and Top Secret Political data. Top Secret is the sensitivity, and Nuclear and Political define two different categories.

SELinux syntax:

```plaintext
category id [ alias id_set ];
```

**level**: MLS levels define legal combinations of sensitivities and category sets.

SELinux syntax:

```plaintext
level sl : [ ca_set ];
```

**dominance**: MLS sensitivities are organized into a hierarchy. Higher sensitivities represent higher security clearances or higher security classification. The first sensitivity in the dominance statement is assigned the lowest position in the hierarchy and the last
element is assigned the highest position.

SELinux syntax: `dominance \{ sl_{1}sl_{2}...sl_{n} \}`

**mlsconstrain:** This statement restricts access rights assigned in an SELinux policy, according to relationships between the security context of the subject that requests access and the security context of the target object, the class of the target object, and the mode in which the subject wants to access the object. In SELinux objects are classified into classes (filesystem, file, dir,...), and for each class a set of access modes is defined (read, write, create,...).

SELinux syntax: `mlsconstrain C P exp;`

**mlsvalidatetrans:** This statement restricts the ability of a subject to change the security context of a target object, based on the relationships among the security context of the subject that requests the change, the new and old contexts of the target object, and the class of the target object.

SELinux syntax: `mlsvalidatetrans C exp;`

**Example 3.1.1** (MLS setting). This example defines a possible MLS policy. The following rules define sensitivities s0 to s2, a total order on those elements, and the set of legal sensitivity-category combinations.

```mls
sensitivity s0;
sensitivity s1;
sensitivity s2;
dominance \{ s0 s1 s2 \}
category c0;
category c1;
category c2;
level s0:c0.c2;
level s1:c0.c2;
level s2:c0.c2;
```

### 3.1.3 Semantics

In this section we present the analytical model we have developed to understand the meaning of an SELinux MLS policy. We progressively introduce operators that will support our analysis.

This part of the section presents four basic operators: `name`, `classes`, `modes` and `expr`. `name` gets the name of a given rule, `classes` gets the set of classes a rule applies to, `modes`
gets the set of modes a rule applies to, and \textit{expr} gets the boolean expression associated to a rule. Notice that not all of the operators are defined for all the MLS rules; \textit{classes} and \textit{expr} are defined only for \textit{mlsconstrain} and \textit{mlsvalidatetrans}, and \textit{modes} is defined only for \textit{mlsconstrain}. The operators \textit{classes} and \textit{modes} also apply to a Policy. In this case they respectively return all the classes declared, and all the modes in which objects may be accessed, in the specified Policy. Next we present examples with these operators.

\textbf{Example 3.1.2.} Operators \textit{name}, \textit{classes}, \textit{modes}, and \textit{expr}.

\begin{align*}
\text{name}(\text{sensitivity} \ s_1) &= \text{sensitivity} \\
\text{name}(\text{category} \ c_0) &= \text{category} \\
\text{name}(\text{level} \ s_1 : c_0, c_1, c_2) &= \text{level} \\
\text{name}(\text{dominance} \ \{s_1, s_2, s_3, s_4\}) &= \text{dominance} \\
\text{classes}(\text{mlsconstrain} \ \text{file} \ \{\text{create relabelto}\} \ (l_2 \ eq \ h_2)) &= \{\text{file}\} \\
\text{modes}(\text{mlsconstrain} \ \text{file} \ \{\text{relabelto}\} \ (l_2 \ eq \ h_2)) &= \{\text{relabelto}\} \\
\text{expr}(\text{mlsconstrain} \ \text{file} \ \{\text{create relabelto}\} \ (l_2 \ eq \ h_2)) &= (l_2 \ eq \ h_2)
\end{align*}

The model also includes operators to get the components of a given security context: \textit{getu}, \textit{gett}, \textit{getr}, \textit{getl} and \textit{geth}. They take a security context \((\text{user}, \text{role}, \text{type}, \text{mls})\), where the MLS part is a pair \((l, h)\) that represents an MLS range, and return the components of the tuple.

\begin{align*}
\text{getu}((\text{user}, \text{role}, \text{type}, (1, h))) &= \text{user} \\
\text{getr}((\text{user}, \text{role}, \text{type}, (1, h))) &= \text{role} \\
\text{gett}((\text{user}, \text{role}, \text{type}, (1, h))) &= \text{type} \\
\text{getl}((\text{user}, \text{role}, \text{type}, (1, h))) &= 1 \\
\text{geth}((\text{user}, \text{role}, \text{type}, (1, h))) &= h
\end{align*}

SELinux also has a dominance rule that defines a total order over the MLS sensitivities.

\begin{align*}
\text{dominance}(sl_1, sl_2, \ldots, sl_n) &\equiv \text{induces an order } \sqsubseteq \text{ on the elements } sl_1, sl_2, \ldots, sl_n \text{ s.t. } \\
&\quad sl_1 \sqsubseteq sl_2 \sqsubseteq \ldots \sqsubseteq sl_n.
\end{align*}

The model includes operators to get the components of a particular MLS level and to compare two MLS levels. \textit{getsens} and \textit{getcat} get sensitivity and category of a MLS level respectively. \textit{==}, \textit{!=}, \textit{dom}, \textit{domby} and \textit{incomp} compare two MLS levels. The result depends on the order defined by the dominance statement for sensitivities, and on the set defined by the categories associated with each level.
\[ opl(\equiv, l_1, l_2) = (l_1 = l_2) \]
\[ opl(!=, l_1, l_2) = (l_1 \neq l_2) \]
\[ opl(\text{dom}, l_1, l_2) = (\text{getsens}(l_2) \subseteq \text{getsens}(l_1)) \land (\text{getcat}(l_2) \subseteq \text{getcat}(l_1)) \]
\[ opl(\text{domby}, l_1, l_2) = (\text{getsens}(l_1) \subseteq \text{getsens}(l_2)) \land (\text{getcat}(l_1) \subseteq \text{getcat}(l_2)) \]
\[ opl(\text{incomp}, l_1, l_2) = \neg(opl(\text{dom}, l_1, l_2)) \land \neg(opl(\text{domby}, l_1, l_2)) \]

Dominance over roles is defined in a way that is analogous to the dominance over levels. Thus, the operators \text{dom}, \text{domby} and \text{incomp} also apply. Roles are not directly related to MLS policies, but they are used to express conditions on MLS constraints. Therefore, we include them in the model.

\[ \text{dominance}(r_1, r_2, ..., r_n) \equiv \text{induces a partial order, } \subseteq \text{ over the elements } r_1, r_2, ..., r_n \text{ s.t. } r_1 \subseteq r_2 \subseteq ... \subseteq r_n. \]

\[ opr(\equiv, r_1, r_2) = (r_1 = r_2) \]
\[ opr(!=, r_1, r_2) = (r_1 \neq r_2) \]
\[ opr(\text{dom}, r_1, r_2) = (r_2 \subseteq r_1) \]
\[ opr(\text{domby}, r_1, r_2) = (r_1 \subseteq r_2) \]
\[ opr(\text{incomp}, r_1, r_2) = \neg(r_1 \subseteq r_2) \land \neg(r_2 \subseteq r_1) \]

We define an operator to generate the set of all valid ranges in a given Policy. Recall that some subjects and multi-level objects have access to multiple MLS levels. SELinux makes this possible through MLS ranges, but not every range is allowed. The following operator defines the set of legal ranges for a given policy.

\[ \text{ranges}(\text{Policy}) = \{(l_1, l_2) \mid l_1, l_2 \in \text{levels}(\text{Policy}) \land (\text{getsens}(l_1) \subseteq \text{getsens}(l_2)) \land (\text{getcat}(l_1) \subseteq \text{getcat}(l_2))\} \]

The definition of the previous operators is straightforward. They serve primarily to support the main definition that is based on the operators \(\gamma_{MLS}\) and \(\gamma_{MLSvt}\). These operators determine the result of applying all relevant constraints to a particular subject, object, object class, and access mode. If the result of applying all relevant constraints (a possibly empty set) is true, then the result for the operator is true, otherwise it is false.

**Definition 3.1.3** \((\gamma_{MLS})\). \(\gamma_{MLS}\) detects and evaluates the MLS constraints that apply when a subject is trying to access a given object.

\[ \gamma_{MLS}(s, o, c, p) = \{\text{stmt} \mid \text{stmt} \in \text{Policy}, \text{name}(\text{stmt}) = \text{mlsconstrain}, c \in \text{classes}(\text{stmt}), p \in \text{modes}(\text{stmt}), \| \text{expr}(\text{stmt}) \|_{s,o} = \text{FALSE} \} = \emptyset \]

Next we present an inductive definition for the semantics of \(\| \text{expr}(\text{stmt}) \|_{s,o} \) in \(\gamma_{MLS}\). \(s\) represents the subject that is requesting the operation that initiates the check
of the constraint, and \( o \) is the object that \( s \) attempts to access. \( \text{opt} \) represents any of the operators defined to compare MLS levels, i.e., \( ==,!,\neq,\text{dom},\text{domby},\text{and incomp} \). In addition, the values of the fields user, role and type from the subject’s security context or the object’s security context may be tested against predefined values. The same operations may be evaluated for \( u2 \) (object’s user), \( r1 \) and \( t1 \) (subject’s role and type) and \( r2 \) and \( t2 \) (objects’ role and type), supported by the operators \( \text{getr} \) and \( \text{gett} \).

\[
\begin{align*}
| \text{not}(\text{exp}) |_{s, o} &= \neg (| \text{exp} |_{s, o}) \\
| \text{exp}_a \text{ and } \text{exp}_b |_{s, o} &= | \text{exp}_a |_{s, o} \wedge | \text{exp}_b |_{s, o} \\
| \text{exp}_a \text{ or } \text{exp}_b |_{s, o} &= | \text{exp}_a |_{s, o} \vee | \text{exp}_b |_{s, o} \\
| u1 = u2 |_{s, o} &= (\text{getu}(s) = \text{getu}(o)) \\
| u1 \neq u2 |_{s, o} &= (\text{getu}(s) \neq \text{getu}(o)) \\
| t1 = t2 |_{s, o} &= (\text{gett}(s) = \text{gett}(o)) \\
| t1 \neq t2 |_{s, o} &= (\text{gett}(s) \neq \text{gett}(o)) \\
| l1 \text{ opt } l2 |_{s, o} &= \text{opr}(\text{opt}, \text{getl}(s), \text{getl}(o)) \\
| h1 \text{ opt } h2 |_{s, o} &= \text{opr}(\text{opt}, \text{geth}(s), \text{geth}(o)) \\
| l1 \text{ opt } h2 |_{s, o} &= \text{opr}(\text{opt}, \text{geth}(s), \text{getl}(o)) \\
| h1 \text{ opt } h2 |_{s, o} &= \text{opr}(\text{opt}, \text{getl}(s), \text{geth}(o)) \\
| l1 \text{ opt } l1 |_{s, o} &= \text{opr}(\text{opt}, \text{getl}(s), \text{getl}(o)) \\
| r1 \text{ operator } r2 |_{s, o} &= \text{opr}(\text{operator}, \text{getr}(s), \text{getr}(o)) \\
| u1 = \text{userset} |_{s, o} &= (\text{getu}(s) \in \text{userset}) \\
| u1 \neq \text{userset} |_{s, o} &= (\text{getu}(s) \notin \text{userset}) \\
| u2 = \text{set} |_{s, o} &= (\text{getu}(o) \in \text{userset}) \\
| u2 \neq \text{set} |_{s, o} &= (\text{getu}(o) \notin \text{userset}) \\
| r1 = \text{roleset} |_{s, o} &= (\text{getr}(s) \in \text{roleset}) \\
| r2 = \text{roleset} |_{s, o} &= (\text{getr}(o) \in \text{roleset}) \\
| r1 \neq \text{roleset} |_{s, o} &= (\text{getr}(s) \notin \text{roleset}) \\
| r2 \neq \text{roleset} |_{s, o} &= (\text{getr}(o) \notin \text{roleset}) \\
| t1 = \text{typeset} |_{s, o} &= (\text{gett}(s) \in \text{typeset}) \\
| t2 = \text{typeset} |_{s, o} &= (\text{gett}(o) \in \text{typeset}) \\
| t1 \neq \text{typeset} |_{s, o} &= (\text{gett}(s) \notin \text{typeset}) \\
| t2 \neq \text{typeset} |_{s, o} &= (\text{gett}(o) \notin \text{typeset})
\end{align*}
\]

**Example 3.1.4.** This example shows the behavior of \( \gamma_{\text{MLS}}(s, o, c, p) \). A user with MLS range \( s1-s2 \) has a file with MLS level \( s1 \), and tries to upgrade his file to \( s2 \). We must check all of the rules that handle the access modes required by the operation. In this case, that requires checking three sets of rules involving files: \( \text{mlsconstrain} \) rules for the access mode \( \text{relabelto} \), \( \text{mlsconstrain} \) rules for the access mode \( \text{relabelfrom} \), and the rules for \( \text{mlsvalidatetrans} \) on files.
Step 1: relabelto rules. The following mlsconstrain rules are checked when accessing a file in mode relabelto

mlsconstrain {file lnk_file fifo_file} { create relabelto } (l2 eq h2);
mlsconstrain {dir file lnk_file chr_file blk_file} relabelto (h1 dom h2);

The evaluation of these constraints gives:

\[ \gamma_{MLS}(\text{staff}_u:\text{staff}_r:\text{staff}_t:s_1-s_2:c_0.c_2,\]
\[ \text{staff}_u:\text{object}_r:\text{user}_\text{home}_\text{dir}_t:s_2, \text{file}, \text{relabelto}) = \text{TRUE} \]

Step 2: relabelfrom rules. The following mlsconstrain rule is also checked.

mlsconstrain = { file lnk_file fifo_file }
{ write create setattr relabelfrom rename }
( (l1 eq l2) or
((t1 == mlsfilewritetoclr) and (h1 dom l2) and (l1 domby l2)) or
(t1 == mlsfilewrite) or
(t2 == mlstrustedobject)
);

The evaluation of this constraint gives:

\[ \gamma_{MLS}(\text{staff}_u:\text{staff}_r:\text{staff}_t:s_1-s_2:c_0.c_2,\]
\[ \text{staff}_u:\text{object}_r:\text{user}_\text{home}_\text{dir}_t:s_1, \text{file}, \text{relabelfrom}) = \text{FALSE} \]

Step 3: mlsvalidatetrans rules. The evaluation of mlsvalidatetrans rules depends on \( \gamma_{MLSvt} \). We present that definition in the next paragraphs and we complete the test afterwards (Example 3.1.6).

Definition 3.1.5 (\( \gamma_{MLSvt} \)). \( \gamma_{MLSvt} \) detects the result of the constraints that apply when a subject is trying to change the MLS level assigned to a given object.

\[ \gamma_{MLSvt}(o_1,o_2,s,c) = (\{ stmt \mid stmt \in \text{Policy, } name(stmt) = \text{mlsvalidatetrans},\]
\[ c \in \text{classes}(stmt), \| expr(stmt) \|_{o_1.o_2,s} = \text{FALSE} \} = \emptyset) \]

Next, we present an inductive definition for the semantics of \( \| expr(stmt) \|_{o_1.o_2,s} \) in \( \gamma_{MLSvt} \). These definitions are similar to the ones presented for mlsconstrain, though an important difference is that mlsvalidatetrans takes three inputs instead of two. The input \( o_1 \) is the old security context, \( o_2 \) is the new security context, and \( s \) is the security context of the process that requests the transition. In the boolean expression, elements
indexed with 1 (u₁,r₁,t₁) make reference to o₁, elements indexed with 2 (u₂,r₂,t₂) make reference to o₂, and elements indexed with 3 (u₃,r₃,t₃) make reference to s.

\[
\| \text{not}(exp) \|_{o₁,o₂,s} = \neg (\| exp \|_{o₁,o₂,s})
\]
\[
\| exp_a \text{ and } exp_b \|_{o₁,o₂,s} = \| exp_a \|_{o₁,o₂,s} \land \| exp_b \|_{o₁,o₂,s}
\]
\[
\| exp_a \text{ or } exp_b \|_{o₁,o₂,s} = \| exp_a \|_{o₁,o₂,s} \lor \| exp_b \|_{o₁,o₂,s}
\]

Next, we define the meaning of boolean expressions for mlsvalidatetrans. opt represents any of the operators we defined to compare roles and MLS levels.

\[
\|
\begin{align*}
&u₁ \equiv u₂ \|_{o₁,o₂,s} = (\text{getu}(o₁) = \text{getu}(o₂)) \\
&u₁ \neq u₂ \|_{o₁,o₂,s} = (\text{getu}(o₁) \neq \text{getu}(o₂)) \\
&r₁ \text{ opt } r₂ \|_{o₁,o₂,s} = \text{opr}(\text{opt}, \text{getr}(o₁), \text{getr}(o₂)) \\
&t₁ \equiv t₂ \|_{o₁,o₂,s} = (\text{gett}(o₁) = \text{gett}(o₂)) \\
&t₁ \neq t₂ \|_{o₁,o₂,s} = (\text{gett}(o₁) \neq \text{gett}(o₂)) \\
&l₁ \text{ opt } l₂ \|_{o₁,o₂,s} = \text{opl}(\text{opt}, \text{getl}(o₁), \text{getl}(o₂)) \\
&l₁ \text{ opt } h₂ \|_{o₁,o₂,s} = \text{opl}(\text{opt}, \text{getl}(o₁), \text{geth}(o₂)) \\
&h₁ \text{ opt } l₂ \|_{o₁,o₂,s} = \text{opl}(\text{opt}, \text{geth}(o₁), \text{getl}(o₂)) \\
&h₁ \text{ opt } h₂ \|_{o₁,o₂,s} = \text{opl}(\text{opt}, \text{geth}(o₁), \text{geth}(o₂)) \\
&l₁ \text{ opt } h₁ \|_{o₁,o₂,s} = \text{opl}(\text{opt}, \text{getl}(o₁), \text{geth}(o₁)) \\
&l₂ \text{ opt } h₂ \|_{o₁,o₂,s} = \text{opl}(\text{opt}, \text{getl}(o₂), \text{geth}(o₂))
\end{align*}
\]

Similar to the mlsconstrain semantics, the values of the fields user, role and type from the involved security contexts may be tested against predefined values:

\[
\|
\begin{align*}
&u₁ \equiv \text{userset} \|_{o₁,o₂,s} = (\text{getu}(o₁) \in \text{userset}) \\
&u₁ \neq \text{userset} \|_{o₁,o₂,s} = (\text{getu}(o₁) \notin \text{userset}) \\
&u₂ \equiv \text{userset} \|_{o₁,o₂,s} = (\text{getu}(o₂) \in \text{userset}) \\
&u₂ \neq \text{userset} \|_{o₁,o₂,s} = (\text{getu}(o₂) \notin \text{userset}) \\
&r₁ \equiv \text{roleset} \|_{o₁,o₂,s} = (\text{getr}(o₁) \in \text{roleset}) \\
&r₁ \neq \text{roleset} \|_{o₁,o₂,s} = (\text{getr}(o₁) \notin \text{roleset}) \\
&r₂ \equiv \text{roleset} \|_{o₁,o₂,s} = (\text{getr}(o₂) \in \text{roleset}) \\
&r₂ \neq \text{roleset} \|_{o₁,o₂,s} = (\text{getr}(o₂) \notin \text{roleset}) \\
&t₁ \equiv \text{typeset} \|_{o₁,o₂,s} = (\text{gett}(o₁) \in \text{typeset}) \\
&t₁ \neq \text{typeset} \|_{o₁,o₂,s} = (\text{gett}(o₁) \notin \text{typeset}) \\
&t₂ \equiv \text{typeset} \|_{o₁,o₂,s} = (\text{gett}(o₂) \in \text{typeset}) \\
&t₂ \neq \text{typeset} \|_{o₁,o₂,s} = (\text{gett}(o₂) \notin \text{typeset})
\end{align*}
\]

mlsvalidatetrans involves a third security context, therefore we define analogous operators to handle that context. For instance:
∥ u3 == userset ∥ o1:o2,s = (getu(s) ∈ userset)
∥ u3 != userset ∥ o1:o2,s = (getu(s) ∉ userset)
∥ r3 == roleset ∥ o1:o2,s = (getr(s) ∈ roleset)
∥ r3 != roleset ∥ o1:o2,s = (getr(s) ∉ roleset)
∥ t3 == typeset ∥ o1:o2,s = (gett(s) ∈ typeset)
∥ t3 != typeset ∥ o1:o2,s = (gett(s) ∉ typeset)

Example 3.1.6. In this example we evaluate the step we left pending in Example 3.1.4. Recall that we are evaluating the rules that apply when a user with MLS range s1-s2 that owns a file with MLS level s1, tries to upgrade the MLS level associated to the file to s2. We already showed the results for γ_{MLSvt}(s,o,c,p). Now we show the result of γ_{MLSvt}(o1,o2,s,c).

In the current SELinux policy there is only one mlsvalidatetran statement. It states that any change of the context of an object is possible only if the old context, the new context, and the context of the process that request the change meet a precise condition:

```
# the file upgrade downgrade rule
mlsvalidatetran { dir file lnk_file chr_file blk_file sock_file fifo_file }
( ( ( l1 eq l2 ) or
  ( ( t3 == mlsfileupgrade ) and ( l1 domby l2 ) ) or
  ( ( t3 == mlsfiledowngrade ) and ( l1 dom l2 ) ) or
  ( ( t3 == mlsfiledowngrade ) and ( l1 incomp l2 ) )
) and ( ( h1 eq h2 ) or
  ( ( t3 == mlsfileupgrade ) and ( h1 domby h2 ) ) or
  ( ( t3 == mlsfiledowngrade ) and ( h1 dom h2 ) ) or
  ( ( t3 == mlsfiledowngrade ) and ( h1 incomp h2 ) )
)
```

The condition in the statement indicates that the type of the process that makes the request, t3, has attribute mlsfileupgrade or mlsfiledowngrade, and the low or high levels, li and hi respectively, of the old and new contexts, 1 and 2 respectively, meet the conditions. For example, the clause ( t3 == mlsfileupgrade ) and ( 11 domby 12 ) specifies that the context of the process making the request has the attribute mlsfileupgrade and the low mls level of the old context, 11, is dominated by the low mls level of the new context, 12.

The result is FALSE as the condition is not met:

\[ γ_{MLSvt}( \text{staff}_u:object_r:user_home_dir_t:s1, \text{staff}_u:object_r:user_home_dir_t:s2, \text{staff}_u:staff_r:staff_t:s1-s2:c0.c2, \text{file}) = FALSE \]
For an operation to be authorized all checks, $\gamma_{MLS}(s, o, c, p)$ and $\gamma_{MLSvt}(o1, o2, s, c)$ must be TRUE. Thus, this example is not authorized.

The analytical model described in this section offers a logical framework to analyze MLS policies. The size and scope of the policies inhibits doing any realistic analysis by hand, so we provide an automated tool, PALMS, to assist the analyst.

3.2 Evaluating Compliance

Policy compliance is important in a distributed system when labels are being communicated over sockets and an SELinux machine wants to be certain that the machine to which it is sending its data will be compliant with its own policy. For example, when machine A connects to machine B over a socket with MLS label $s_2$, will machine B honor the policy of machine A and not leak data passed through that socket to a lower level such as $s_1$?

Policy compliance may also be useful to evaluate applications running on a particular OS. If an application has access to multiple levels of data, administrators may want to know if the application’s flows obey a particular security lattice. Can we test compliance of the application flows against the host OS’s MLS policy?

In this section we use the semantics we developed for an SELinux MLS policy to formally define how to evaluate compliance of two policies. First, we give some general definitions of information flows and functions that operate on them, and then we give an algorithm that indicates how to instantiate these functions for an SELinux MLS policy.

3.2.1 Determining Information Flows

We use the next definitions to determine the information flows enabled by any given MLS policy.

**Definition 3.2.1 (Information Flow Policy).** A policy consists of a set of security levels arranged in a lattice with partial order $\subseteq$, and a set of statements determining each subject’s read/write permissions for a given object. The statement are based on the security levels of the subject and object (and possibly also on other factors such as the class of the object).

**Example 3.2.2 (Military MLS policy).** Consider a typical military MLS information flow policy with four security levels and no categories. Typically, military policies have
permissions that implement the simple security condition (ssc) and ⋆-property.

\[ \text{levels}(\text{Mil}) = \{\text{unclassified}(UC), \text{confidential}(CO), \text{secret}(S), \text{topsecret}(TS)\} \]

where: \( UC \subseteq CO \subseteq S \subseteq TS \), and reads and writes obey the simple security condition (ssc), and ⋆-property:

- **Simple security condition**: For a subject labeled \( l_s \) and an object labeled \( l_o \), the subject can read from the object iff \( l_o \subseteq l_s \).
- **⋆-property**: For a subject labeled \( l_s \) and an object labeled \( l_o \), the subject can write to the object iff \( l_s \subseteq l_o \).

**Definition 3.2.3** (Information flow). An *information flow* from \( l_1 \) to \( l_2 \) exists in a system when a single process can read from a resource labeled with \( l_1 \) and write to a resource labeled with \( l_2 \).

**Example 3.2.4.** For the military policy given in Example 3.2.2, there is an information flow \((UC, S)\), because for a subject at level \( CO \), there is a valid read of an object at level \( UC \) and a valid write of that object out to \( S \). (Note: there are also other ways to generate this information flow, with a subject at level \( UC \) or \( CO \), but not at \( TS \).)

**Definition 3.2.5** (AllFlows). The function \( \text{AllFlows} \) returns all information flows allowed in a given Policy with levels \( \text{levels}(\text{Policy}) \).

\[ \text{AllFlows} : \text{Policy} \to \wp(\text{levels}(\text{Policy}) \times \text{levels}(\text{Policy})) \]

To instantiate this function for the Mil policy, we must find all information flows, such that the ssc and the ⋆-property are preserved.

**Example 3.2.6** (AllFlows\(_{\text{Mil}}\)).

\[ \text{AllFlows}_{\text{Mil}} = \{(l_1, l_2) \mid l_1, l_2 \in \text{levels}(\text{Mil}) \land \exists l_s \in \text{levels}(\text{Mil}). l_1 \subseteq l_s \subseteq l_2\} \]

\[ \text{AllFlows}_{\text{Mil}} = \{(UC, UC), (UC, CO), (UC, S), (UC, TS), (CO, CO), (CO, S), (CO, TS), (S, S), (S, TS), (TS, TS)\} \]

### 3.2.2 Comparing policies

In a distributed system, it is important to know how the policies of two operating systems compare before they start exchanging labeled data.

When comparing two information flow policies, we require a mapping from the levels in one policy to the levels in the other. The mapping need not be defined for every
level, but must map the levels in policy A to a subset of the levels in Policy B. All levels that are not shared between policy A and policy B are mapped to $\bot$ (undefined). In the following, we define both the renaming of a single level and the renaming of a flow (overloading the name $\text{rename}$).

**Definition 3.2.7** (rename).

\[
\text{rename}_{A \rightarrow B} : \text{levels}(A) \rightarrow (\text{levels}(B) + \bot)
\]
\[
\text{rename}_{A \rightarrow B} : \text{levels}(A) \times \text{levels}(A) \rightarrow (\text{levels}(B) + \bot) \times (\text{levels}(B) + \bot)
\]

**Definition 3.2.8** (Shared levels). A level $l$ is said to be shared between two policies A and B iff $\text{rename}_{A \rightarrow B}(l) \neq \bot$.

Compliance can then be defined for two policies by comparing the flows allowed in one policy with the flows allowed in the other. Specifically, we are interested in the flows between levels shared by the two policies.

**Definition 3.2.9** (Compliance). An information flow policy A is said to be compliant with an information flow policy B, iff

\[
\text{Flows}'_A \subseteq \text{Flows}_B
\]

where

\[
\text{Flows}_A = \text{AllFlows}_A(A)
\]
\[
\text{Flows}_B = \text{AllFlows}_B(B)
\]
\[
\text{Flows}'_A = \text{rename}_{A \rightarrow B}(\text{Flows}_A)
\]

Although the definition of compliance implies that all flows in both policies should be determined, in order to determine whether the flows in policy A are a subset of policy B, only the flows of policy A need to be exhaustively determined. Then each flow allowed by A can be checked to see if it is also allowed in policy B. This can lead to some performance improvement if policy B is significantly larger than policy A (as in the case when B is an OS policy and A is only an application policy).

### 3.2.3 Information flows for SELinux MLS

When implementing these information flow functions for SELinux policy, we must make some adjustments. The first consideration is that SELinux policy parameterizes MLS.
access rules based on object class \((c)\), as described in Section 3.1.2. Thus, an information flow can occur using multiple classes, such as by reading from a public file and then writing to a secret \(ipc\). This requires us to define information flows by iterating over all possible object classes.

The second consideration is that the policy also parameterizes accesses based on the possible modes for that class. So, continuing the previous example, information could be read from a public file using the \(\text{getattr\_mode}\) and written to a secret \(ipc\) using the \(\text{open\_mode}\). We follow other systems \([40, 134]\) in grouping modes into ‘read-like’ and ‘write-like’ modes. Some modes fall into both categories, such as \(\text{dir\_create}\) which certainly is ‘write-like’, but is also ‘read-like’ because it will reveal whether the directory already existed. We extend our formal semantics to include the functions \(\text{readlike}(p)\) and \(\text{writelike}(p)\), they return true if the mode \(p\) is read-like or write-like, respectively.

We can instantiate the \(\text{AllFlows}\) algorithm with the SELinux MLS policy by using the constraint \(\gamma_{\text{MLS}}\) and operators \(\text{classes, modes, and ranges}\), from our formal semantics given in Section 3.1.3. The function is divided into two checks corresponding to two different ways that information flows can occur. The first way is by reading (in some mode) from some class at one level and writing (in some mode) to some class at another level. The second way is by simply relabeling an object from one level to another level.

Although we are not primarily concerned about general security contexts (including user, role and type) for our analysis of the MLS policy, \(\gamma_{\text{MLS}}\) does require that the full security context of the subject and object be provided. This is because, generally speaking, the subject might have some special privileges that affect the MLS constraints. For this analysis, we are concerned with the most basic scenario, so we fix our subject and object to have a vanilla type \(t\) with no extra privileges and to have insignificant user and role fields. For a more thorough analysis, our MLS analysis could be combined with existing analyses \([148, 58, 134, 40]\) that consider information flows introduced by type enforcement. The independence of TE policies from MLS policies, however, facilitates the approach we have taken. The only additional interaction that could be considered is when a type transition might move the subject into a state in which it has some additional MLS privileges. We leave the consideration of this fringe case to future work. Thus, the set of flows can be found by generating the union of the sets as follows.

**Algorithm 3.2.10** \((\text{AllFlows}_{\text{SELinux}})\).

\[
\text{AllFlows}_{\text{SELinux}}(\text{Policy}) = \\
\{(l_1, l_2) \mid \exists c_1, c_2 \in \text{classes}(\text{Policy}). \exists p_1, p_2 \in \text{modes}(\text{Policy}). \exists l_s \in \text{ranges}(\text{Policy}). \text{readlike}(p_1)\}
\]
Figure 3.1: Information flows enabled by SELinux vs. flows enabled by a Jif application. The Jif compiler guarantees that flows from p1 to p2 and from p2 to p3 are allowed only if p3 dominates p2 and p2 dominates p1 in a given Jif policy. The question is whether a flow from s1 to s2 is allowed by the correspondent SELinux policy.

\[ \land \text{writelike}(p_2) \land s = (u, r, t, l_s) \land o_1 = (\text{sys}, \text{obj}, t, l_1) \land o_2 = (\text{sys}, \text{obj}, t, l_2) \]
\[ \land \gamma_{\text{MLS}}(s, o_1, c_1, p_1) \land \gamma_{\text{MLS}}(s, o_2, c_2, p_2) \} \cup \]
\[ \{(l_1, l_2) \mid \exists c \in \text{classes(Policy)}. \exists l_s \in \text{ranges(Policy)}. s = (u, r, t, l_s) \land o_1 = (\text{sys}, \text{obj}, t, l_1) \]
\[ \land o_2 = (\text{sys}, \text{obj}, t, l_2) \land \gamma_{\text{MLS}}(s, o_1, c, \text{relabelfrom}) \land \gamma_{\text{MLS}}(s, o_2, c, \text{relabelto}) \]
\[ \land \gamma_{\text{MLSvt}}(o_1, o_2, s, c) \} \]

3.3 Jif-SELinux Compliance Problem

This section describes the Jif-SELinux compliance problem. While Jif enforces a security policy, the underlying SELinux operating system enforces an independently developed security policy. The relationship between these two policies is uncertain a priori. We want to automatically check whether a Jif policy and an SELinux policy are compliant, thus proving that the application enforces system security requirements.

3.3.1 SELinux and Jif Policies

Mandatory access controls implemented in SELinux allow the operating system to control the security characteristics and class of resources an application has access to. However, in several cases, applications require access to data with multiple security labels in order to work in a proper way. An email client, for example, needs access to all the security levels associated with a given user. A trusted system utility may need to operate on log files or configuration files associated with many different security levels. In general, servers, client software, and high integrity programs with low integrity inputs such as network-facing daemons may all require such privileges.

Since the operating system enforces policies over applications at the granularity of
inputs and outputs, it cannot trace how information is handled within the application domain. Therefore, applications allowed to access data with multiple security levels may leak that data contrary to system policy. Security-typed languages address this problem by making guarantees about the security policy that the application enforces.

Jif enforces a lattice policy where information in a variable with security label $\ell_1$ is allowed to flow to a variable with security label $\ell_2$, only if $\ell_2$ is equal to or dominates $\ell_1$ in the lattice. For example, in a military setting, information in $\ell_1$ may flow to $\ell_2$ only if $\ell_2$ is Top Secret and $\ell_1$ is Secret. The Jif compiler checks a given program and generates object code only if all the information flows enabled by the program meet the security requirements established in a Jif policy (i.e. data may never flow to unauthorized variables) [50, 47, 51].

Figure 3.1 illustrates the compliance problem. SELinux controls access to inputs and outputs, but within the program, such resources may be managed in multiple ways. In the example, the application accesses the resources $i$ and $j$. The Jif policy converts the OS levels $s_1$ to $p_1$ and $s_2$ to $p_3$. For its part, Jif guarantees that application flows are allowed only if $p_3$ dominates $p_2$, and $p_2$ dominates $p_1$, in the application lattice. In this case, this allows a flow from operating system resource $i$ to operating system resource $j$. Consequently, the application flow from $p_1$ to $p_3$ should only be allowed so long as the operating system flow from $i$ to $j$ is allowed. In other words, this application with policy $p_1 \sqsubseteq p_2$, $p_2 \sqsubseteq p_3$, and rename function $\{p_1 \mapsto s_1, p_3 \mapsto s_2\}$, should only be allowed to execute if the operating system policy allows flows from $s_1$ to $s_2$ [50, 47, 51].

### 3.3.2 SIESTA Implementation

Figure 3.2 presents the architecture of SIESTA. The architecture includes two main elements: an interface and a background daemon. The interface enables users to request the execution of an application. It also establishes communication with the daemon on behalf of the user. The daemon verifies that the requested application is secure enough according to the system requirements. The daemon runs the algorithms presented in the previous section to verify compliance. If the application is accepted, SIESTA starts it with special privileges (access to multiple levels is allowed), assuming that the operating system is also configured to accept such execution. If a user does not have such privilege, the operating system will prevent the execution of the application.

SIESTA evaluates applications and starts them only if they are compliant. When evaluating compliance SIESTA considers three elements: Jif signature, policy, and declasifiers. The Jif signature indicates that the program was developed in a security-typed
Figure 3.2: The SIESTA service includes two main components, interface and background daemon. The former receives requests and forwards them to the latter. The latter evaluates compliance, if the policies are compliant SIESTA starts it with the requested privileges.

language; it is a proof of an appropriate information flow management (no leakage). What to do in order to accept a signature as valid or not is a problem well known in multiple applications and orthogonal to our main problem. Currently, SIESTA generates the MD5 hash code of the requested application and compares that code against a list of previously accepted applications. If the MD5 code is in the list, the application is accepted as a Jif application. SIESTA may be modified to perform a different signature checking, thus adjusting to different system requirements. The application policy and the system policy must be compliant. This indicates that the application enforces the security requirements embodied by the OS policy; all the information flows enabled by the application are also enabled by the OS policy. The declassifiers used by the application must be a subset of the declassifiers allowed by the system; in the current implementation SIESTA expects administrators to predefine the list of valid declassifiers.

3.4 Discussion

We have presented an analytical model to represent SELinux MLS policies and reason about them. We implemented the model in XSB Prolog [27], and use this implementation to evaluate whether the current specification of the SELinux MLS policy meets the MLS security properties, namely, the simple security and ‘*’ properties. We also presented SIESTA, the Service for Inspection and Execution of Security-Typed Applications. SIESTA reads two policies, the operating system policy, and the MLS policy a
Jif program enforces, and evaluates whether the program policy is compliant with the operating system policy. A limitation the approach we presented in this Chapter is that although the compliance evaluation is an automated process, we require administrators to manually define a mapping between the policies and the security goals to evaluate.

SIESTA assumes that the Jif policy is defined in a separate file, not as part of the program. However, security-typed applications usually have the policies they enforce embedded into the code itself. Currently, we rely on the ideas presented in Trusted Declassification [44], and Channels [45]. They assert the advantages of having a high-level policy specification tied to the program. However, the proposal is recent, and has not been embraced by the community yet. Works such as SIF [23] and CIVITAS [25] use defined Jif operators and methods, that is, they do not specify a high-level policy. The absence of a high-level policy specification limits the scope of our work. An alternative is to develop a tool to automatically compute and store in a separate file the policy a Jif program enforces, by analyzing its source code. Since the Jif language has special constructors to embed policy into the source code, we can use source code analysis to look for those constructors. The main issue would be to define a mechanism to handle calls to the dynamic runtime library, as they enable developers to create dynamic principals. In any case, this is a task we left as future work.
Chapter 4

Chapter 3 presented the Service for Inspection and Execution of Security Typed Applications, SIESTA, which evaluates MLS compliance. In that work we manually specify mapping functions between a program’s policy and a system’s policy [50]. To enable automated compliance evaluation of general programs, we need approaches to automatically build compliance problems from available data.

This chapter presents an approach for automatically verifying that trusted programs correctly enforce system security goals when deployed. Currently, these programs are trusted without concrete justification, but the emergence of tools for building programs that guarantee policy enforcement, such as security-typed languages (STLs) and user-level mandatory access control systems, offers a basis for justifying trust in such programs. Since program and system policies are independently developed, compliance for program deployment is difficult to achieve in practice. We introduce PIDSI, a model we developed to infer the relationship between program and system policies, enabling automated compliance verification. We evaluated the approach on an SELinux system and found that it is consistent with the SELinux reference policy for trusted programs.

4.1 Background

As a basis for an automated approach, we observe that trusted programs and the system data upon which it operates have distinct security requirements. For a trusted program, we must ensure that the program’s components, such as its executable files, libraries,
configuration, etc., are protected from tampering by untrusted programs. For the system data, the system security policy should ensure that all operations on that data satisfy the system’s security goals. Since trusted programs should enforce the system’s security goals, their integrity must dominate the system data’s integrity. If the integrity of a trusted program is compromised, then all system data is at risk. Using the insight that Program Integrity Dominates System Integrity for trusted programs, we propose the PIDSI approach to designing program policies, where we assign trusted program objects to a higher integrity label than system data objects, resulting in a simplified program policy that enables automated compliance verification.

4.1.1 Program Deployment

We must consider how trusted programs are deployed on systems to determine what it takes to verify compliance. In Linux, programs are delivered in packages. A package is a set of files including the executable, libraries, configuration files, etc. A package provides new files that are specific to a program, but a program may also depend on files already installed in the system (e.g., system shared libraries, such as libc). Some packages may also export files that other packages depend on (e.g., special libraries and infrastructure files used by multiple programs).

For a trusted program, such as logrotate, we expect that a Linux package would include two additional, noteworthy files: (1) the program policy and (2) the SELinux policy module. The program policy is the file that contains the declarative access control policy to be enforced by the program’s reference monitor or STL implementation.

In SELinux, the system policy is the result of the integration of several policy modules; each SELinux policy module specifies the contribution of a particular package to the overall SELinux system policy [88]. Although SELinux policy modules are specific to programs, currently expert system administrators design them. Our logrotate program policy is derived from the program’s SELinux policy module, and we envision that program policies and system policy modules will be designed in a coordinated way (e.g., by program developers rather than system administrators) in the future, although this is an open issue.

An SELinux policy module consists of components that originate from three policy source files. First, a .te file defines a set of new SELinux types for this package. It also defines the policy rules that govern program accesses to its own resources as well as

---

1At present, module policies are not included in Linux packages, but RedHat, in particular, is interested in including SELinux module policies in its rpm packages in the future [152].

2SELinux uses the term type for its labels, as it uses an extended Type Enforcement policy [18].
system resources. Second, a .fc file specifies the assignment of package files to SELinux types. Some files may use types that are local to the policy module, but others may be assigned types defined previously (e.g., system types like etc.t is used for files in /etc). A .if file defines a set of interfaces that specify how other modules can access objects labeled with the types defined by this module.

When a package is installed, its files are downloaded onto the system and labeled based on the specification in the .fc file or the default system specification. Then, the trusted program’s module policy is integrated into the SELinux system policy\(^3\), enabling the trusted program to access system objects and other programs to access the trusted program’s files. There are two ways that another program can access this package’s files: (1) when a package file is labeled using an existing label or (2) when another module is loaded that uses this module’s interface or types. As both are possible for trusted programs, we must be concerned that the SELinux system policy may permit an untrusted program to modify a trusted program’s package file.

For example, the logrotate package includes files for its executable, configuration file, documentation, man pages, execution status, etc. Some of these files are assigned new SELinux types defined by the logrotate policy module, such as the executable (logrotate_exec.t) and its status file (logrotate_var_lib.t), whereas others are assigned existing SELinux types, such as its configuration file (etc.t). The logrotate policy module uses system interfaces to obtain access to the system data (e.g., logs), but no other processes access logrotate interfaces. As a result, logrotate is vulnerable to tampering because some untrusted processes might modify some of the system-labeled files that it uses.

We are also concerned that a logrotate process may be tampered by the system data that it uses (e.g., Biba read-down [15]). For example, logrotate may read logs that contain malicious data. We believe that systems and programs should provide mechanisms to protect themselves from the system data that they process. Some interesting approaches have been proposed to protect process integrity [136, 80], so we consider this an orthogonal problem that we do not explore further here.

### 4.1.2 Program Enforcement

To justify a system’s trust, any trusted program must enforce a policy that complies with the system’s security goals. The reference monitor concept [7] has been the guide for

---

\(^3\)As described above, this must be done manually now, via semodule, but the intent is that when you load a package containing a module policy, someone will install the module policy.
determining whether a system enforces its security goals, and we leverage this concept in defining compliance. A reference monitor requires three guarantees to be achieved: (1) complete mediation to ensure that all security-sensitive operations are authorized; (2) tamperproof guarantees that other programs cannot alter its behavior; and (3) verifiability to ensure correctness. While the reference monitor concept is most identified with operating system security, a trusted program must also satisfy these guarantees to ensure that a system’s security goal is enforced. As a result, we define that a program enforces a system’s security goals if it satisfies the reference monitor guarantees in its deployment on that system.

Our SIESTA service (described in section 3.3.2) compares program policies against SELinux system policies, and only executes programs whose policies permit information flows authorized in the system policy [47]. This work considered two of the reference monitor guarantees. First, we used the SIESTA service to verify trust in the Jif STL implementation of the `logrotate` program. Since the Jif compiler guarantees enforcement of the associated program policy, this version of `logrotate` provides complete mediation, modulo the Java Virtual Machine. Second, SIESTA performs a policy analysis to ensure that the program policy complies with system security goals (i.e., the SELinux MLS policy). Compliance was defined as requiring that the `logrotate` policy only authorized an operation if the SELinux MLS policy also permitted that operation. Thus, SIESTA is capable of verifying a program’s enforcement of system security goals.

We find two limitations to this work. First, we had to construct the program access control policy relating system and program objects in an ad hoc manner. As the resultant program policy specified the union of the system and program policy requirements, it was much more complex than we envisioned. Not only it is difficult to design a compliant program access control policy, but also that policy may only apply to a small number of target environments. Program policies should depend on system policies, particularly for trusted programs that we expect to enforce the system’s policy, making them non-portable unless we are careful. Second, this view of compliance does not protect the trusted program from tampering. As described above, untrusted programs could obtain access to the trusted program’s files after the package is installed, if the integrated SELinux system policy authorizes it. For example, if an untrusted program has write access to the `/etc` directory where configuration files are installed, SIESTA will not detect that such changes are possible.

In summary, Figure 4.1 shows that we aim to define an approach that ensures the following requirements:
Figure 4.1: Deployment and Installation of a trusted package. First, we check two compliance goals: (1) the system protects the application and (2) the application enforces system goals. Second the package is installed: the policy module is integrated into the system policy and application files are installed.

- For any system deployment of a trusted program, automatically construct a program policy that is compliant with the system security goals, thus satisfying the reference monitor guarantee of being simple enough to verify.

- For any system deployment of a trusted program, verify, in a mostly-automated way, that the system policy does not permit tampering of the trusted program by any untrusted program, thus satisfying the reference monitor guarantee of being tamperproof. The typical number of verification errors must be small and there must be a set of manageable resolutions to any such errors.

We aim to develop an approach to solve both of these requirements.

4.1.3 Policy Compliance

Verification of these two trusted program requirements results in the same conceptual problem, which we call policy compliance problems. Figure 4.2 shows these two problems. First, we must show that the program policy only authorizes operations permitted by the system’s security goal policy. While we have previously shown a method by which compliance can be tested [48, 47], the program policy was customized manually for the system. Second, we also find that the system policy must comply with the program’s tamperproof goals. That is, the system policy must not allow any operation that permits tampering the trusted program. As a result, we need to derive the tamperproof goals from the program (e.g., from the SELinux policy module).
Figure 4.2: Policy Compliance Problems for Trusted Applications: (1) verify that the program policy complies with the system’s information flow goals and (2) verify that the system policy, including the program contribution (e.g., SELinux policy module), enforces the tamperproofing goals of the program.

In section 2.2.4 we defined the formal model for verifying policy compliance suitable for both the problems above. We represent a security goal as a security lattice $L = (L, \sqsubseteq)$, where can flow to ($\sqsubseteq$) defines a partial order over the elements ($L$) of the lattice. We say that an information flow graph, $G_A = (V_A, E_A)$, is compliant with a security lattice, $L$, given a mapping function, $h$, that maps elements from the graph into elements in the lattice when, if there is a path between two vertices $u, v$ in $G_A$ ($u \rightarrow G_A v$), then $h(u)$ can flow to $h(v)$ in $L$ ($h(u) \sqsubseteq L h(v)$):

$\text{Given } h : V_A \rightarrow L, \text{ then } \forall u, v \in V_A . [(u \leftrightarrow G_A v) \rightarrow (h(u) \sqsubseteq L h(v))]$

However, as can be seen from Figure 4.2, the challenge is to develop system security goals, program policies, and tamperproof goals in a mostly-automated fashion that will encourage successful compliance. The PIDSI approach in Section 4.2 provides such guidance.

### 4.1.4 Difficulty of Compliance Testing

The main difficulty in compliance testing is in automatically constructing the program, system, and goal policies shown in Figure 4.2. Further, we prefer design constructions that will be likely to yield successful compliance.

The two particularly difficult cases are the program policies (upper left in the figure) and the tamperproof goal policy (lower right in the figure). The program policy and tamperproof goal policies require program requirements to be integrated with system requirements, whereas the system policy and system security goals are largely (although not necessarily completely) independent of the program policy.

First, it is necessary for program policies (upper left in the figure) to manage system
objects, but often program policy and system policy are written with disjoint label sets. Thus, some mapping from program labels to system labels is necessary to construct a system-aware program policy before the information flow goals encoded in $\mathcal{L}$ can be evaluated. Let $P$ be an information flow graph relating the program subjects and objects and $S$ be information flow graph relating the system subjects and objects. Let $P \oplus S$ be the policy that arises from combining $P$ and $S$ to form one information flow graph through some sound combination operator $\oplus$; that is, if there is a runtime flow in the policy $S$ where the program $P$ has been deployed, then there is a flow in the information flow graph $P \oplus S$. Currently, there are no automatic ways to combine such program and system graphs into a system-aware program policy, meaning that $\oplus$ is implemented in a manual fashion. A manual mapping was used in previous work on compliance [47].

Second, the tamperproof goal policy (lower right in the figure) derives from the program’s integrity requirements for its objects. Historically, such requirements are not explicitly specified, so it is unclear which program labels imply high integrity and which files should be assigned those high integrity labels. With the use of packages and program policy modules, the program files and labels are identified, but we still lack information about what defines tamperproofing for the program. Also, some program files may be created at installation time, rather than provided in packages, so the integrity of these files needs to be determined as well. We need a way to derive tamperproof goals automatically from packages and policy modules.

4.2 PIDSI

We propose the PIDSI approach (Program Integrity Dominates System Integrity), where the trusted program objects (i.e., package files and files labeled using the labels defined by the module policy) are labeled such that their integrity is relative to all system objects. The information flows between the system and the trusted program can then be inferred from this relationship. We manually studied the integrity relation between the elements of a set of trusted programs and the system objects that the program reads, we found that almost all trusted program objects are higher integrity than system objects (i.e., system data should not flow to trusted program objects). One exception that we have found is that log files may be written by multiple programs. However, a trusted program should not depend on the data in a log file. While general cases may be identified automatically as low integrity at present, we may have a small number of cases where the integrity level must be set manually.
Figure 4.3: Scheme of the mapping mechanism for the PIDSI approach. It relates program labels $G_A$ to system labels $G_B$, such that the program-defined objects are higher integrity than the system data objects (assigned to $H$), with some small number of low integrity exceptions (assigned to $L$).

Our approach takes advantage of a distinction between the protection of the trusted program and protection of the data to which it is applied. Trusted program packages contain the files necessary to execute the program, and the integrity of the program’s execution requires protection of these files. On the other hand, the program is typically applied to data whose protection requirements are defined by the system.

### 4.2.1 PIDSI Definition

By using the PIDSI approach between trusted program and the system, we can deploy that trusted program on different systems, ensuring compliance. Figure 4.3 demonstrates this approach. First, the program defines its own set of labels. For these labels, they are either designated as high or low integrity. When the program is deployed, the labels on that system’s data are placed in between the program’s high and low integrity labels.

In the event that the supposedly trusted program allows low integrity data to affect high integrity data, via an information flow directly from low to high, then this can trick the system into trusting low integrity data. To eliminate this possibility, we automatically verify that no such flows are present in the program policy.

In this context, the compliance problem requires checking that the system’s policy, when added to the program, does not allow any new illegal flows. We construct the system policy $G'_B$ from $G_A$ and $G_B$. First, split $G_A$ into subgraphs $H$ and $L$ as follows: if $u \in G_A$ is such that $\text{Integrity}(\text{Type}(u)) = \text{high}$, then $u \in H$, and if $u \in G_A$ is such that $\text{Integrity}(\text{Type}(u)) = \text{low}$, then $u \in L$. $G'_B$ contains copies of $G_B$, $H$, $L$, with edges from each vertex in $H$ to each vertex in $G_B$, and edges from each vertex in $G_B$ to $L$. The constructed system policy $G'_B$ corresponds to the deployment of the program policy $G_A$. 
Figure 4.4: logrotate instantiation for the two policy compliance problems: (1) the program policy is derived using the PIDS approach and the SELinux MLS policy forms the system’s information flow goals, and (2) the system policy is combined with the logrotate SELinux policy module and the tamperproofing goals are derived from the logrotate Linux package.

Theorem 4.2.1. Given test policy $G_A$ and target policy $G_B$, if for all $u \in H$, $v \in L$, there is no edge $(v, u) \in G_A$, then the test policy $G_A$ is compatible with the constructed system policy $G'_B$.

Given the construction, the only illegal flow that can exist in $G'_B$ is from a vertex $v \in L$, which has a low integrity label, to one of the vertices $u \in H$, which has a high integrity label. The graph $G_B$ is compliant with $G'_B$ by definition, and the edges that we add between subgraphs are from $H$ to $G_B$ and $G_B$ to $L$: these do not upgrade integrity [127].

4.2.2 PIDSI in practice

In this section, we describe how we use the PIDSI approach to construct the two policy compliance problems defined in Section 4.1.3 for SELinux trusted programs. Our proposed mechanism for checking compliance of a trusted program during system deployment was presented in Figure 4.1. We now give the specifics how this procedure would work during an installation of logrotate. Figure 4.4 shows how we construct both problems for logrotate on an SELinux/MLS system. For testing compliance against the system security goals, we use the PIDSI approach to construct the logrotate program policy and use the SELinux/MLS policy for the system security goals. For testing compliance against the tamperproof goals, we use the SELinux/MLS policy that includes the logrotate policy module for the system policy and we construct the tamperproof goal policy from the logrotate package. We use logrotate on SELinux/MLS as an example to explain why these constructions are satisfactory for deploying trusted programs.
For system security goal compliance, we must show that the program policy only permits information flows in the system security goal policy. We use the PIDS approach to construct the program policy as described above. For the Jif version of logrotate, this entails collecting the types (labels) from its SELinux policy module, and composing a Jif policy lattice where these Jif versions of these labels are higher integrity (and lower secrecy) than the system labels. Rather than adding each system label to the program policy, we use a single label as a template to represent all of the SELinux/MLS labels \cite{47}. We use the SELinux/MLS policy for the security goal policy. This policy clearly represents the requirements of the system, and logrotate adds no additional system requirements. While some trusted programs might embody additional requirements that the system must uphold (e.g., for individual users), this is not the case for logrotate. As a result, to verify compliance we must show that there are no information flows in the program policy from system labels to program labels, a problem addressed by previous work \cite{47}.

For tamperproof goal compliance, we must show that the system policy only permits information flows that are authorized in the tamperproof goal policy. The system policy includes the logrotate policy module, as the combination defines the system information flows that impact the trusted program. The tamperproof policy is generated from the logrotate package and its SELinux policy module. The logrotate package identifies the labels of files used in the logrotate program. In addition to these labels, any new labels defined by the logrotate policy module, excepting process labels that are protected differently as described in Section 4.1.1, are also added to the tamperproof policy. The idea is that untrusted programs may not modify these labels. That is, untrusted process labels may not have any kind of write permission to the logrotate labels. Unlike security goal compliance, the practicality of tamperproof compliance is clear. It may be that system policies permit many subjects to modify program objects, thus making it impossible to achieve such compliance. Also, it may be difficult to correctly derive tamperproof goal policies automatically. In Section 4.3, we show precisely how we construct tamperproof policies and test compliance, and examine whether tamperproof compliance, as we have defined it here, is likely to be satisfied in practice.

4.3 Verifying Compliance in SELinux

In this section, we evaluate the PIDS approach against actual trusted programs in the SELinux/MLS system. As we discussed in Section 4.2.2, we want to determine
whether it is possible to automatically determine tamperproof goal policies and whether systems are likely to comply with such policies. First, we define a method for generating tamperproof goal policies automatically and show how compliance is evaluated for the `logrotate` program. Then, we examine whether eight other SELinux trusted programs satisfy tamperproof compliance as well. This group of programs was selected because: (1) they are considered MLS-trusted in SELinux and (2) they have Linux packages and SELinux policy modules. Our evaluation finds that there are only 3 classes of exceptions that result from our compliance checking for all of these evaluated packages. We identify straightforward resolutions for each of these exceptions. As a result, we find that the PIDS approach appears promising for trusted programs in practice.

4.3.1 Tamperproof Compliance

To show how tamperproof compliance can be checked, we develop a method in detail for the `logrotate` program on a Linux 2.6 system with a SELinux/MLS strict reference policy. To implement compliance checking with the tamperproof goals, we construct representations of the system (SELinux/MLS) policy and the program’s tamperproof goal policy. Recall from Section 4.1.3 that all the information flows in the system policy must be authorized by the tamperproof goal policy for the policy to comply.

4.3.1.1 Build the Tamperproof Goal Policy

To build the tamperproof goal policy, we build an information flow graph that relates the program labels to system labels according to the PIDS approach. Building this graph consists of the following steps:

1. Find the high integrity program labels.
2. Identify the trusted system subjects.
3. Add information flow edges between the program labels, trusted subject labels, and remaining (untrusted) SELinux/MLS labels authorized by the PIDS approach.

*Find the high integrity program labels.* This step entails collecting all the labels associated with the program’s files, as these will all be high integrity per the PIDS approach. These labels are a union of the package file labels determined by the file contexts (.fc file in the SELinux policy module and the system file context) and the newly defined labels in the policy module itself. First, the `logrotate` package includes
the files indicated in Table 4.1. Second, some program files may be generated after the package is installed. These will be assigned new labels defined in the program policy module. An example of a \texttt{logrotate} label that will be assigned to a file that is not included in the package is \texttt{logrotate_lock_t}. In Section 4.4, we discuss other system files that a trusted program may depend upon.

\textit{Identify trusted subjects.} Trusted subjects are SELinux subjects that are entrusted with write permissions to trusted programs. Based on our experience in analyzing SELinux/MLS, we identify the following seven trusted subjects: \texttt{dpkg\_script\_t, dpkg\_t, portage\_t, rpm\_script\_t, rpm\_t, sysadm\_t and prelink\_t}. These labels represent package managers and system administrators; package managers and system administrators must be authorized to modify trusted programs. Programs other than \texttt{logrotate} also trust these subjects. We would want to control what code is permitted to run as these labels, but that is outside the scope of our current controls.

\textit{Add information flow edges.} This step involves adding edges between vertices (labels) in the tamperproof goal information flow graph based on the PIDSI approach. The PIDSI approach allows program labels to read and write each other, but the only SELinux/MLS labels that may write program labels are the trusted subjects (and read as well). Other SELinux labels are restricted to reading the program labels only. Figure 4.5 presents an example of a tamperproof goal policy’s information flow graph. Notice that only the system trusted labels (dotted circles) are allowed to write to program labels (solid line circles). Because the application has high integrity requirements for \texttt{etc\_t}, the graph includes edges that represent these requirements. The same set of edges will be also added for the other program labels (presented to the right in the figure).

\subsection{Build the System Policy}

The system policy is represented as an information flow graph (as defined in Section 2.2.1). Building this graph consists of the following steps:

1. Create an information flow graph that represents the current SELinux/MLS policy.

2. Add \texttt{logrotate} program’s information flow vertices and edges based on its SELinux policy module.

3. Remove edges where neither vertex is in the tamperproof goal policy.
Create an information flow graph. We convert the current SELinux/MLS policy into an information flow graph. Each of the labels in the SELinux/MLS policies is converted to a vertex. We create information flow edges by identifying read-like and write-like permissions [40, 134] between labels. The following example illustrates the process we follow to create a small part of the graph. Rules 1-3 and 6 are system rules, rules 4-5 are module rules (defined in the logrotate policy module).

1. allow init_t init_var_run_t:file {create getattr read append write setattr unlink};
2. allow init_t bin_t:file {read getattr lock execute ioctl execute_no_trans};
3. allow init_t etc_t:file {read getattr lock ioctl};
4. allow logrotate_t etc_t:file {read getattr lock ioctl};
5. allow logrotate_t bin_t:file {read getattr lock execute ioctl execute_no_trans};
6. allow chfn_t etc_t:file {create ioctl read getattr write setattr append link unlink rename};

Figure 4.6 shows the result of parsing the previous rules. In this example, subjects with type init.t are allowed to read from and write to init_var_run.t, and logrotate.t is allowed to read from etc.t and bin.t.

Also, note that Figure 4.6 shows that chfn.t has write access to etc.t and logrotate.t can read it. While logrotate cannot write any file with the label etc.t, it provides such a file via its package installation, so it depends on the integrity of files with that label. This exception will emerge in our compliance analysis below.

We parse the text version of an SELinux policy (file policy.conf) with a C program integrated with Flex and Bison. It is also possible to analyze the binary version of the
Figure 4.6: Information flow graph for the system policy, including the logrotate program’s policy module. chfn_t is not trusted to modify other trusted programs, but it has write access to logrotate’s files labeled etc_t.

<table>
<thead>
<tr>
<th>File</th>
<th>SELinux Type</th>
<th>Policy</th>
<th>Writers</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>/etc/logrotate.conf</td>
<td>etc_t</td>
<td>system</td>
<td>18</td>
<td>integrity</td>
</tr>
<tr>
<td>/etc/logrotate.d</td>
<td>etc_t</td>
<td>system</td>
<td>18</td>
<td>integrity</td>
</tr>
<tr>
<td>/usr/sbin/logrotate</td>
<td>logrotate_exec_t</td>
<td>module</td>
<td>8</td>
<td>no</td>
</tr>
<tr>
<td>/usr/share/doc/logrotate/CHANGES</td>
<td>usr_t</td>
<td>system</td>
<td>7</td>
<td>no</td>
</tr>
<tr>
<td>/usr/share/man/logrotate.gz</td>
<td>man_t</td>
<td>system</td>
<td>8</td>
<td>no</td>
</tr>
<tr>
<td>/var/lib/logrotate.status</td>
<td>logrotate_var_lib_t</td>
<td>module</td>
<td>8</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 4.1: logrotate Compliance Test Case and Results: there are two exceptions, but they originate from the same system label etc_t.

SELinux system policy.

Add logrotate program’s information flows. In a similar fashion to the method above, we extend the information flow graph with the vertices (labels) and edges (read and write flows) from the logrotate policy module.

Remove edges where neither vertex is in the tamperproof goal policy. As these flows cannot tamper the logrotate program, we remove these edges from the system policy for compliance testing.

4.3.2 Evaluating logrotate

This section presents how we automatically test tamperproof compliance. Tamperproof compliance is based checking the system policy for information flow integrity as defined by the tamperproof goal policy.

Integrity Compliance Checking. To detect integrity violations, we identify information flows that violate the Biba integrity requirement [15]: an information flow from a low integrity label (type in SELinux) to a high integrity label. read and write arguments
are subject and object.

\[
\text{NonBibaFlows}_{\text{SELinux}}(\text{Policy}) = \{(t_1, t_2) : t_1, t_2 \in \text{types}(\text{Policy}). \text{highintegrity}(t_1) \land \\
\text{lowintegrity}(t_2) \land (\text{read}(t_1, t_2) \lor \text{write}(t_2, t_1))\}
\]

We use the XSB Prolog engine [27] as the underlying platform. We developed a set of prolog queries based on the NonBiba Flows rule to detect the labels that affect compliance (i.e., the high integrity requirement that are not enforced by the system policy).

As mentioned in the previous section, we evaluate tamperproof compliance at installation time. We load the policy graphs generated above into the Prolog engine and we run the integrity Prolog queries to determine if any flows satisfy (negatively) the NonBiba Flows, thus violating compliance.

\textbf{Results.} Table 4.1 presents the list of files in the logrotate package, the label assigned to each, whether such label is a program label (i.e., defined by the program’s policy module) or a system label, and the results for compliance checking logrotate against the generated tamperproof goal policy (see column 4).

Only etc.t has unauthorized writers. In the SELinux/MLS reference policy, these writers are programs with legitimate reasons to write to files in the /etc directory, but none have legitimate reasons to write to logrotate files. For example, chfn, groupdadd, passwd, and useradd are programs that modify system files that store user information in /etc, kudzu is an program that detects and configures new and/or changed hardware in a system and requires to update its database stored in /etc/sysconfig/hwconf, and updfstab is designed to keep /etc/fstab consistent with the devices plugged in the system.

The obvious solution would be to refine the labels for files in /etc to eliminate these kinds of unnecessary and potentially risky operations.

\subsection{4.3.3 Evaluating other Trusted Programs}

Table 4.2 shows a summary of the results from applying the PIDSI approach to eight SELinux trusted programs for which policy modules and packages are defined. The table shows: (1) trusted package, (2) file labels (SELinux types) used per package, (3) number of writers detected per type (Writers) and (4) exceptions. The integrity requirement assigned by default is high integrity for all types, except for the ones marked with **.
<table>
<thead>
<tr>
<th>Package</th>
<th>SELinux Label</th>
<th>Writers</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>cups</td>
<td>initrc_exec_t</td>
<td>8</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>textrel_shlib_t</td>
<td>9</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>lpr_exec_t</td>
<td>8</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>dbusd/etc_t</td>
<td>7</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>system types</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td></td>
<td>var_log_t**</td>
<td>14</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>var_run_t**</td>
<td>10</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>var_spool_t</td>
<td>10</td>
<td>no</td>
</tr>
<tr>
<td>dmidecode</td>
<td>dmidecode_exec_t</td>
<td>8</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>system types</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>hald</td>
<td>locale_t</td>
<td>7</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>initrc_exec_t</td>
<td>8</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>hald_exec_t</td>
<td>8</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>dbusd/etc_t</td>
<td>7</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>system types</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>iptables</td>
<td>iptables_exec_t</td>
<td>8</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>initrc_exec_t</td>
<td>8</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>system types</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>kudzu</td>
<td>locale_t</td>
<td>7</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>initrc_exec_t</td>
<td>8</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>system types</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>Network Manager</td>
<td>NetworkManager_var_run_t</td>
<td>8</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>NetworkManager_exec_t</td>
<td>8</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>dbusd/etc_t</td>
<td>7</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>system types</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td>rpm</td>
<td>rpm_exec_t</td>
<td>8</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>rpm_var_lib_t</td>
<td>7</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>system types</td>
<td>†</td>
<td>†</td>
</tr>
<tr>
<td></td>
<td>var_spool_t</td>
<td>10</td>
<td>no</td>
</tr>
<tr>
<td>sshd</td>
<td>sshd_exec_t</td>
<td>8</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>sshd_var_run_t</td>
<td>8</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>ssh_keygen_exec_t</td>
<td>8</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>ssh_keysign_exec_t</td>
<td>8</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 4.2: Results of applying the PIDS approach to SELinux Trusted Packages. Columns with a ‘†’ are displayed in Table 4.3.

Because of the semantics associated to `/var`, various applications write to this directory, we assign low integrity requirement to `var_log_t` and `var_run_t`.

The common system types (`bin_t`, `etc_t`, `lib_t`, `man_t`, `sbin_t` and `usr_t`) are marked with † in the last two columns. The results for these types are displayed in Table 4.3. The results show only two exceptions, none in Table 4.2 and two in Table 4.3.

We present the cause of the conflicts and suggest methods to resolve them in Table 4.4. One resolution method is to refine policies: programs should have particular labels for
Table 4.3: System labels referenced by the packages presented in Table 4.2. Only etc_t and man_t have conflicts; the number of conflicting types per case cannot be high (Writers column is an upper limit since it includes trusted writers), so we can precisely examine each exception and suggest resolutions (shown in Table 4.4).

<table>
<thead>
<tr>
<th>SELinux Label</th>
<th>Writers</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin_t</td>
<td>9</td>
<td>no</td>
</tr>
<tr>
<td>etc_t</td>
<td>18</td>
<td>integrity</td>
</tr>
<tr>
<td>lib_t</td>
<td>8</td>
<td>no</td>
</tr>
<tr>
<td>man_t</td>
<td>8</td>
<td>integrity</td>
</tr>
<tr>
<td>sbin_t</td>
<td>8</td>
<td>no</td>
</tr>
<tr>
<td>usr_t</td>
<td>7</td>
<td>no</td>
</tr>
</tbody>
</table>

their files, even if they are installed in system directories, instead of using general system labels. The use of a general system label gives all system programs access to these files (case APP LABELS in Table 4.4).

Policy refinement is not always possible, as sometimes a program actually requires access to system files. Thus we have another resolution method: to classify the program as trusted (case ADD in Table 4.4). For example, some trusted programs read information from the /etc/passwd file, so those subjects permitted to modify that file must be trusted. Only a small number of such programs must be trusted.

Finally, some programs are given permissions that they do not require. In these cases the resolution method is to remove conflicting types/permissions from the application policy (case REMOVE in Table 4.4).

4.4 Discussion

Trusted programs may use system files, such as system libraries or the password file, in addition to the files provided in their packages. Because some of our trusted program packages install their own libraries under the system label lib_t our analysis included system libraries. Therefore, application integrity not only depends on the integrity of the files in the installation package but also on some other files. In general, the files that the program execution depends on should be comprehensively identified. These should be well known per system.

An issue is whether a trusted program may create a file whose label is a system label. For example, a trusted program generates the password file, but this file is used by the system, so it has a system label. We did not see this case in our testing, but it is possible in practice. We believe that more information about the integrity of the contents
Table 4.4: **Compliance Exceptions and Resolutions.** This table details the exceptions to tamperproof compliance presented in Table 4.3. It shows the list of conflicting, untrusted subjects and the resolution method, per case.

generated by the program will need to be used in compliance testing. For example, if the program generates data it marks as high integrity, then we could leverage this in addition to package files and program policy labels to generate tamperproof goal policies.

The approach we describe in this chapter applies only to trusted programs. We make no assumptions about the relationship between untrusted program and the system data. In fact, we are certain that there is system data that should not be accessed by most, if not all, untrusted programs. Note that there is no advantage to verifying the compliance of untrusted programs, because the system does not depend on untrusted programs to enforce its security goals. Such programs have no special authority.

This work is driven by the idea of unifying application and system security policies. Since applications and systems policies are independently developed, they use different language syntax and semantics. As a consequence, it is difficult to prove or disprove that programs enforce system security goals. The emergence of mandatory access control systems and security typed languages makes it possible to automatically evaluate whether applications and systems enforce common security goals. We reshape this problem as a verification problem: we want to evaluate if applications are *compliant* with system policies.
We found that compliance verification involves two tasks: we must ensure that the system protects application from being tampered with, as well as verify that the application enforces system security goals. In order to automate the mapping between the program policy and the system policy, we proposed the PIDSI (Program Integrity Dominates System Data Integrity) approach. The PIDSI approach relies on the observation that in general program objects are higher integrity than system objects. We tested the trusted program core of the SELinux system to see if its policy was compatible with the PIDSI approach. We found that our approach accurately represents the SELinux security design with a few minor exceptions, and requires little feedback from administrators in order to work.
Chapter 5

Compliant Configuration for Virtual Machines

Chapters 3 and 4 presented SIESTA and PIDSI respectively, two services we developed to support compliance evaluation. They both assist administrators in building compliance problems for environments with two policies: operating system (OS) and application.

This chapter explores the construction of compliance services for end-to-end systems, systems with multiple hosts where each host has a multilayer stack software. Our target environments are virtual machine systems (VM systems) with support for distributed applications, although we focus on web applications to be concrete.

The goal is to use VMs as flexible controlled domains and define access control to be compliant with system’s security goals. We aim to achieve this goal by identifying the VM operations that must be mediated and developing a supporting service that acts as a security server; it answers access control requests started by the software layer that mediates the operations. Additionally, we generate policy at run time, according to predefined security goals, that governs the behavior of new VMs in the system.

5.1 Introduction

Virtual machine systems have been promoted recently as an effective basis for building secure systems. Virtual machine systems enable the execution of commodity operating systems and applications within an isolated environment, thus protecting them from other code running on the same platform, and vice versa. For distributed applications, a virtual machine system still needs to provide the ability for VMs to share data and
communicate via untrusted networks. For example, commercial virtual machine systems do not provide control over access to network resources, so while a VM may be isolated from others on that platform, unauthorized communications may result in the leakage of information or integrity violations. Virtual machine systems are being extended with reference monitoring to control inter-VM communications [26, 132], but approaches to leverage such controls have yet to be developed.

Virtual machine systems that provide an architecture for enforcement are not very common. NetTop [94] and Solaris Containers [77] are products that use virtualization technologies to enforce MLS policies. These approaches benefit from the assumption of well-defined MLS policies across all platforms, so when a VM is launched, it is clear in a global sense what its permissions are. Some distributed applications are not so clear-cut. Our goal is to develop a virtual machine architecture whereby VMs may be used as flexible, controlled domains, unlike the MLS VM systems, and where the controls are defined to be consistent with system’s security goals (compliant), unlike Tahoma. The insight is that it should be possible to configure VM policies necessary to ensure that the new VM’s operations satisfy the system’s security goals. In this chapter, we identify the VM operations that require control, how to configure policy, and the supporting services we need to implement.

5.2 Problem Definition

Figure 5.1 shows a canonical distributed application. This is a client-server application where information may flow from the client to the server (e.g., web forms, document uploads), from the server to the client (e.g., information queries, media downloads), or both. Either information flow may be risky to the client (or server) because the client (or server) may leak information that she did not intend or she may receive information that may compromise the integrity of the client (or server) system. Example applications that fit this scenario include grid applications, where the results of the computation must be high integrity and may need to be secret, and web applications, which may be able to transfer secret data or may possibly include low integrity data.

5.2.1 Web Applications

In this research work, our focus is on web applications. Originally, web applications were simple client-server applications in which the client downloads content from a server, but web applications are now complex, multi-party computations where content may
be composed based on client inputs from multiple sources. Web mashups compose web pages from content, including executable code, that originate from multiple sources. As a result, a web application may involve downloading data from multiple servers. Instead of the traditional client-server application shown in Figure 5.1, such web mashups have a client interacting with multiple servers, each in a different administrative domain potentially.

Further, clients may use such web applications to upload data to any one of those servers. This presents both secrecy and integrity issues. Clearly, secret information may be passed among users, but a variety of information also comes from public, so-called ‘open sources’ \(^1\), so inadvertent and malicious leakage must be prevented. Also, integrity protection is an issue, as web mashups that interact with some untrusted servers may compromise the integrity of computations with trusted servers.

Previous security architectures for web applications derived security requirements from the server’s perspective, but the web applications above indicate that the client also has security requirements that must be enforced. The Tahoma security architecture isolates web applications into individual VMs that can only access URLs specified by the server [29]. As multiple servers may be involved in a web application, some determined by third parties, to enable advertisement for instance, it is neither practical nor effective for a server to define the security policy over a client’s interaction with all servers. The Swift architecture protects server data from unauthorized access by clients while automatically decomposing applications into client and server components to improve performance [21]. If the client has secret information that it wants to protect from the server or believes that the server is providing low integrity content, we cannot express such requirements in the Swift system.

\(^1\)Freely available information, which is analogous to open source software.
5.2.2 Layered Architectures

Virtual machine monitors and applications are now capable of enforcing mandatory access controls, in addition to operating systems. We argue that each layer will play an important role in enforcing system security goals. Virtual machine monitor security architectures, such as Xen sHype [132] and XSM [26], enable flexible, controlled interaction among VMs. Using such approaches, we can configure web applications into their own VMs and can control which servers each VM can communicate with. This is called the separation kernel approach [131], and is the basis for several approaches such as Tahoma [29] and NetTop [94]. The problem is that not all interactions within a web application are the same from a security perspective; a web application may involve data with different security requirements, therefore, VM separation would not be enough, access control within the VM is also necessary.

Within the VM, operating systems that implement mandatory access controls, such as SELinux [111], and policy enforcing applications, such as those based on security-typed languages [103, 138] or the ones that use application-level reference monitors [153, 91, 116], can enforce mandatory access control policies. For web applications, we believe that enforcement both at the OS and application level, and even over network communication channels, will be necessary. We find that some security functions are more effectively handled by applications, for instance integrity protection [124].

The problem is that having multiple layers of enforcement means that the MAC enforcement may be inconsistently enforced in some levels. The administrator must ensure that MAC enforcement requirements are supported by the VMM, VM's operating system, and applications, where necessary. Since we do not want to put the burden of examining multiple policies on administrators, our approach must provide automatic mechanisms to convey MAC policy consistently among these layers.

As a basis for defining consistent MAC policies, all MAC policies must comply with the host system’s security goals. Unlike the Tahoma system, where each VM is an application component that does not interact with the host system, we envision that our distributed applications may leverage some host system resources. Thus, we cannot use the Tahoma approach where policies are defined by the web server, but rather the host system’s administrative domain must govern the accesses available to each VM. In this work, we develop a virtual machine architecture that ensures that the host’s MAC requirements are enforced across all layers (VMM, OS, and application) in a VM system; it provides end-to-end enforcement.
5.3 End-to-End Enforcement

We claim that a system provides end-to-end enforcement for an application if a mandatory access control (MAC) policy is enforced consistently across all software layers. Anderson defined the reference monitor concept [7], which states the guarantees that must be satisfied to enforce a MAC policy correctly. We propose the construction of a multi-layer reference monitor for end-to-end enforcement. Table 5.1 shows the system layers (in rows), the MAC policy concepts (in columns), and the requirement assignment of MAC policy to layers (in each cell). Our task is to define a multi-layer reference monitor that enforces a coherent system-wide MAC policy and demonstrate what is necessary to build it correctly (Section 5.3.1). We also define a mandatory protection system, which motivates why a MAC policy is necessary for our multi-layer reference monitor and identifies the MAC policy concepts that must be enforced (Section 5.3.2).

5.3.1 Multi-Layer Reference Monitor

Table 5.1 shows four layers needed to enforce a MAC policy: (1) the application layer controls access to application objects (e.g., browser tabs and URLs); (2) the VM layer consists of the operating system that controls process (including the application) access to VM system objects (e.g., files and sockets); (3) the VMM layer (e.g., for Xen, its hypervisor and privileged host VM) controls inter-VM interactions (e.g., shared memory and communications); and (4) the network layer that authorizes communication and dictates how secure communication is performed (e.g., chooses cryptographic protocols).

We state that there may be multiple components at each layer that are required to enforce a MAC policy, such as multiple application processes or multiple VMs. Each of the components that we depend upon to enforce a MAC policy must be part of the multi-layered reference monitor for that application.

A reference monitor must satisfy the requirements embodied in the reference monitor concept [7]: (1) complete mediation—all security-sensitive operations must be mediated by the reference monitor; (2) tamper protection—the reference monitor must be protected from illicit modification; and (3) verifiability—the reference monitor mechanisms and policies must be verified to enforce site security goals correctly. A multi-layer reference monitor must ensure that the composition of layers satisfies these requirements. Below, we examine these requirements and the tasks that must be performed to satisfy them in a layered environment.

We leverage existing mediation in OSs (e.g., SELinux [111]) and VMMs (e.g., Xen
shype [132]), but we also require mediation for other layers and an approach for inter-layer communication of security information. For the application layer, we extend the browser application with mediation mechanisms that leverage OS labels. Although we focus on web applications here, the same principles apply for any distributed application running in the same environment. We use labeled IPsec [61] to authorize access at the network layer using system labels that specifies secure communication requirements. Additionally, we use the same set of system labels for all layers for consistence across layers.

Tamper protection is generally done by the layer below. For example, an SELinux policy must protect the browser process from tampering by other processes if it is to enforce its policy correctly. In prior work, we evaluated SELinux policies to ensure that reference monitoring processes cannot be tampered with by untrusted processes [127]. As the browser application is just another instance of a reference monitoring application, the same technique can be used, so we do not discuss this further. Ensuring that a VMM policy protects VMs from tampering can be performed similarly.

For the verifiability requirement, it is necessary to show that the multi-layer reference monitor implementation enforces the intended MAC policy. We claim that a monolithic MAC policy is enforced by a multi-layer reference monitor if: (1) each component in the multi-layer reference monitor meets the guarantees for complete mediation and tamper protection, accounting for trust in its environment; (2) each policy decision by a component in the multi-layer reference monitor is the same decision as that would have been made by a monolithic reference monitor using that policy. First, we note that a component in a multi-layered reference monitor may only be entrusted with a subset of the MAC policy, depending on its environment and its level of assurance. For example, a conventional OS may not be trusted to protect itself from untrusted processes, so it

<table>
<thead>
<tr>
<th>Layer</th>
<th>Protection State</th>
<th>Labeling State</th>
<th>Transition State</th>
<th>Enforcement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>DLM restrictions</td>
<td>Data</td>
<td>Low to high secrecy</td>
<td>Control data leakage</td>
</tr>
<tr>
<td>VM</td>
<td>SELinux policy</td>
<td>Processes</td>
<td>Within VM label range</td>
<td>Prohibit loading applications with illegal security levels</td>
</tr>
<tr>
<td>VMM</td>
<td>XSM policy</td>
<td>VMs</td>
<td>Spawning VMs</td>
<td>VM ranges limited to policy</td>
</tr>
<tr>
<td>Network</td>
<td>IPsec-policy</td>
<td>Sockets</td>
<td>Single level</td>
<td>Prevents connection with insecure remote machine</td>
</tr>
</tbody>
</table>

Table 5.1: Elements of the mandatory protection system needed at multiple layers for end-to-end secure distributed systems.
must not be given access to high integrity data. Second, each component in the multi-layer reference monitor must be given a policy specification that enables it to make the same decision as the monolithic reference monitor on decisions it is trusted to make. We aim to achieve this claim by construction. To do so, we need to precisely define our policy model and how policy specifications can be distributed while enforcing the same semantics as the monolithic case.

5.3.2 Mandatory Protection Systems

The four columns of Table 5.1 indicate the three policy concepts that must be supported by each layer (protection, labeling, and transition state) and types of enforcement decisions that the MAC policy enables (Enforcement). An access control policy defines whether a particular subject (e.g., process) can perform an operation on a particular object (e.g., file and URL) based on a fixed set of security labels. In traditional discretionary access control models, access control is specified in terms of dynamically-created entities, such as files, and managed by processes. The dynamic nature of this model makes it intractable to determine whether a process may obtain an unauthorized permission (problem known as the undecidability of the safety problem [43]). As a result, security-critical systems use MAC, where the system (e.g., an administrator and/or trusted system service) defines a fixed policy using an immutable set of security labels. While the security policy is fixed, new processes and files are still created dynamically, so MAC policy includes concepts to maintain consistency between the dynamically-evolving system and the static access policy. Ultimately, a service that configures a MAC policy
in a multi-layer reference monitor must be able to interpret these MAC policy concepts to deploy MAC policy across all layers to enforce end-to-end security correctly.

Our MAC policy is defined using a model, called a mandatory protection system (MPS) for historical reasons. An MPS is derived from the classical protection system model of Lampson. Figure 5.2 shows an MPS. It consists of: (1) a protection state (e.g., an access matrix) that defines the operations that subjects can use to access objects (i.e., the access control policy); (2) a labeling state that defines the mapping between system objects (including subjects) and their MAC labels; and (3) a transition state that defines how the assignment of a MAC label to an object may be changed. A classical protection system also defines a set of protection state operations that may be used to change from one state to another, but as a MAC system only allows a trusted entity to modify policy no such specification for protection state operations is necessary.

An MPS protection state is the same as a traditional protection state, except that subjects and objects are defined in terms of a set of labels which are immutable. The protection state then defines the operations that subject labels can perform on object labels, rather than subjects and objects directly.

Since new objects may be created and must be assigned a label, we need a labeling state to define the rules for mapping a new object (e.g., process or file) to its label. When newfile is created, it must be assigned one of the object labels in the protection state. In Figure 5.2, it is assigned the secret label. MAC models must include labeling state. For Bell-LaPadula models, the labeling state is implicit, as the labels of new objects inherit the label of the creating process (although writeup is possible). For SELinux, policy rules define exceptional labeling requirements, such as labeling a file based on the directory in which it is created.

The MPS also defines when an object’s label is changed via a transition state. A transition state changes label of a process, thereby altering the its protection domain which defines the permissions it can access. For example, SELinux defines rules that change the label of a process on exec. This mechanism enables the appropriate permissions to be assigned when a program is run (e.g., to control setuid). The transition state also describes label transitions on objects.

Our challenge is to develop an architecture where we can distribute MAC policy in terms of these concepts among the components in each layer, such that the original,
monolithic policy is enforced correctly. From Table 5.1, we know that the protection state associated to the VM layer is an SELinux policy specifically designed to run a given application securely. If the goal were only to implement a single application, a manual policy configuration could suffice, but for a general purpose system, we need automated configuration. Therefore, we propose an architecture that distributes MAC policy to the multi-layer reference monitor, and provides supporting services to ensure consistency.

5.4 Architecture

Here we provide an architecture for a multi-layered MAC system that provides end-to-end enforcement for distributed applications, although we focus here on web applications.

5.4.1 Components

Figure 5.3: A Multi-layer MAC Architecture enforces a single MAC application policy across multiple reference monitor components. Upon starting a new Browser VM, (1) the Application Authority sends the application’s policies to the VM Loader, which then (2) installs the policy into the VM and VMM, networking and FlowwolF reference monitors. (3) The FlowwolF retrieves URL labels from the Label Mapper as needed and communicates securely with the application’s server.

Our system, illustrated in Figure 5.3, consists of an Application Authority, which stores monolithic mandatory policies for one or more web applications, and a Client VMM, which loads application VMs that it runs under the MAC policy. The Client VMM has a VM Loader that distributes a monolithic policy to the appropriate layer’s reference monitors, including its own. Figure 5.3 shows a Browser VM that hosts a label-enforcing browser, called a FlowwolF. The Browser VM is started by the VM Loader with MAC policy for its reference monitor and the browser’s reference monitor. The Browser
VM also has a Label Mapper, a service designed to find the authoritative labeling state for the application’s web resources (URLs).

The monolithic policy is distributed from the Application Authority to the layers and components that will enforce it. The protection state, labeling state, and transition state for each layer are installed into the correspondent layer’s reference monitor, as depicted in Figure 5.3. The VM Loader has to instantiate the network policy at run-time. The reason is that, we only know the IP address for IPsec policy when the server is contacted. The VM Loader service uses IPsec certificates to authenticate that the web server at that IP address is valid.

The mapping of object identifiers (e.g., file names and URLs) to labels is not stored in the reference monitor. Each application VM is installed with a label mapper, which is set up to determine the authority for labeling an object identifier, and retrieve labels from the correspondent authorities.

Since not all web application data will be available a priori, a web server may be an authority for labeling its URL objects. For example, when a dynamic page is constructed from data retrieved from a database, the web server can use the data’s labels from the database to assign a label for the data. The label mapper enforces policy from the Application Authority to limit the scope of labels that an authority can suggest. For example, a web server may be limited to serving only confidential and public data. Note that if a web server is only authorized to serve only one label, then the web server need not be queried for that label.

5.4.2 Protocols

Here, we define the main protocols executed by the services in our architecture. First, the architecture must disseminate the MAC policy across each relevant component, constructing policy enforcement that is consistent with the overall MAC policy. Policy enforcement occurs when a Browser VM makes a data request. In some cases, the policy may stipulate a browser transition when such a request is made. We give protocols here for initial policy configuration, URL processing, and browser transitions.

5.4.2.1 Initial Policy Configuration

A policy configuration is triggered when a browser loads a new web application. If there is not already a Browser VM for this application, the VM Loader must load one and install the application MAC policy in the VM and VMM as follows:
1. To start a new web application, a URL must be sent to the VM Loader. The VM Loader looks up the application’s Application Authority from a directory and retrieves the application’s complete policy. An example policy for an application is shown in Table 5.2.

2. The VMM and networking policy are installed in the VMM’s own reference monitor and in its networking subsystem.

3. If a VM is already running, it can be updated with the application’s VM and network policies. Otherwise, the VM is configured with the policies and then started up by the VM Loader. If the VM is newly started, it is assigned a label based on the VMM labeling state.

4. The FlowwolF protection state, labeling state and transition state are configured according to the application’s policy. The FlowwolF is started by the VM and assigned a label determined by the VM labeling state.

5. The FlowwolF is sent the URL that was initially sent to the VM Loader and it processes that URL within the newly configured system (see Section 5.4.2.2).

5.4.2.2 Processing a URL

The FlowwolF does not have a priori knowledge of the label state for a given web application. Thus, each time a URL is requested, it must load that URL’s label from the URL Label Mapper. Here, we illustrate how our architecture carries out URL processing, starting with looking up the URL label. In the process we show how policy is enforced with the reference monitor distributed over multiple layers. For correctness, the system must authorize communication requests from the browser VMs and propagate security requirements to support end-to-end information-flow enforcement.

1. A running instance of the browser receives a request to load a new URL. The browser queries the Label Mapper to find the labeling state for the particular URL. For example: GETLABEL http://website/myapplication/showthread.php?tid=5.

2. The VM Loader has previously configured the Label Mapper to know where to find the label for the given URL. We expect application policy to include a pointer to the correspondent authoritative mapper. The Label Mapper queries the appropriate authority (e.g., the web server holding that resource) and returns the label associated with the URL, e.g. Label: TopSecret. The label mapper stores a mapping between the server identifier and the set of labels that that server can assign for the application.
3. The FlowwolF checks to determine whether the current tab can read in data with the requested label (TopSecret). If not, a transition may be triggered (see Section 5.4.2.3).

4. Otherwise, the instrumented browser creates a socket connection with the proper label to retrieve the data. As the network connection must pass through both the VM’s and the VMM’s networking subsystems, the requested label (TopSecret) must be allowed by their policies’ protection state, as well. That is, the MAC OS policy installed in the Web Browser VM authorizes the communication based on the security label of the socket associated to the communication, and establishes a secure communication channel (security association), to convey the label, between the VM and the VMM. Likewise, the VMM networking policy determines the label of the inter-VM secure communication channel, based on the policy installed during the initial policy configuration phase.

5. If authorized at each layer, the communication is forwarded on a secure communication channel (labeled IPsec) to the web server. The secure channel conveys the requested security label and protects the secrecy/integrity of the communication, with the web server. Communication is authenticated using certificates generated by the Application Authority. We describe the process in more detail below.

5.4.2.3 Browser Transitions

Browser transitions may be triggered at the application, VM or VMM layers, causing a new tab to be opened, a new process to be started or a new VM to be loaded, respectively. Here, we describe a protocol for a VM transition.

1. When a FlowwolF receives back a URL label, e.g., TopSecret, that cannot be supported by the current VM, the browser sends a message to the VM Loader asking for a new VM.

2. The VM Loader service checks the VMM transition state and authorizes the request if the calling VM is allowed to transition to a new VM, labeled such that it can serve the application. If the request is authorized, the service looks for an appropriate VM image that can handle the URL’s label (TopSecret).

3. The VM Loader dynamically creates a new VM with the appropriate label for handling the TopSecret URL. It then loads the application’s policies and follows the initial configuration protocol as described in Section 5.4.2.1.
Figure 5.4: An example of the labeling state of our distributed web system when a browser makes an HTTP request to a web server. Label meanings are defined by the protection state in the security policies of the application, VM, VMM and network. Note: some labels may be the same, e.g. $N_1$ and $N_2$.

5.4.3 Enforcement

**VMM Layer** The VMM’s reference monitor controls whether a VM can be loaded at a particular label range for a particular web application. In Figure 5.4, the protection state for the client VMs’ reference monitors had to be consulted before the client could be opened with label $VM_2$.

**VM Layer** The Browser VM’s reference monitor controls access to all operating system resources, including web data stored in files, network sockets, and all process behaviors. The VM’s reference monitor prevents any malware from modifying data, unauthorized users from corrupting browser code, malicious browser plugins from leaking data to unprotected files or from opening web sites that are outside the VM’s range (too secret or too public). In short, the VM sandboxes applications and monitors system-level activity to prevent any process from violating its policy.

**Network Layer** The network layer is responsible for negotiating connections between VMs according to network policy. In Figure 5.4 this would require checking whether labels $N_1$ and $N_2$ were compatible.

**Application Layer** The application layer controls access to labeled objects in the browser application. From Figure 5.4 this would control whether data labeled $N_2$ can be read into a tab labeled $C_{tab}$ or whether inputs entered into the tab (receiving the label $C_2$) could be written out to $N_2$. This approach can be used to prevent a variety of common web attacks like script injection (cross-site scripting), cross-site request forgery (CSRF), drive-by downloads and can implement the same origin policy, the same origin mutual authentication (SOMA) policy or be flexible enough to implement other, relaxed versions of these policies. In short, the FlowwolF reference monitor ensures that secret inputs (username, password, or other data entered by the user in a secret tab) are not leaked to less secret sites through HTML posts or Javascript XMLHttpRequest opera-
tions. The application reference monitor also prevents integrity violations that might result in a cross-site request forgery (CSRF) attack.

The labeling state determines how each subject and object is labeled, but the protection state defines the meaning of the labels. Consider the example in Figure 5.4. In this figure, checks must be made for each object access. For example, when the web browser is started with label $P_2$, the VM policy must be checked to determine whether the VM labeled $VM_1$ may start up a program labeled $P_2$. Then the VM policy must be checked to see whether the process labeled $P_2$ may read and write to a socket object labeled $N_2$. Within the application, when a new tab is opened labeled $C_{tab}$, the VM policy is queried by the application to determine whether an application labeled $P_2$ may open a tab labeled $C_{tab}$. Furthermore, before content data labeled $C_2$ may be read into a tab labeled $C_{tab}$, the VM policy must be queried by the browser to determine if that is legal.

### 5.5 Policy Compliance

We want to enforce end-to-end information flow policies by integrating enforcing mechanisms at several layers. Since we have independent tools at every layer we need to ensure that each layer enforces the system’s security goals, which we call the policy compliance problem. In our architecture, we need to verify policy compliance at two layer transitions: (1) between the application policy and the operating system policy in the web application (Browser) VM and (2) between the web application VM policy and the overall system policy stored in the VMM.

We say that one policy (e.g., an application’s) is compliant with another policy (e.g., a OS’s) if all the information flows authorized by the first policy are also authorized by the second policy. If so, then the reference monitor enforcing the first policy cannot permit an illegal flow of information relative to the second policy. In previous work, we developed SIESTA service [47] to check compliance of a program before it can be executed to ensure that every program executed complies with an OS policy.

Compliance testing is necessary for a class of programs, called trusted programs, that may be entrusted with responsibilities in enforcing the system’s policy. A program is said to be trusted if it is given permissions that enable it to create an information flow that would violate the system policy, but it is trusted to prevent such a flow. While trusted programs are often trusted blindly in current systems, the emergence of application-level reference monitors [153, 91, 116] and security-typed languages [103, 138] now provide a
basis for a program to control its information flows.

5.5.1 Application Compliance

However, operating system and trusted program policies are developed independently, meaning that they cannot be directly compared because of differences in access control models, semantics and/or name spaces. Manual mapping is an option, but it is not scalable. To overcome this problem we observed that trusted programs and their components (e.g., configuration files and executable) usually are higher integrity than the data upon which the programs are applied. The system security policy defines the legal information flows for the data in the system, whereas information should normally flow from trusted program components to the data upon which they are applied. Intuitively, a program’s code and configuration should not depend on the data in which it processes, but the output data necessarily depends on the code used to generate it. As a result, we propose The PIDSI (Program Integrity Dominates System Integrity) [127] approach where trusted program objects are assigned a higher integrity level than the system data, which the trusted program processes. This assignment results in a simple integration of program policy and system policy that enables automated compliance verification of trusted programs.

Using PIDSI, verifying policy compliance of a trusted program policy (e.g., browser) with the application VM policy works as follows. The SIESTA service is started during the Browser VM bootup. While loading the Browser VM, the VMLoad service assigns secrecy and integrity bounds to the VM and also appropriately updates the SIESTA configuration. SIESTA ensures that the given trusted program (e.g., the browser) is executed on the system only if the system and application policies comply: (1) the browser falls within the secrecy/integrity range of the VM (i.e., no flows to illegal labels in system) and (2) does not permit illegal indirect flows between system policy labels (i.e., does not permit a flow from data to a trusted program component directly, which is prevented by construction, or indirectly through any uncommon low integrity program objects). By checking this we guarantee that the browser and the Browser VM system policies comply.

5.5.2 VM Compliance

In this work, we introduce the policy compliance between a VMM policy and its application VMs. As the PIDSI approach helped us configure program policies to enable simplifications in compliance testing, we also want to configure application VMs to com-
ply with VMM policies. The advantage here is that the application VM starts as a blank slate, so it can be configured to comply. The challenge then is to provide an architecture for configuring VMs based on the functional and security goals of the application.

In our architecture, the VMLoad service obtains dynamically-generated application policies for access control, network security, labeling, etc. from the policy store. The VM Loader that runs at the VMM layer uses policy templates as a basis for generating the access control and network policies, as these must include the network identity (i.e., IP address) of the server. The architecture uses IPsec certificates to verify that the servers are authorized to participate in the application. Since the VMM and VM policies are downloaded together, they are presumed to be compliant by default. This compliance should be verified offline, as the information flows are independent of the servers implementing them.

The label mapper policies for labeling distributed resources do not need to be specialized at runtime at present. We use the DNS names for servers, and use IPsec certificates to justify their participation in the application. An important issue is that objects in the VM may already be labeled prior to this execution. If a malicious system mislabels some secret data as public, then the VM could leak it despite using a compliant policy. We do not assess whether a prior labeling of VM objects is correct at present. This problem is similar to verifying the labels of imported objects in system assurance.

The VM Loader retrieves the policies from an application authority. The question is whose authority is represented in the application authority: the client’s, the server’s, or both. In our applications, we are concerned about inadvertent leaks or misuse of low integrity data by our clients, so our application authorities represent the client administrative domain’s view of the policy. Where the server only provides data of one label, it only needs to use the client’s labels in Labeled IPsec communication [61]. Where the server may choose among multiple labels, the server is trusted to choose the appropriate label. We imagine that an offline process is necessary for client domain administrators to agree with application servers on such labeling. Further investigation on this topic is required.

5.6 Implementation

We implemented a prototype according to the design presented in Section 5.4. To handle MAC at the VMM layer, we use the Xen [10] VMM to sandbox browser instances and web servers. To ensure MAC policy enforcement at the VM layer, we run SELinux [111]. At
the network layer we enforce security requirements with labeled IPsec [61], which works with SELinux access control and applies SELinux labels to network sockets. At the application layer, we developed the FlowwolF, an instrument a Java-based web browser (Lobo [122]) to label input and output data and enforce MAC security requirements on them as they flow through the browser.

The web browser gets the labels for the URLs from the Label Mapper, at run-time. Once the browser gets the label assigned to an URL object, it checks the policy to ensure that it is legal to display the requested object in the correspondent browser tab. The browser also sets the SELinux label on the network socket, according to the object label, before attempting the connection to the URL's domain. Finally, the browser requests the VM Loader a new Browser VM when the URL Label Mapper returns a label for the object, thus the network socket, that is rejected by the local SELinux policy.

5.6.1 Implementing an Application Policy

When an administrator installs a new application in the system, he must also define or add a link to the Application Authority for the new application. The Application Authority is expected to define a complete security policy (protection state, labeling state, and transition state) for the new application.

To make the description more concrete, we draw on a bulletin board application [99] we instrumented, so it can take advantage of our system. The instrumented bulletin board (IBB) application enables users to create message threads with security labels. Table 5.2 shows excerpts of the policy for the IBB application. The VMM and VM policies enable the application to access its own resources.

A dimension of the policy is missing in Table 5.2: the transition policy at the VMM layer. This policy determines how to label new Browser VMs when a transition is called for (see the transition protocol in Section 5.4.2.3). We expect administrators to define such policy as a predefined table in the VMM. This decision recognizes the VMM administrator’s stake in controlling how new VMs may be opened and labeled. The VMM administrator may be in a better position than the application developer for making this decision. The following table illustrates a possible transition policy. In this table, ws3 serves bb objects with MLS ranges s0 and s1, ws1 and ws2 serve serve bb objects with MLS range s2 and s3.

```python
table = ('bb', 's0-s1','1' ,'ws3'),
       ('bb', 's2-s3','2' ,'ws1', 'ws2'),
       ('DEFAULT', '0')
```
VMM Policy:
# socket management
allow fwolf_t fwolf_t:tcp_socket {create .. read};
allow fwolf_t bb_t:tcp_socket {create .. read};
allow fwolf_t bb_t:association {recvfrom sendto};
allow fwolf_t ipsec_spd_t:association polmatch;
...

VM Policy:
# socket management (same as VMM policy)
allow fwolf_t bb_t:tcp_socket {create .. read};
...
# file management
allow fwolf_t bb_t:file {read .. setattr};
...

Network Policy: (IPsec labeled)
spdadd <src> <dst> any -ctx 1 1 <context> -P out ipsec esp/tunnel/ <src><dst> /req;
spdadd <dst> <src> any -ctx 1 1 <context> -P in ipsec esp/tunnel/ <dst><src> /req;

Static URL Policy:
http://abc.mil/bb/index.php system_u:system_r:bb_t:s0
http://abc.mil/bb/js/main.js?ver=1400 ... ... system_u:system_r:bb_t:s0
...
Dynamic URL Policy:

Table 5.2: Excerpts of the mandatory policies for the IBB application. The VMM and VM policies enable FlowwolF (fwolf_t) to handle IBB objects (bb_t). The Network Policy is a template to be instantiated with the actual IP addresses of the involved nodes. The URL policy specifies the labels assigned to every URL object in the IBB application. In some cases, this label only can be generated on request (dynamic labeling policy). Also, notice that the labels assigned to elements that belong to an application can be different.

In our implementation we largely derived the application layer policy from the hosting VM’s policy. Since labeled resources originate outside the browser, it is sensible for the browser to inherit the policy for those resources from the OS. By default, the FlowwolF allows MLS-style transitions of data (i.e. data may become more secret or lower integrity, but not vice versa). It would be possible to define the application layer policy in the Application Authority and install it from the VM Loader. In any event, policy validation services [47] should be used to ensure that application policy is compliant with OS policy (this is important for ensuring the distributed reference monitor enforces complete mediation). Ensuring that OS policy is compliant with application policy is important for tamper protection [127].
5.6.2 Improving Performance

We find that caching is possible in various parts of the system: for loading new VMs, for retrieving labels for URLs, and for retrieving application policies.

Loading a new VM when the transition policy requires it, for instance, is a time-consuming operation. To reduce response time, we keep a pool of VMs pre-loaded for certain expected security ranges. In our example above, it would be sensible to keep a pre-loaded VM for reading Top Secret messages. To adjust pre-loaded VMs, to the actual requirements, we developed an update daemon. It loads, for instance, the VM and network policies for a particular application. Having pre-loaded VMs makes a significant performance improvement as described in Section 5.7.2.

Locally caching the URL labels also has a significant impact on the usability of the Flowwolf, because every figure, page and other web objects must have a label. In some cases we use regular expressions (following the practice of SELinux file system labeling) to represent groups of labeled objects.

Lastly, caching application policies in the policy store (part of the VM Loader) accelerates the time for installing the policies on new VMs or pre-loaded VMs. These policies include label state for an application’s URLs, maps from applications to their hosting application authorities and certificates for authenticating IPsec tunnels. On a local miss, the policy store queries the application’s hosting Application Authority.

5.7 Evaluation

To evaluate our approach, we instrumented a bulletin board messaging application. IBB enables users to create message threads with security labels, and displays the content of such threads in a single browser tab only if the tab’s label dominates the thread’s label. We deployed the IBB application on top of our architecture and evaluated results. We presented parts of the IBB policy in Table 5.2.

5.7.1 Security Evaluation

Recall our claim of security from Section 5.3: a monolithic MAC policy is enforced by a multi-layer reference monitor if: (1) each component in the multi-layer reference monitor meets the guarantees for complete mediation and tamperproofing, accounting for trust in its environment; (2) each policy decision by a component in the multi-layer reference monitor is the same decision as that would have been made by a monolithic reference monitor using that policy. In addition to the mediation in SELinux [111] and
Xen [10], we also added mediation to FlowwolF, and defined labeled IPsec [61] policies for networking and inter-VM communication. For tamper protection, we performed a tamperproof analysis [127] of the deployment of our browser package on SELinux, and found that only trusted processes could modify the browser files. For policy decisions, our VMLoader protocol distributes MAC policy for protection state and transition states as is, so the same decisions are made as in the monolithic policy. A component cannot make an unauthorized decision as the VMLoader assigns each a security label. For labeling state, the monolithic MAC policy defines the labeling authorities (components) for different parts of the name space.

In this paragraph, we focus on the instrumented bulletin board application. Our architecture can prevent the following attacks. (1) In a stored XSS attack, a public user of the IBB might try to poison a secret user by planting a URL to a malicious script that will be automatically loaded when the secret user views the message. Because the malicious script’s URL will be labeled public by default, however, the script will have no access to secret data in the secret user’s browser. (2) In an CSRF attack, a public user attempts to trick a secret user into posting a secret message using a command like XMLHttpRequest that will not be noticed by the secret user. In our system, the data being posted by the XMLHttpRequest would be labeled public by default, and so could not be posted in a secret message. (3) Network layer attacks like eavesdropping and MITM are prevented by the authentication and encryption provided by Labeled IPsec. And (4) MAC at the VM layer contains malware and prevents it from modifying data stored in the system and owned by other applications. As part of our future work, we will further test our architecture by loading well known XSS and CSRF attacks and analyzing results.

5.7.2 Performance

In this section, we evaluate FlowwolF by measuring its overall runtime overhead and overhead for its major components. All of the experiments were run on two machines that represent the client and server sides of our applications. These machines run Linux 2.6.19 with Xen support. They both have Intel Pentium processors with 2.8 GHz, and 1 GB of RAM. We repeated all the experiments 10 times and present the average of each operation (Table 5.3).

In Table 5.3, we compare three cases: (1) install of a new browser VM to load a web

---

3A Linux package is a self-contained distribution of files necessary for an application, including its libraries, configuration files, and executable.
Table 5.3: Browser configuration and (first) page loading times for a new VM, for a pre-loaded VM, and Firefox.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>New VM</td>
<td></td>
</tr>
<tr>
<td>Configure/Load Policy</td>
<td>4.90</td>
</tr>
<tr>
<td>Load VM/Browser/Page</td>
<td>18.50</td>
</tr>
<tr>
<td>Pre-Loaded</td>
<td></td>
</tr>
<tr>
<td>Configure/Load Policy</td>
<td>2.52</td>
</tr>
<tr>
<td>Load Browser/Page</td>
<td>3.44</td>
</tr>
<tr>
<td>Firefox</td>
<td></td>
</tr>
<tr>
<td>Cold-start Load Page</td>
<td>7.45</td>
</tr>
<tr>
<td>Warm-start Load Page</td>
<td>1.50</td>
</tr>
</tbody>
</table>

page into FlowwolF; (2) configuring a pre-loaded VM to load a web page into FlowwolF; and (3) a native browser. In the first case, we configure the policy files in the VM image, and then load the VM, FlowwolF browser, and finally the web page. The overall load time is over 20 seconds. Comparing the new VM and pre-loaded page loading times, we can see that the time to load the VM dominates the 18.50 seconds. The time to load VM services takes a major portion of this time, particularly the VNC server we use to load the browser automatically (nearly 8 seconds to load). The Tahoma prototype takes over 9 seconds to load a web page [29]. Removing most services would result in a similar performance. We also note that the Tahoma prototype was unoptimized, so further significant improvements are likely.

We also note that the pre-loaded VM takes half as long to configure as the new VM – 2.52 seconds for a pre-loaded VM versus 4.90 seconds for a new VM. For a pre-loaded VM, it is not necessary to write to the VM image, so several disk operations were removed. As a result, we recommend using an update daemon to load policy, even for new VM loading.

In comparison, the bottom line shows the page load times for a Firefox browser on the VMM (Xen domain 0). Loading a page in a pre-loaded VM with FlowwolF is comparable, due to FlowwolF’s low cold-start time. In general, we would expect to add the configure/load policy time of 2.5 seconds to any security-aware browser on the first page load.

The IPsec overhead required to negotiate secure channels is not shown in Table 5.3. At present, we are using three IPsec channels to convey security labels: (1) from browser VM to client host VMM; (2) from client host VMM to server host VMM; and (3) from server host VMM to server VM. Each channel takes approximately 3 seconds to negotiate, but once it is established it can be reused. In general, we only need one channel to protect the communication between the server and the host, so we are looking into mechanisms with faster setup to convey labels locally, such as netlabel [110].
5.8 Related Work

The primary focus of our work is in developing a multi-layer mandatory protection system. While others have developed MAC systems at various layers, including the components we use (SELinux, labeled IPsec, Xen, etc.), there are no other efforts seeking to harmonize these layers for implementing a single end-to-end policy. We apply our developments directly to web applications. There are some other systems that use isolation technologies to provide security for web systems.

Tahoma [29] provides support for running isolated web applications. Tahoma enables web servers to define policy for the clients, but we are also interested in enabling clients to express their own policy. OP [38] redesigns the browser’s architecture. The new architecture has subsystems for separate plug-ins and makes explicit all communications between them. The browser kernel, the heart of the OP design, can mediate all communications between subsystems and enforce policies on those communications. While OP can enforce policies on the communications between subsystems, it is not designed to enforce policies on the communication channels between clients and servers, nor to dynamically obtain and configure client and server policies. Like OP, MashupOS [55] suggests the creation of abstractions to address security issues on browser implementations. In particular, abstractions to represent resources within the browser (i.e. disk, network) and access controls to rule access to those resources. The Same Origin Mutual Approval policy, SOMA [113], aims to block malicious scripts from downloading data from arbitrary web sites as part of pages served by known sites. To do so SOMA policy requires browsers to verify, with web servers, whether data inclusion is allowed or not. Our approach and these approaches may need to address similar problems, however while we need to represent multiple layers, these approaches do not have to.

NetTop [94] provides support for running multiple single security level systems. These systems are virtually isolated and may communicate with others only if they belong to the same security level. NetTop provides support at additional layers (not only application layer) but it only supports a fix predefined MLS policy while we want to support a wider range of policies. DStar [163] aims to provide support for distributed applications. It propagates labels defined at the OS layer across systems, throughout label enforcement at the network layer. However, DStar does not integrate enforcement at the application layer, although that is mentioned as future work. OMOS [160] provides communication mechanism for allowing different domains to exchange data among them authorized by a security policy. As DStar, it does not enforce any restrictions on data of different security levels within an application itself. Also it does not provide the system
wide enforcement guarantees of our architecture.

5.9 Discussion

Currently, virtual machine environments do not provide adequate mechanisms to configure flexible security policies for VMs and the other layers that are involved in configuring security enforcement for end-to-end systems. In this paper, we presented an architecture that enables administrators to configure VMs and their communications so they are compliant with a system security goal. We implemented this architecture based on tools that provide mechanisms to specify and enforce mandatory access controls at virtual machine monitor, operating system, network, and application layers, and to convey security information between them.

We evaluated our implementation using an instrumented bulletin board web application. Our architecture establishes secure channels between the Web Browser VM, and the bulletin board server, across all layers in the multi-layer reference monitor, according to a specified configuration. We found, that our architecture prevents some well known attacks to web clients.

We also evaluated the response time to load a new VM, in the context of our test application. We used the concept of pre-loaded VMs to reduce the latency to load a new VM. Our evaluation shows that despite the overhead imposed by different components, the proposed system incurs only nominal performance overhead.

As future work, we plan on examining the effectivity of our approach for other distributed applications. We also plan to explore ways use other mechanisms to label network communications and convey security requirements between virtual machines.
Chapter 6

Distributed Compliant Systems

This chapter presents an analytical model we develop to assist administrators in the deployment of distributed MAC systems that defend security-relevant resources from external attacks. It also presents the services we implement to automate the construction of the model and its resolution if the system it represents does not meet the expected security goal, i.e., it does not defend security-relevant resources from attacks.

Deploying a distributed MAC system that meets a security goal is a challenging task for system administrators because the number of components (including programs, operating systems, and possible virtualized environments), their multiple interactions, and the complexity of some of them. Since our services address these problems they reduce the burden on administrators to deploy distributed MAC systems making these systems more practical.

6.1 Introduction

Computing system compromises occur because system integrity is not managed effectively. Classical integrity models [15, 24] tell us that unassured processes cannot be trusted to modify high integrity data securely if they receive untrusted inputs, but in a typical deployment some processes are required to receive untrusted input, making them accessible to adversaries. Making matters worse, system deployments are now more complex than ever, consisting of multiple collaborating machines, where each may be built from multiple, independent software layers, including virtual machines, middleware, and their programs. Thus, defending a distributed system from adversaries proactively is challenging.
Mandatory access control (MAC) enforcement was supposed to fix these kinds of problems by limiting the number of processes accessible to adversaries and confining the rest. MAC enforcement is now available in virtual machine monitors [26, 132, 82] (VMMs) and user-level programs [148, 103, 157], in addition to operating systems, enabling such control throughout the system. Furthermore, such MAC enforcement is now integrated with network access control [97, 61, 110], presenting an opportunity for distributed control. However, the addition of all this enforcement does not seem to be changing the dynamics of security management. The task of system administrators, OS distributors, and programmers is still reactive, fixing vulnerabilities as adversaries identify them. Now, in addition to complex programs, there are complex access control specifications to re-configure, making the job of defense even more difficult.

To prevent vulnerabilities proactively, we need to configure distributed system deployments to minimize the cost of defending information flow integrity violations. We state that such violations are defended through mediators, which express where runtime system behavior must prevent violations from becoming vulnerabilities. There are three main challenges in achieving this goal. First, it is difficult to find integrity errors in a distributed system. Integrity requirements are rarely stated in terms of information flow in modern systems. Instead, MAC policies aim for least privilege [133], where processes are run with only the permissions they require. However, this goal is functional, so it may result in many integrity violations. Second, once an error is located there may be many ways to mediate it. If an adversary’s input to an unprivileged process can indirectly lead to the installation of a rootkit, a proactive approach would aim to mediate this flow. However, there may be many locations in the system to mediate such a requirement, and only some may be capable of enforcing the desired requirements. Third, it may also be possible to modify the policy to reduce the need for mediation, but MAC policies are now too complex for system administrators to modify. Instead, OS distributors define generic MAC policies for their distributions [155, 111, 109, 98, 154], but these policies must be liberal enough to authorize a variety of feasible deployments. Thus, while system administrators may need to customize their policies to their deployments to defend their systems proactively, the task of doing so is now impractical.

We identify three insights that motivate us to develop a methodology for proactive integrity protection of distributed systems. First, the security community knows how to compute information flow integrity errors for complex policies, including non-information flow policies. Methods have already been developed to find information flow errors [63, 134, 20, 40, 159, 125] for system and program policies. However, using these
tools requires significant manual effort to configure the target deployment for analysis and choose among possible mediator locations. Second, methods have also been proposed to place mediators to resolve violations. Pike [120] and our team [71, 72] demonstrated independently that resolving information flow errors can be modeled as a graph-cut problem. However, these methods were only applied to single policy systems with two-level lattices. Also, practical considerations, such as the appropriate estimation function for the cost of mediation, were not explored. Third, emerging computing architectures, such as utility computing, encourage the construction of pre-configured distributions to be run on known host systems, which may amortize the cost of configuring for deployment. In this case, one carefully pre-configured deployment may be used by many customers.

In this chapter, we develop the Proactive Integrity Methodology, a mostly-automated approach for generating system-wide, information flow integrity policies to minimize the cost of defending information flow integrity violations. This methodology consists of four main tasks.

- The first task constructs a system-wide data flow graph automatically from available configuration information and security policies (e.g., MAC and network). This data flow graph preserves the inherent hierarchical nature of systems to produce the first system data flow graph that is distributed and layered.

- The second task produces the security constraints necessary to identify information flow integrity violations. We find that relationships exist in systems that enable inference of many of these constraints, although some manual specification is required.

- The third task computes a minimal cost mediation as a graph-cut. We extend the proposed approaches to support a distributed system and a general lattice.

- The fourth task automatically generates a system-wide information flow integrity policy in the Decentralized Information Flow Control (DIFC) model [76, 162]. This policy model expresses information flow requirements and where mediation is placed. To date, DIFC policies have only been configured for a few components, so providing a method to generate DIFC policies automatically for all components in a distributed system is a significant contribution.

This research makes the following contributions.

- We develop an automated method to construct a hierarchical data flow graph [5]
that represents all independent MAC and network policies governing a given distributed system.

- We show how to construct a graph-cut model for a hierarchical data flow graph and compute a cut-set for: (1) an arbitrary lattice policy, where (2) nodes may have limited mediation ability. Our method includes the implementation of a graph-cut approach to find the minimal cost of mediation for each cut problem and composes them into a system-wide mediation that is near-minimal.

- We show that a secure configuration in this model is equivalent to a secure configuration in the DIFC model used in the Flume system [76]. Based on this result, we develop a method to generate DIFC-Flume policies automatically.

- We demonstrate this approach on a distributed web application system consisting of hosts running the Xen virtual machine system [10] and/or application-specific SELinux distributions [111]. We show that the methods are practical for use in configuring distributed systems and that a variety of useful analyses to reduce mediation cost are possible.

The remainder of the chapter is structured as follows. In Section 6.2, we motivate the problem, including a discussion of related work. In Section 6.3, we describe the tasks in the Proactive Integrity Methodology. In Section 6.4, we provide background for the technologies that we are building upon. In Section 6.5, we show how to build our model of a system, how to compute near-minimal cost mediation using a graph-cut method, and how to generate a system-wide DIFC policy. In Section 6.6, we evaluate the efficiency and effectiveness of this method to compute attack surfaces and information flow integrity policies. Finally, we briefly discuss related issues and conclude in Section 6.7.

### 6.2 Problem Definition

#### 6.2.1 Example System

Distributed systems are increasingly more complex. They consist of multiple interconnected hosts, each running several programs hosted by operating systems and in some cases by virtualized environments. Administrators create distributed systems that aggregate programs, operating systems, and virtualized environments, developed by third parties, to meet functional requirements while enforcing a security goal, more precisely, while preventing unauthorized modification or leakage of data.
Figure 6.1: End-to-end web application system

Figure 6.1 shows a simplified web application system. In general, web applications, such as the so-called LAMP systems, consist of (Apache) web servers, (MySQL) databases, and various web application (Perl/PHP/Python) programs running on one or more operating systems (Linux). In Figure 6.1, a web server application and DB each run in separate guest VMs. The purpose of this system is for the web application to respond to requests from authorized clients, where such requests may require querying or updating the database. The guest VMs are deployed on a host system consisting of a virtual machine monitor (VMM) and its privileged VM, such as the Xen hypervisor [10] and its domain 0 host, respectively. Such systems may also depend on the integrity of network servers to prevent network attacks, such as DNS hijacking, and occasionally the authorized clients themselves. Such a deployment would be analogous to many server systems as well as some utility computing environments, such as a IaaS clouds (public or private).

While the web server provides access to the web application for external clients, it also provides a means for adversaries to launch attacks against the application and the system at large. Even if the web server protects itself from compromise, it forwards data that is accessible to other web application components (e.g., PHP scripts and databases) that may compromise those components (e.g., Saner attacks [9] and SQL injections [118, 143]). The web server also interacts with a variety of system processes, which may be compromised by those interactions or local exploits launched from compromised web application components. Finally, a compromised system hosting a web server or application component can then launch attacks against other hosts and the virtualization infrastructure. Of course, web applications are just one of the many networked applications and services to worry about.

Traditionally, it has been the responsibility of system administrators to configure security policies for their system deployments, but the complexity of systems and of
the security policies themselves has made this task impractical. To ease the system administrators’ job, several OS distributions are now deployed with their own mandatory access control (MAC) policies [155, 111, 109, 154, 98], and administrators now depend on these policies for the security of their deployments. In addition, other software layers now have MAC enforcement mechanisms, including VMMs [132, 26] and some privileged programs [153, 22, 37]. Such MAC enforcement is also integrated with network control [61, 110, 97] to enable distributed enforcement. Despite the comprehensive definition of MAC policies, however, system administrators are still reactive, responding to adversary exploits as they are released.

Instead, what a system administrator would like is a system-wide MAC policy for their deployment that prevents attacks proactively. When a system administrator chooses the deployment software (including their default MAC policies for those components that perform MAC enforcement) and configures the software and network policy, she would like a MAC policy that identifies how all threats created by such a deployment are defended, preferably in terms of classical information flow integrity [15, 12]. Despite the individual efforts of OS distributors and others, determining the threats resulting from a system deployment of multiple, independent software components is still ad hoc. Thus, the impact of a web server on the security of a web application and a system at large, such as a cloud computing environment, is unclear, risking unforeseen compromise.

### 6.2.2 Policy Generation Problem

Figure 6.2 shows the general problem we explore in this paper. We want to use the existing *component MAC policies* in a system deployment and a *goal specification* of the legal system operations in that deployment to generate a *system-wide MAC policy* that prevents operations that do not satisfy the goal specification. To construct this problem, we suggest using an information flow approach used in program and system policy analysis, where: (1) component MAC policies are converted to data flow graphs representing the authorized information flows [100, 63, 148] and (2) goal specifications are specified in terms of lattice policies [31, 15, 12]. Source and sink nodes in the data flow graph are assigned to levels in the lattice policy, enabling identification of information flow errors, nodes whose input data causes an information flow that violates the lattice policy.

The aim is to generate a system-wide MAC policy that contains no information flow errors, thus preventing all operations that do not satisfy the lattice policy. This is achieved either: (1) by removing data flow edges in the graph until no information
flow errors remaining or (2) by adding mediation that changes the level of data output by a mediating node to resolve information flow errors. First, removing data flow edges removes authorized operations from a MAC policy, possibly removing necessary function. Thus, the goal specification needs to be augmented with functional constraints in addition to security constraints (i.e., the lattice policy). Typically, functional constraints are not explicitly specified, but commodity MAC policies often approximate functional requirements because they are constructed from runtime analysis [123]. Thus, it is difficult to remove operations without further justification. Second, security-typed languages permit the addition of mediation expressions to resolve information flow errors in programs. However, adding new subjects to system policies may involve adding new programs (e.g., guards [15]) or adding (or acknowledging) mediation abilities in existing programs (e.g., transformation procedures [24]), although neither are typically formally assured [136].

This policy generation problem is computationally complex because both contradictory security (negative) constraints and functional (positive) constraints must be satisfied simultaneously [42]. There are typically two approaches taken to solving such problems: (1) such problems are converted to a format solvable using off-the-shelf SMT solvers [30, 66] or (2) a greedy method is used to find a satisficing, although potentially sub-optimal solution. SMT solvers have been used to generate network access control policies [145]. While it may be possible to encode our problem for an SMT solver, a key problem is that both functional and security constraints may be missing, at least initially. For example, the MAC policies provide no indication of the functional constraints required for a particular deployment. This is something that the policy designer must learn, and a solution to the policy generation problem may tell them the cost of having an operation (e.g., in terms of the mediation required).
Thus, we explore greedy solutions to the policy generation problem, but even developing a greedy solution has several challenges. First, the multiple, independent MAC policies in a system must be composed into a single, coherent model. For example, OS MAC policies do not specify which processes may access virtualized resources managed by a VMM MAC policy, and the VMM MAC policy views all OS processes as equivalent (i.e., have the same label in the VMM MAC policy). Second, often security goals are not explicitly specified either, meaning that they must be inferred or discovered by experimentation. While researchers often generate security goals for individual MAC policies, we note that a system-wide analysis requires security constraints between entities in different MAC policies. Third, once a solution is found it must be converted into a system-wide MAC policy. However, commodity MAC policies only express authorized operations, not mediation, so we cannot simply convert the result back into one of the commodity models without losing information. We aim to build a greedy approach that solves these problems to enable policy designers to explore functional and security trade-offs in generating system-wide MAC policies that remove or mediate all attacks.

6.2.3 Related Work

Researchers have long known that information flow is a principle upon which distributed integrity protection is possible. However, classical integrity models [15, 24] rely on formal assurance for all high integrity processes, particularly those that receive untrusted inputs. Many “trusted” processes may receive untrusted input, but formal assurance methods have not been developed that scale to the size of modern programs. As a result, least privilege [133] has been adopted as the security goal for integrity protection in commodity systems [111, 109, 98, 154, 155]. However, different deployments imply different privileges, so commodity systems try to accommodate this through mechanisms to easily compute permissions requested by processes [123, 109, 135] and by providing runtime configurations options to add function. Nonetheless, it is important that default configurations work in most cases, so researchers have found that the default MAC policies still permit several operations that would violate classical integrity [20]. Researchers have recently proposed practical approaches to integrity that approximate classical integrity models [136, 80, 144, 76, 102]. These practical integrity models are all strictly weaker than classical models, in that they do not require formal assurance for code with the authority to protect integrity while receiving untrusted inputs. However, they express restrictions on how such authority may be used by high-integrity programs. For example, the Decentralized Information Flow Control [76, 162] (DIFC) model provides
processes with capabilities to make decisions about handling untrusted inputs.

With the emergence of MAC enforcement in commodity operating systems, researchers proposed various policy design tools to help system administrators and OS distributors configure policies [63, 149, 40, 134, 159, 20]. Broadly speaking, these tools enable a policy designer to evaluate reachability to identify whether an adversary can perform an unauthorized operation, even indirectly. Similar reachability analyses have also been performed for network policies in the form of attack graphs [117, 137, 108]. However, defining which operations are unauthorized and resolving any problems are manual tasks. SELinux policy generation from such resolutions is possible, but the reasons for the resolutions and the resulting risks are lost. Pike [120] and our prior work [71, 72] independently found that resolutions could be estimated as graph-cuts automatically. However, these approaches approximate minimal solutions for two-level lattices only, so they cannot reason about distributed systems with multiple-independent policies, as we see today. Another key question is what security property should be optimized by the choice of such cuts. Researchers have recently focused on defending an attack surface [54, 84], which is the set of program interfaces accessible to adversaries. The idea would be that if we can minimize the number of interfaces accessible to attackers, we could minimize our defensive effort.

The problem of verifying that a MAC policy satisfies a set of security requirements is a policy compliance problem. In a policy compliance problem, a policy is said to comply with a goal if all the operations authorized by the policy satisfy the constraints of the goal [90, 32, 86, 52, 127]. The problem is that MAC policies often fail to comply with integrity requirements, as discussed above, so we must repair non-compliant cases. However, any MAC policy changes must also preserve necessary function, and balancing functional and security requirements is computationally complex in general. Researchers have shown that network policies can be generated automatically from a set of functional and security constraints using Satisfiability Modulo Theories (SMT) solvers [145]. Such solvers [30, 66] can now solve large logical formulas in practice. Unfortunately, neither security nor functional constraints are specified explicitly. Further, there may not be satisficing solutions for functional and information flow integrity constraints. This further motivates the need to leverage “practical” integrity models to weaken information flow integrity constraints in a limited way.
6.3 Solution Methodology

Figure 6.3 shows an overview of our Proactive Integrity Methodology to deploy systems to minimize information flow integrity risks. First, we automatically convert a distributed system specification consisting of independent OS distributions, including their MAC and network policies, into a single data flow graph that represents the operations authorized by that system and its policies. We detail some of the key tasks in the conversion in Section 6.5.2. Second, we develop a graph-cut model using this data flow graph in Section 6.5.3. We use rule bases from our prior experiences to infer where security-critical data is imported into the system and the integrity relationships between distinct imports. This step may also involve some manual specification of security constraints, as the inferred requirements must be conservative. From this point, information flow integrity errors can be identified automatically using known techniques, so mediation placement may be determined semi-automatically. Third, the graph-cut model is input to a graph-cut method, which produces system-wide mediation placements for arbitrary integrity lattices in Section 6.5.4. Finding a minimal graph-cut over a general lattice is an NP-hard problem, but we solve the individual cut problems in an order to maximize reuse. Functional options may be specified manually or inferred to suggest changes to the graph-cut model that may reduce the mediation cost. Fourth, we use the graph-cut to generate a system-wide information flow integrity policy in the DIFC-Flume policy model in Section 6.5.5. We first demonstrate equivalence between the lattice in the graph-cut model and the DIFC-Flume policy model, and then describe a method for generating such a policy automatically. The result is a MAC policy that reflects the practical integrity requirements to defend the distributed system.
6.4 Solution Background

In this section, we provide background on the key technologies used in developing the Practical Integrity Methodology’s design, which is described in Section 6.5. The first problem is to represent a system consisting of multiple, independent MAC policies for distributed systems in a uniform way that enables efficient representation that captures the security context accurately. For this, we leverage the representation of a hierarchical graph as defined by the hierarchical state machine model (HSM) [5], described in Section 6.4.1. After that we build a system-wide information flow model that we use to generate mediation placements that will resolve information flow errors system-wide. For this, we leverage a general graph-cut model [72], described in Section 6.4.2. Finally, we must convert a cut set that solves information flow errors to a system-wide information flow integrity policy. For this, we leverage the decentralized information flow control [76] (DIFC) model discussed in Section 6.4.3.

6.4.1 Hierarchical Graphs

Prior work has converted one MAC policy to a data flow graph for various analyses [148, 134, 63, 20], but systems often consist of multiple entities that enforce independent MAC policies. Thus, there is a need to build a unified data flow graph from multiple policies that can represent these flows effectively and enable model optimizations. In this section, we show how to use a hierarchical model to express an information flow graph resulting from multiple, independent MAC policies.

The hierarchical state machine (HSM) model [5] is an hierarchical, encapsulated graph data structure used frequently by the model checking community. Analysis of the model’s performance in solving a variety of computing problems, such as reachability,
equivalence, and inclusion, over this data structure are well-known [4].

**Definition 6.4.1.** A hierarchical state machine $K$ is a tuple $(K_1, \ldots K_n)$ of modules, where each module $K_i$ has the following components:

- A finite set $V_i$ of nodes.
- A finite set $B_i$ of boxes.
- A subset $I_i$ of $V_i$, called *entry nodes*.
- A subset $O_i$ of $V_i$, called *exit nodes*.
- An indexing function $Y_i : B_i \rightarrow \{i + 1, \ldots, n\}$ that maps each box of the $i$-th module to an index greater than $i$. That is, if $Y_i(b) = j$ for box $b$ of module $K_i$, then $b$ can be viewed as a reference to the definition of module $K_j$.
- If $b$ is a box of the module $K_i$ with $j = Y_i(b)$, then pairs of the form $(b, u)$ with $u \in I_j$ are the *calls* of $K_i$ and pairs of the form $(b, v)$ with $v \in O_j$ are the *returns* of $K_i$.
- An edge relation $E_i$ consisting of pairs $(u, v)$, where the source $u$ is either a node or a return of $K_i$ and $v$ is either a node or a call of $K_i$.

The key concept in HSM is a *module*, which represents a single graph of nodes and edges within the HSM. Each component that enforces its own MAC policy is represented by a module. Components that do not enforce a MAC policy are represented as nodes. The other important feature of the HSM model is its flexible model for expressing data flows between modules. *Boxes* describe independent data flows from a parent node to a child node through calls and returns. This flexibility is important because data may enter a module from multiple sources, and each of these distinct inputs may be represented precisely using the HSM model. Also, a module may receive multiple flows from the same source with distinct security requirements, but the model can be constructed to represent such flows precisely while eliminating redundant flows.

We find that the HSM model can accurately model the data flows among components in distributed systems, because software layers are arranged hierarchically and components are encapsulated by the peer layers, as shown in Figure 6.4. A system is *hierarchical* in that the responsibility for security decisions is monotonically-reduced from the root to the leaves. For example, a VMM has full access to the physical resources of the platform, VMs have a distinct subsets of these resources (e.g., consider physical memory), and each VM process may only access a subset of the resources available to its VM. A system is also *encapsulated* in that all interactions between two peer components are implemented by an ancestor layer (usually the parent). For example, an operating system or VMM provides the only mechanisms available for two processes to communicate\(^1\).

\(^1\)While two processes may setup shared memory to communicate without OS intervention, the OS
6.4.2 Graph-Cut Model

Once a model of the distributed system is constructed, we want to compute mediation placements that resolve all information flow errors and have a minimal cost. Pike [120] and our prior work [71, 72] independently found that such mediation placements could be estimated as graph-cuts.

In this work, we use the information flow model defined below:

1. A directed data flow graph \( G = (V, E) \) consisting of a set of nodes \( V \) connected by edges \( E \).
2. A lattice \( L = \{L, \leq\} \). For any two levels \( l_i, l_j \in L \), \( l_i \leq l_j \) means that \( l_i \) ‘can flow to’ \( l_j \).
3. There is an import mapping function \( M : V \rightarrow \mathcal{P}^L \) where \( \mathcal{P}^L \) is the power set of \( L \). Some nodes are mapped to a set of levels in \( L \).
4. The lattice imposes security constraints on the information flows enabled by the data flow graph. Each pair \( u, v \in V \) s.t. \( [u \rightarrow_G v \land l_u \not\leq_L l_v] \), where \( \rightarrow_G \) means there is a path from \( u \) to \( v \) in \( G \), represents an information flow error.

It has been shown that programs can be automatically translated into such a model [102], as used by King et al. [72], and we show that a MAC policy can also be translated into the model automatically, based on previous work [120, 148, 134, 20, 63]. In a MAC policy, operations can be classified as read-only, write-only, and read-write, between subject labels and object labels in the policy. From this classification, a directed data flow graph can be constructed where the nodes are the subject and object labels and the edges are the flows created based on the authorized flows among them. Also, previous work has shown that MAC policies, even ones not based on information flow models (e.g., Type Enforcement [18]), can be associated with a lattice policy by declaring an import mapping that assigns some of the labels in the MAC policy to levels in the lattice [63]. From this model, information flow errors can be detected.

The insight of King et al. and Pike is that placing a mediator that resolves an illegal information flow between two levels \( l_i \) and \( l_j \) is tantamount to generating a vertex cut of the information flow graph with \( l_i \) as the source and \( l_j \) as the sink. They call this

---

2 In this paper, we use label for the MAC policies (e.g., SELinux and DIFC-Flume) and level for the lattice nodes, as in the graph-cut model definition.

3 In graph theory, given a graph \( G = (V, E) \), a vertex cut of this graph with respect to a source and a sink is a set of vertices whose removal will divide the graph into two parts, one containing the source and the other containing the sink, such that the sink can no longer be reached from the source.
property Cut-Mediation Equivalence [72]. The graph-cut problem for a general lattice is an instance of the \textit{weighted cut-conjunction problem}, which has been shown to be NP-hard [70], so we need a practical approach to estimate near-minimal cuts effectively. Second, the goal is to minimize the cost of defending the system, so we need to explore the effectiveness of possible cost metrics.

\subsection*{6.4.3 DIFC-Flume Information Flow Policy}

To represent the information flow integrity policy resulting from the proposed approach, we use the Decentralized Information Flow Control (DIFC) model defined for the Flume system [76]. Unlike prior approaches to practical integrity [136, 80, 144], the DIFC-Flume model is capable of representing arbitrary information flow requirements, including exceptions to information flow enforcement in the form of capabilities. Thus, we find that DIFC-Flume is expressive enough to represent complex MAC models, such as SELinux [111], and represent information flow exceptions that are not explicit in SELinux. However, like SELinux, manual specification of DIFC-Flume system-wide can be quite complex, so instead, we advocate using the prior manual experience in policy design for individual components to produce a system-wide DIFC-Flume policy automatically that has a near-minimal defense cost.

Flume uses tags to track data in a system, and labels to authorize operations. A \textit{tag} represents an identifier of a secrecy or integrity property, while a \textit{label} is a set of tags. The labels form a lattice under the subset relation, $\subseteq$, that determines whether data reading or writing operations are safe. Each process $p$ has a label, $I_p$, that defines its integrity. For every tag $t$ in the label $I_p$, every input to $p$ must be endorsed to satisfy the integrity requirements for $t$. A message from a process $p$ to a process $q$, or any type of communication, is legal only if all of $p$’s inputs have been endorsed for each tag in $q$’s label, at minimum: $I_q \subseteq I_p$.

A DIFC-Flume process $p$ may also have capabilities, special privileges that allow the modification of process’s labels and the data it generates. A process $p$ with $t^+$ in its capability set $O_p$ ($t^+ \in O_p$) can add tag $t$ to its label $I_p$, and with $t^-$ in its capability set ($t^- \in O_p$) can remove tag $t$ from its label, granting itself permission to endorse data with higher integrity or to receive data with lower integrity, respectively. A process can temporarily change its label only for tags for which it has dual privileges $D_p$. A process with $D_p = \{t\}$ can add and remove $t$ from its label. Therefore, a message from $p$ to $q$ is legal if $I_q - D_q \subseteq I_p \cup D_p$. 
6.5 Solution Design

In this section, we describe the design of the tasks in the Proactive Integrity Methodology described in Section 6.3 using the technologies defined in Section 6.4. The overall goal is to compute practical information flow integrity policies that minimize the cost of mediation for distributed systems.

6.5.1 Specification

Section 6.2.2 introduces the compliance problem for distributed MAC systems. Briefly, the compliance problem determines whether a system policy (that represents a system’s behavior) meets the requirements defined by a security goal (that represents the accepted behavior).

Figure 6.2 presents an overview of the compliance problem. First, we construct the system policy from all its components and infer a goal policy and the corresponding mapping function. Then we evaluate compliance. Second, if the system is not compliant, we use the concept of mediators and compute a set of mediators that resolve the errors in the system. Finally, we generate an information flow policy, based on the initial policy, that integrates the mediators we compute.

The inputs we consider as described in Figure 6.2 are the system components, the connecting policies, optional specification, and expert knowledge. Each component, $C_i = (id, parent_id, MAC\text{policy})$, represents an entity that enforces a MAC policy and its hierarchical relations. A MAC policy, $G_i = (V_i, E_i)$, defines a set of labels $V_i$ and the flows that are allowed among those labels $E_i$. The hierarchical relation represents an underlying relation of dependency. For example, VMs depend on their VMMs and applications depend on their operating systems. There is one single root component we use to represent communications between VMMs.

A connecting policy defines paths between labels corresponding to different components. For example, the result of combining two network policies at component $C_i$ and $C_j$ defines what packets may be sent from network ports in component $i$ and received in network ports in component $j$. These paths between two components may go through their parent, as this is the way communications happen in real systems. In the case of VMMs for instance, the monitor receives a signal from the sender VM indicating that data is ready to be sent and sends a signal to the target VM indicating that data is ready to be read.

We enable administrators to define additional data (optional specification in Fig-
Figure 6.5: Outputs: goal policy $\mathcal{L}$, mapping function $\text{map}$, and an Information Flow (IF) Policy. The policy is based on the input components $\text{Component}_1, \ldots, \text{Component}_i$. The IF policy assigns an integrity level and mediation capabilities to every label defined by the initial policy. The level is the integrity a label (i.e., the program represented by that label) can provide, and the mediation capabilities identify lower integrity inputs that the label can handle and sanitize.

Figure 6.2) to refine our process to construct a specification and find a solution for a non-compliant system. For example, an administrator may want to define explicitly a set of resources that should be protected, thus affecting the goal and mapping function that we may infer. Finally, we collected and defined rules, i.e., expert knowledge, we use to infer an integrity goal that represents a conservative behavior of the system. Similarly, we infer a mapping function that identifies security-relevant resources. We explain our inference procedure and rules in Section 6.5.3.

The results of the process Figure 6.2 describes are: (1) an integrity goal, $\mathcal{L} = (L, \preceq)$, (2) the corresponding mapping function, $\text{map} : V' \to L$, where $V' \subseteq V$ (the union of the vertices of each component $C_i$) and (3) an information flow (IF) policy. We already defined the purpose of the goal and the mapping function. About the resulting information flow (IF) policy, we use a policy language that can express mediators. The IF policy assigns an integrity level, $l$, and mediation capabilities, $m$, to every label, $b$, defined by the initial policy. $l$ is the highest integrity that $b$, that is the program represented by that label, can ensure, and $m$ identify all integrity levels lower than $l$ that $b$ can handle, sanitize, and upgrade to a higher integrity. We further explain mediation capabilities and the algorithm we use to compute them later. Figure 6.5 describes the resulting policy.

### 6.5.2 Building HSM Data Flow Graphs

In the first task, we use the input components $C_i$ (associated to VMMs, VMs, and programs) to construct an HSM data flow graph that represents all the possible data flows in a system. In this section, we demonstrate some of the key steps in this construction
and optimizations to limit the size of the graphs while maintaining security semantics.

In general, the overall approach is to create an HSM module for each MAC policy, that covers all of the information flows authorized by those policies. Then, we connect the HSM modules using policies that govern interaction between components. We demonstrate this using the network policy that Table 6.1 shows, but other policies, such as the implicit policy governing access to security-sensitive VMM hypercalls also enable connections.

**Create the Root HSM Module.** The root module consists of data flows among the subjects and objects of root system policy. For a distributed system, the root policy is a network policy, and for a single host the root policy is a VMM or OS MAC policy. For a network policy, the root module includes a node for each IP address and a directed edge \((u,v)\) if node \(u\) is authorized to send a communication to node \(v\). For a MAC policy, a node is a subject or object label in the policy, and a directed edge \((u,v)\) is added for each subject \(u\) that can write to object \(v\) or each object \(u\) that can be read by subject \(v\). In both cases, the root module represents the authorized data flows among components.

**Create a Child HSM Module.** If one of the nodes in the root HSM module enforces its own MAC policy\(^4\) (VM or host), then a child HSM module will replace this node. The internal nodes and edges of a child HSM module will be created in the same manner as for the root module, but additional nodes and edges are necessary to connect this child to its parent.

To determine the connections, we use the policies that define external connections, such as the network policy of the VM or host\(^5\). For a particular network policy rule, an entry (exit) node is created in the child module and the child’s internal nodes representing its subjects with access to that network communication are connected to that entry (exit) node. In the parent module, a box is created that is connected to the entry (exit) node. Nodes in the parent module that want to communicate with the child must perform a call (box and entry node) and return (box and exit node).

Table 6.1 shows excerpts from the Secmark network [97] and SELinux MAC [111] policies associated to a web server VM. The Secmark network policy would cause entry and exit nodes to be created for communications from 130.203.32.56. The SELinux policy authorizes the internal node for the `httpd_t` subject to read from that entry node and write to that exit node. A call and return are created in the parent module for the box associated with this module to send and receive communications from the node.

---

\(^4\)A subnet policy is also possible below a root network policy.

\(^5\)VMs also have connections to their VMM through hypercalls, so we find that there may be multiple connection policies for a module.
Table 6.1: SELinux and iptables Secmark policy excerpts. The SELinux policy specifies the subjects that have access to network communication channels and to what channels, while the network policy specifies the authorized communication channels.

Figure 6.6: Representing information flows between nodes in the HSM model by: (a) merging all security requirements; (b) keeping the security contexts separate via separate flows; and (c) depending on nodes to demultiplex security contexts.

representing host 130.203.32.56. If 130.203.32.56 also has a module, then the returns and calls would be linked together in the parent module. There are different ways to do this, which we discuss below.

**Connecting through Parent Modules.** A question is how to connect an entry (exit) node in one module to a call (return) for another module through a parent. The naive way is to use the edge between hosts authorized by the parent’s network policy, as shown in Figure 6.6(a). However, as we can see, this would mix all of the data with all security labels in the single edge, so all the host transmitted data would reach the destination, not only the data with the expected label. The HSM model enables these flows to be separated by having a distinct flow from a host’s call to the destination’s return as shown in Figure 6.6(b).

The latter approach preserves the security context for the distinct communications, but the downside is that it may create many edges. When we consider a VM system, such as Xen, which has all networking redirected to a privileged VM, many edges with
the same destinations will be created to keep the security information distinct. Our aim is to multiplex security contexts where nodes are simply transporting information with distinct security contexts and demultiplex security contexts at nodes entrusted to do such enforcement. Figure 6.6(c) shows that a single path can be created between the two modules for transmitting data with multiple security contexts, but the flows from the receiver’s entry node are annotated with the demultiplexed security labels before being forwarded to internal module nodes. Such a mechanism corresponds to network demultiplexing in the kernel and/or privileged VM.

Multiplexing is possible when the exit and entry nodes correspond to the same remote endpoint. The semantics of this approach can be expressed in a more general way by associating edges with boolean expressions, as in Bebop [8]. Using that approach, the edge between the source and target modules in Figure 6.6(c) would carry the boolean expression $a \lor v \lor i$ and the entry node edges would be assigned properly. In a correct model, each demultiplexing assignment must satisfy the boolean program. We will explore this generalization in the future.

6.5.3 Constructing Graph-Cut Models

In this section, we construct an instance of the graph-cut model defined in Section 6.4.2 to evaluate system-wide integrity. The HSM model constructed in the previous section is the model’s directed data flow graph. Thus, one challenge is to determine the model’s security constraints: the system integrity lattice and the import mapping function. Typically, this information must be specified manually, but that is a difficult task for a distributed system. Information flow errors are resolved by mediators, so it is necessary to define the semantics of mediation for system-level information flows. Unlike programs, where mediation is associated with individual expressions, system mediation is associated with complex components (hosts, VMs, programs).

Typically, a system’s security-critical data, their security levels, and the relationships among those security levels are identified manually. While external sources of data are still easily identifiable from the network policies, the line is blurred between internal sources and sinks. Unlike a program, which has expressions that represent external sources and sinks (e.g., syscalls), systems often do not have explicit sinks and many sources are integrated into the system at deployment time. For example, a kernel is a source of high integrity, but it is also a sink reachable by all processes. Prior research has demonstrated methods to estimate which data is critical to which application or system, and we leverage such insights to provide an initial set of security constraints.
In general, such methods are conservative, which generally leads to false positives: non-critical imports identified as critical may be modified by untrusted processes, which leads to a false error, and the lack of relationship makes two levels incomparable, which leads to a false error if a flow between them is allowed.

**Identifying Imports.** Instead of thinking in terms of sources and sinks, we find that it is sufficient to define an import mapping function, which shows where security-critical data enters or is stored in the system, and the data flows will identify information flow errors automatically (see Section 6.4.2). However, identifying where data with security requirements is imported within an OS distribution is difficult as all the code and data appears as one unit. We process a set of *import rules* to identify common cases of data with security requirements. We leverage our prior work, where the integrity-critical labels for a process are those defined by that process’s MAC policy module [127].

For example, SELinux is a MAC OS policy that assigns standard labels to resources that are critical for the kernel like `memory.device_t`, `boot_t`, and `modules.object_t` to the files that represent main memory, boot files, and kernel modules respectively. Other labels that are critical for the integrity of a system are `policy.config.t`, `file.context_t`, and `shadow.t`, which are assigned to files that store MAC policy flows, MAC labels, and information about users and their privileges in a system respectively.

We automatically generate mappings for these labels in a distributed system. For example, for a particular VM, `webvm`, we would generate the following mapping values:

```plaintext
map(webvm,memory.device_t)=kernel-web
map(webvm,boot_t)=kernel-web
map(webvm,modules.object_t)=kernel-web
```

Where `kernel-web` represents the integrity level associated to the kernel of that VM.

Although we currently only support rules to infer mappings for SELinux policy, the model is not limited to SELinux. The idea is to extend our model with support for other policy languages. Identifying labels assigned to security critical resources for a particular policy is work that needs to be done only once and it can be reused multiple times in our model.

We find that a few key imports are necessary in pre-configured VMs. The idea is that policy designers can identify which application(s) they deem security-critical in a pre-configured instance. For example, for the web server VM, we only consider the integrity of the Apache process and the kernel\(^6\). The integrity of other processes is assumed to

\(^6\)Some MAC policies define different configurations for the same application (e.g., SELinux policy booleans), so we may have multiple types of data imported to one application. We will explore the
serve one or the other. We believe this reflects the goals of a pre-configured instance of a web server VM, and the same approach is used for the other VMs and hosts.

**Inferring the Lattice.** We must also determine the integrity relationships among the distinct imports to create a system-wide integrity lattice. For this we leverage prior work that aims to define a conservative lattice based on well-known relationships [130]: (1) kernel-application; (2) client-server; and (3) labeled communication. First, applications are lower integrity than the kernel and incomparable to each other by default. Likewise, all guest VM levels are lower integrity than privileged VM levels. Second, some processes or VMs may serve resources to clients, implying that clients depend on them. For example, a database serves to web applications and, indirectly, clients. Finally, labeled networking identifies collaboration between two parties (if bidirectional). Such communication represents a composition of their integrity levels, like LOMAC [15, 34], so MAC label of the channel will be a lower bound. In general, predicting integrity relationships between imported levels is the most difficult task to automate, so we envision that policy designers will have to refine and augment initial predictions.

Summarizing, the construction of a goal policy has two steps: identifying security levels and establishing a partial order among the levels. To determine the levels we first create an integrity level for each component, $C_i$, in the system policy and then add the levels defined by administrators via optional specification. To establish the ordering of the levels we use the hierarchical relation between components to establish a partial order between pairs of levels in the goal policy. Recall from Section 6.5.1 that each component has a hierarchical relation with another component, $(id, parent_id)$ in $C_i = (id, parent_id, MACpolicy)$. We also consider additional information specified by administrators. For example, for a VM `webvm` that depends on the privileged dom0 in a Xen environment, we generate the following order between the corresponding integrity levels:

$depends(dom0, webvm)$

A reasonable question is how to evaluate whether the mapping function, and the security goal we infer for a given system are a good approximation of the actual requirements of the system. There are two aspects to consider. First, there are resources in a host system that have standard semantics, for example, the boot files in a Linux installation. Our heuristics can properly address these cases and would be limited only for non-standard installations, for example, a system that labels boot files with something implications of this in Section 6.6.
different from the standard label. Second, some applications are individually designed and implemented to meet very specific functional requirements. Each one of these applications has its own semantics and we cannot create rules to address their integrity requirements. However, a good indicator regarding the security goal we construct is that it should have at least one integrity level per application. If a distributed application has several components with different semantics, administrators can specify such information, via optional specifications, and the security goal should have at least one level per component.

**Optimizations.** Since the HSM model explicitly identifies multiple instances of the same module, we take advantage of this feature to define the cases in which we can reuse the results of the `MediationResolution` program.

- **Bounded Inputs.** Given a VMs $vm_1$ that is an instance of a module with data flow graph $G = (V, E)$, an integrity lattice $L = (L, \preceq)$, a mapping function $M : V \rightarrow \mathcal{P}^L$, and all input levels are bound, we can run `MediationResolution` for this VM independent from the rest of the model.

- **Equivalent Mappings.** Given two VMs $vm_1$ and $vm_2$ that are instances of the same module, i.e., they both have the same data flow graph $G = (V, E)$, an integrity lattice $L = (L, \preceq)$, and two mapping functions $M_1 : V \rightarrow L \cup \{\bot\}$ and $M_2 : V \rightarrow L \cup \{\bot\}$, we say that $M_1$ and $M_2$ are equivalent ($M_1 \approx^* M_2$), if for every pair of lattice labels they map a node to, the labels are the same or hold the same relationship with every other label in the lattice, i.e., they dominate ($dom$), and are dominated by ($domby$) the same labels. Formally:

$$M_1 \approx^* M_2 : \forall v \in V. \ [M_1(v) = M_2(v) \lor M_1(v) \approx^* M_2(v)]$$

$$l_1 \approx^* l_2 : l_1, l_2 \in L \land \forall l \in L. [dom(l, l_1) \leftrightarrow dom(l, l_2) \land domby(l_1, l) \leftrightarrow domby(l_2, l)]$$

We simplified the definition of the mapping functions so they map a node to a unique lattice label. To extend the definition to have functions that map a node to a set of labels, we need to identify a graph isomorphism between the lattices induced by the mapping functions. Although the graph isomorphism problem is NP in a general case, in this particular case we have common labels, and we take advantage of this to search for a solution.

Notice that we still need to adjust the results, as we may need to map the level a mediator mediates for, from $M_1(v)$ to $M_2(v)$. 
Repairing Information Flow Errors. Once the imports and lattice are defined, a complete graph-cut model for the HSM data flow graph can be constructed. Using this model, information flow integrity errors can be identified as defined in Section 6.4.2. Such errors are caused when a path in the data flow graph \((u, v)\) enables data of level \(l_1\) from node \(u\) to reach a node \(v\) with data of level \(l_2\) and \(l_1 \not\preceq l_2\) in \(L\). To fix such an error, we perform a level manipulation at an edge along the path \((u, v)\) such that the data on that edge has some level \(l'\) such that \(l' \preceq l_2\) (i.e., the level of the data on that edge is at least \(l_2\)). This solves the error.

In practice, such errors are fixed manually by adding mediators, such as declassifiers for secrecy and endorsements for integrity \[161\]. For programs, such mediators apply to expressions, where they change its security level to resolve the errors (e.g., raise the expression result’s integrity). For systems, information flow integrity errors may be fixed in two ways: (1) by enforcing the integrity of data sent or (2) by upgrading the integrity of data received. First, we may identify that the sending subject has the responsibility to enforce integrity to protect its receivers. For example, a privileged VM may enforce isolation of its guest VMs, and a web application may prevent SQL injections from being forwarded to the database. Second, we may identify that the receiver must protect itself from lower integrity input. Rule C5 in the Clark-Wilson integrity model \[24\] requires a process that receives lower integrity data to upgrade or discard such data correctly. We note that attack surfaces \[54\] only refer to component entry points, but our approach also enables mediation of output. However, mediating output implies that the enforcing component (sender) knows the security requirements of the receiver.

6.5.4 Computing Mediation Placements

Given a graph \(G=(V,E)\) and a lattice \(L=\{L, \preceq\}\), there is a cut problem when there is an information flow error between two nodes with the import relation \(M\).

A cut problem set consists of the information flow errors found in an information flow analysis model, as defined in Section 6.4.2. King et al \[72\] only solved the cut problem set for a two-element lattice and showed that solving the cut problem set for a general lattice was an instance of the weighted cut-conjunction problem which has been shown to be NP-hard \[72, 70\]. They proposed a simple greedy solution to the problem that returns the union of the solutions for each individual cut problem.

To improve on the simple greedy approach, we use the insight that classical integrity encourages processes to upgrade input integrity to their level \[24\]. We note that each component has a limit on how high it may upgrade, called its maxraiselevel. The graph-
cut procedure produces a raiselevel for mediators which is at most maxraiselevel. This corresponds to the behavior of real systems. For example, we do not expect processes in a low integrity VM to mediate for a high integrity VM. We compute constraints based on the component a node belongs to: the upper bound of the integrity levels a node can mediate for cannot be higher than that of the component the node belongs to.

We use the raiselevels to define a property called Mediation Dominance. This property states that solving a cut problem in graph $G$ for level $l_1$ may solve any overlapping problem in the same graph $G$ for a level $l_2$ if $l_1 \preceq l_2$. The intuition behind this is that since $l_1$ is higher integrity than $l_2$, the semantics of mediation implies that any mediators that can mediate for $l_1$ can automatically mediate for $l_2$. Therefore, by solving the cut problems for $l_1$ before $l_2$, we get two advantages:

1. We may solve a smaller problem (size of the graph) for $l_2$ (compared to solving the problem for $l_2$ independent of $l_1$) since mediation dominance enables us to remove the mediators computed for $l_1$ and any flows they fostered before solving the problem for $l_2$.

2. If there is an overlap in the graph between the different cut problems of comparable lattice levels, then the size of ordered cut solution we propose is equal or smaller than the naive solution that solves each problem independently and computes a union of the individual solutions.

We propose an algorithm, MediationResolution, that solves cut problems following a topological sort of the lattice $\mathcal{L}$. Figure 6.7 presents the pseudocode for the MediationResolution algorithm. The algorithm receives a data information flow graph $G$, a lattice $\mathcal{L}$, and the mediation limits $MaxRaise$. The algorithm first sorts the labels in the lattice and creates its topological sort $TS$ (Line 2), determines the set of integrity levels that according to the lattice are not allowed to flow to the sink that is being analyzed (Lines 4,5), creates a SuperSource connected to the elements in that set (Line 6), creates a slice of the graph with only the nodes that can reach the sink (Line 7), adjusts weights to account for the fact that not all nodes can be mediators for a sink (Line 8), and finally computes the minimum cut for the particular SuperSource and sink that define one cut problem (Line 9), and finally adds the new set of mediators to the global set of mediators.

The running time of the algorithm is dominated by the time of computing the MinimumCut for each sink, (for every label in the lattice $\mathcal{L} = \{L, \preceq\}$). The running time of
1 \texttt{MediationResolution}(G, \mathcal{L}, \text{MaxRaise}) \{ \\
2 \quad \text{LS} \leftarrow \text{TopologicalSort}(\mathcal{L}) \\
3 \quad \textbf{for} (l \in \text{LS}) \textbf{do} \\
4 \quad \quad \text{Sources} \leftarrow \{ l_i \in \mathcal{L} \mid l_i \not\preceq l \} \\
5 \quad \quad G' \leftarrow \text{GraphCopy}(G) \\
6 \quad \quad \text{AddSuperSource}(G', \text{Sources}, \text{SuperSource}') \\
7 \quad \quad \text{SR} \leftarrow \text{SliceReachable}(G', \text{SuperSource}', l) \\
8 \quad \quad \text{AdjustWeights}(\text{SR}, l, \text{MaxRaise}) \\
9 \quad \quad \text{Mediators} \leftarrow \text{Mediators} \cup (l, \text{MinimumCut}(\text{SR}, \text{SuperSource}', l))
10 \}

Figure 6.7: Mediation Resolution Algorithm

the MinimumCut algorithm is $O(n^3)$, thus, the running time of the \texttt{MediationResolution} algorithm is $O(|\mathcal{L}|n^3)$.

Next we formalize the condition that a system meets when considering the semantics of mediators. Let $G_c$ be the extended version of the hierarchical graph that represents the test policy. We add a special set of nodes that represent the lattice levels and edges between the labels and the levels, $(l, v)$ and $(v, l)$, according to the mapping function $\text{map}$.

$$
G_c = (V_c, E_c) : \\
V_c = V \cup L, \\
E_c = E \cup \{ (l, v), (v, l) \mid l \in L, v \in V, \text{map}(v) = l \}\n$$

A mediator is a program expected to implement functions that sanitize the inputs it reads, and change their integrity level. This functionality is known as endorsement [102]. Let:
- $\mathcal{M}$ be a set of mediators that resolves all information flow errors we detect for a given distributed system.
- $n$ be the number of sinks in the system.
- Each set $Cu_i$ be the set of edges whose removal solves all information flow errors for $v_i$.

$$
\mathcal{M} = Cu_1 \cup Cu_2 \cup \ldots \cup Cu_n \\
\text{Sinks} = \{ v_1, v_2, \ldots, v_n \} \text{ is a topological sort of } \mathcal{L} \\
\text{Sources} = L
$$
The algorithm that computes the set of mediators for a given sink $v_i$ assumes that the mediators defined for sinks with higher integrity $1, 2, ..., i - 1$ are already in place $CS_{i-1} = \bigcup_{j=1}^{i-1} Cu_j$.

$$\forall v_i \in \text{Sinks},$$

let $CS_i = \bigcup_{j=1}^{i} Cu_j$

$$\forall u \in \text{Sources},$$

$$(u \hookrightarrow_{G_c-CS_i} v_i) \rightarrow (u \hookrightarrow_{L} v_i)$$

Where $u \hookrightarrow_X v$ means that there is a path from $u$ to $v$ in $X$ and $G_c-CS_i$ means removing all elements in $CS_i$ from $G_c$.

### 6.5.5 Generating Practical Integrity Policy

In this section, we develop a method that uses the mediation placement computed in the last section to generate a practical information flow integrity policy for the Flume Decentralized Information Flow Control (DIFC-Flume) model [76]. Specifically, we show a mapping method from the HSM model with its mediation placement to a DIFC-Flume policy. Based on this mapping, we prove that the notions of safety in both models are equivalent and generate a system-wide DIFC-Flume policy.

We use the HSM model and its mediation placement to automatically compute DIFC-Flume integrity labels and capabilities for objects and processes in a given Linux installation\(^7\). The DIFC-Flume model also includes the ability to create new tags and delegate tags among processes. We do not address these features specifically in this work, but will explore them in future work.

#### 6.5.5.1 Policy Generation

We use an HSM safe model to generate a DIFC-Flume integrity policy. At the crux of this translation is the semantic equivalence of mediator nodes and capabilities in the DIFC system. Mediator nodes are specially designed to remedy information flow errors. In the DIFC system, the same authority is given to specially designated subjects by granting them dual capabilities. These subjects need capabilities to endorse the inputs of lower integrity ($t^+$, where $t$ is the integrity level endorsed) and filter data of particular integrity.

\(^7\)Notice that the data we provide only addresses the generation of DIFC-Flume labels and capabilities, we currently do not provide additional information as to how modify programs to make them DIFC-aware. Recent work by Harris et al. [42] is a starting point for constructing DIFC-aware programs.
levels for outputs \((t^{-}, \text{where } t \text{ is the integrity level removed})\). In general, a process must be able to add and remove capabilities to raise and lower its integrity level, so instead of creating just endorsement or filtering capabilities, we must create dual capabilities \((t^{+} \text{ and } t^{-})\). That is, if \(v\) endorses or filters level \(l\) in the HSM model, we assign a dual capability \((l^{+}, l^{-})\) to \(v\)'s capability set in the DIFC Flume model.

Now, we define the translation from HSM nodes to DIFC nodes. We split the HSM nodes into two types, mediator nodes and non-mediator nodes.

We start by defining Incoming Level (IL), Outgoing Level (OL), Raise Level (RL), and mapping functions \(h2f\) and \(f2h\).

1. \(IL_{hsm}(n_i) \subseteq L\) for node \(n_i\) is the set of incoming integrity levels for node \(n_i\).

2. \(RL_{hsm}(n_i)\) is defined only for mediator nodes. It is the highest level that node \(n_i\) mediates to. We get this as a result of the graph-cut algorithm.

3. \(OL_{hsm}(n_i) \in L\) for node \(n_i\) is the highest integrity level of data that node \(n_i\) is guaranteed to provide. \(OL\) is computed differently for mediator and non-mediator nodes. For a non-mediator node \(n_i\), \(OL_{hsm}(n_i) = GLB(IL_{hsm}(n_i))\), where \(GLB(L_k \subseteq L)\) returns the greatest lower bound of labels in \(L_k\). For mediator node \(n_i\), \(OL_{hsm}(n_i) = RL_{hsm}(n_i)\).

4. Let \(h2f\) and \(f2h\) represent functions that convert HSM lattice levels\(^8\) to labels in the DIFC-Flume lattice and vice versa. We define \(haw\) to compute the mapping between the HSM and DIFC-Flume lattices in Section 6.5.5.2.

We translate HSM nodes to DIFC nodes as follows.

**Non-mediator Nodes.** Non-mediator nodes can only guarantee integrity levels lower than the level of any incoming flows. Such nodes are translated to DIFC-Flume nodes that are associated with a single DIFC-Flume lattice label that has no capabilities.

**Mediator Nodes.** For a mediator node \(n_i\), we assign \(h2f(RL_{hsm}(n_i))\) as the label to the corresponding DIFC-Flume subject. DIFC-Flume nodes selected as mediators also receive dual capabilities to allow them to receive data of lower integrity. Figure 6.8 presents the pseudocode of the procedure to compute DIFC-Flume integrity labels and capabilities. The function \textit{makecaps} creates a set of dual capabilities for a set of levels.

\(^{8}\)Actually, the HSM lattice is the integrity lattice [15] in the graph-cut model, see Section 6.4.2. We call it the HSM lattice distinguish it from the DIFC-Flume version.


```plaintext
1 computeDIFC(G(V, E), cutset, imports) {
2     G' ← G − cutset
3     ILG ← transitiveClosure(G', imports)
4     for (v in V)
5         do if (v ∈ cutset)
6             then
7                 Ds ← makecaps(ILhsm(v) ∩ {Li | RLhsm(v) ≤ Li ∈ L})
8                 Is ← h2f(RLhsm(v))
9             else
10                 Is ← h2f(GLB(ILhsm(v)))
11                 Ds ← {}
12     }
```

Figure 6.8: DIFC-Flume policy generation

greatest lower bound of the input data that reaches them, so that all input edges are legal.

The running time of the algorithm is dominated by the time of computing the transitive closure to propagate labels. The running time of the transitive closure algorithm is $O(n^3)$, thus the running time of our algorithm is $O(n^3)$ too.

### 6.5.5.2 HSM and DIFC-Flume Equivalence

Based on the translation above, we can prove that the notions of safety in both models are equivalent.

**Definition 6.5.1.** [HSM safe]. A system $S$ is safe under the HSM model if and only if for any flow $u → v$, $OL_{hsm}(u) ≤ OL_{hsm}(v)$ or $v$ is a mediator.

**Definition 6.5.2.** [Flume secure]. A system $S$ is secure under the Flume model (Flume secure) if and only if: (1) any label change for a process $p$ from $L$ to $L'$ adheres to the rule that $(L' − L)^+ ∪ (L − L')^- ≤ O_p$, where $O_p$ is the set of positive and negative capabilities assigned to $p$ and (2) a message can be sent from $p$ to $q$ only if $I_q ⊆ I_p$ or $I_q − D_q ⊆ I_p ∪ D_p$ where $D_q$ and $D_p$ are the dual capabilities of $q$ and $p$ respectively.

**Theorem 6.5.3.** [Flume-HSM equivalence]. A system $S$ is safe (for integrity) under the HSM model (HSM safe), if and only if, $S$ is secure (for integrity) under the Flume model (Flume secure).

**Proof.** We start by defining mapping functions from the HSM lattice to DIFC-Flume lattice and vice versa.

**Mapping Functions:** We first define mapping functions, $h2f$ and $f2h$, to map HSM levels into DIFC-Flume labels and vice versa. Note that we use the term HSM level,
as the term HSM label has a different meaning in our model (a label is a set of levels).

Figure 6.9 illustrates the mapping functions.

Given an HSM lattice $L_{hsm} = (L_{hsm}, \preceq_{hsm})$ let $\mathcal{H}2F : L_{hsm} \to L_{flume}$ be $\mathcal{H}2F(l) = \{ x \in L_{hsm} | l \preceq_{hsm} x \}$. We build the corresponding DIFC-Flume lattice $L_{flume} = (L_{flume}, \preceq_{flume})$ as follows:

- for every $l \in L_{hsm}$, we add $\mathcal{H}2F(l)$ to $L_{flume}$
- for every $(l_1, l_2), l_1, l_2 \in L_{hsm} \land l_1 \preceq_{hsm} l_2$, we add $(\mathcal{H}2F(l_1), \mathcal{H}2F(l_2))$ to $\preceq_{flume}$

Note that the relation can flow to $\preceq$ corresponds to the relation superset $\supseteq$ on sets as defined by the DIFC-Flume model.

Given a Flume lattice $L_{flume} = (L_{flume}, \preceq_{flume})$ let $\mathcal{F}2S : L_{flume} \to L_{hsm}$ be $\mathcal{F}2S(\{l_1, ..., l_n\}) = \text{concat}(l_1, ..., l_n)$.

We build the correspondent HSM lattice $L_{hsm} = (L_{hsm}, \preceq_{hsm})$ as follows:

- for every $l \in L_{flume}$, we add $\mathcal{F}2S(l)$ to $L_{hsm}$
- for every $(l_1, l_2), l_1, l_2 \in L_{flume} \land l_1 \preceq_{flume} l_2$ (or equivalently $l_1 \supseteq l_2$), we add $(\mathcal{F}2S(l_1), \mathcal{F}2S(l_2))$ to $\preceq_{hsm}$

We have defined the translation from HSM nodes to DIFC-Flume subjects and objects in Section 6.5.5. The reverse translation from DIFC-Flume subjects and objects to HSM nodes works the same way. Definitions 6.5.1 and 6.5.2 specify safety requirements for each model.

We can now define the safety-equivalence property between the HSM lattice and DIFC-Flume.

**Theorem 6.5.3** A system $S$ is safe (for integrity) under the HSM model (HSM safe), if and only if, $S$ is secure (for integrity) under the DIFC-Flume model (Flume secure).

**Proof of Theorem 6.5.3:** We first prove: (A) if a system is HSM safe then it is Flume secure, and then (B) if a system is Flume secure then it is HSM safe.

**A. HSM safe to Flume secure.** Let flow $p \to q$ be HSM safe. This gives us two cases.

[Case 1]: $q$ is a non-mediator node. Therefore, $OL_{hsm}(p) \preceq OL_{hsm}(q)$. From the translation rules to DIFC-Flume this would imply that $\mathcal{H}2F(OL_{hsm}(p)) \supseteq \mathcal{H}2F(OL_{hsm}(q))$. This would be a safe flow in DIFC-Flume as well.
Figure 6.9: HSM to DIFC-Flume and DIFC-Flume to HSM Mapping Functions.

[Case 2]: Node q is an mediator node and $OL_{hsm}(q) \preceq OL_{hsm}(p)$. When we translate this to DIFC-Flume, we ensure that the dual capabilities of q, $D_q$, contains tags for tags $h2f(OL_{hsm}(q))$ and for integrity levels dominated by $h2f(OL_{hsm}(q))$ (this includes tags for $h2f(OL_{hsm}(p))$, since $OL_{hsm}(q) \preceq OL_{hsm}(p) \in L_{hsm}$) in the DIFC-Flume lattice. This ensures that this flow is secure under DIFC-Flume.

**B. Flume secure to HSM safe.** A system S is Flume secure if and only if all allowed process label changes are safe and all allowed messages are safe.

[Case 1]: All allowed process label changes are safe. This means that any process q can change its label only within the range of its capabilities $D_q$. We map processes with capabilities in Flume to mediators in the HSM model with $RL_{hsm}$ equal to the union of all tags in the dual capability set $D_q$. Mediators are safe nodes in the HSM model as they automatically downgrade higher integrity levels to its raiselevel and endorse lower integrity levels to its raiselevel.

[Case 2]: All allowed messages are safe. This means that for all messages from p to q: $I_q - D_q \subseteq I_p \cup D_p$.

2.1: Assume $D_p = D_q = \{\}$ thus $I_q \subseteq I_p$. That means $f2s(I_q) \succeq f2s(I_p)$. This is a safe information flow in HSM.

2.2: Assume $D_p \neq \{\} \land D_q = \{\}$ thus p will always run with integrity $I_p$ or higher. Since $D_q = \{\}$, then $I_q \subseteq I_p$ in the extreme case. Therefore, $f2s(I_q) \succeq f2s(I_p)$. This is a safe information flow in HSM.

2.3 and 2.4: Assume $(D_p = \{\} \land D_q \neq \{\})$, or $(D_p \neq \{\} \land D_q \neq \{\})$. In both cases we map q to a mediator in the HSM model. Mediators are safe nodes in the HSM model as they automatically downgrade higher integrity levels to its raiselevel and endorse lower integrity levels to its raiselevel.
6.5.6 Implementation

This section illustrates the building of the HSM data flow graph with a system running Xen, the VMM enforcing a XSM/Flask policy, and dom0 and guests enforcing SELinux policies and iptables with the secmark extension.

We build the graph in three main steps: (1) create VMM module, (2) create VM modules and boxes, and (3) create interVM flows. We may perform a fourth step in some cases: (4) create program modules and boxes. We explain each of these steps in the following paragraphs.

1. Create VMM Module. The VMM module is the root of a VM-system. To build this module, the system builder calls the XSM/Flask policy parser, the parser creates the graph representation of the policy.

2. Create VMs. For each one of the VMs defined in the VM system, the system builder identifies the associated SELinux and network policies. To create the module and box associated to a given VM, the builder executes the following operations.

   • It checks if a module has already been created for the SELinux policy. If it has then it gets the id associated to that module, otherwise it creates the module. The algorithm to create the graph representation of the policy is well known.

   • The builder uses the network policy to determine the set of input and output ports associated with a given VM. Table 6.1 shows excerpts of the network...
Rule 1 (in the source):
```
iptables -t mangle -A OUTPUT -p tcp --dport 3306 -s 130.203.32.56 -d 130.203.32.61
-j SECMARK --selctx system_u:object_r:mysqld_client_packet_t:s0
```

Rule 2 (in the target):
```
iptables -t mangle -A INPUT -p tcp --dport 3306 -s 130.203.32.56 -d 130.203.32.61
-j SECMARK --selctx system_u:object_r:mysqld_client_packet_t:s0
```

Table 6.2: Related Network Policies. The first rule specifies the labeling of packets sent by the web server to the database server, while the second rule specifies the labeling of packets as received by the database server from the web server.

and SELinux policies associated to a web server VM. While the network policy indicates how network packets will be labeled on input/output, the VM policy authorizes the httpd_t subject to read and write packets over the correspondent port.

- The system builder also creates an input and an output kernel ports, abstract representations of the VMM resources a VM has access to.
- The system builder creates a box at the VMM level that associates a VM with the correspondent module and set of ports.

3. Create interVM Flows. The builder creates network and VMM based flows. To do so, it executes the following operations.

- It creates network flows based on the network policies. For a flow to be created an output rule and an input rule in two VMs must be related. Table 6.2 shows an example of such rules.

- It creates VMM flows based on the XSM/Flask rules. For a flow from VMs $V_1$ to $V_2$ to be created, the XSM/Flask policy must authorize a read-like operation for the type associated to $V_2$ on resources labeled with the type associated to $V_1$, or a write-like operation for the type associated to $V_1$ on resources labeled with the type associated to $V_2$.

- A global flow involves the output port in the module associated to the source VM, call and return nodes in the VMM module, and the destination port in the module associated to the target VM. In the case of Xen, since processes in dom0 may govern communications between VMs, a connector node may need to be further refined to represent a connection to dom0 via the kernel port, an abstract representation of the communications via VMM resources. This type of flow would involve the output kernel port in the module associated to
the source VM, the correspondent call and return nodes in the VMM module, the input kernel port in the module associated to dom0, the internal flows in dom0 (the flows between the kernel ports and the program that enforces a policy), the output kernel port in dom0, a second pair of return and call nodes in the VMM module, and finally the input kernel port in the target VM.

4. Programs may internally enforce their own policy. To represent these programs in our model, we follow a similar procedure. We replace the original node with a box and a module. The module has the policy information flow graph that represent the operations authorized by the program policy. The box is added to the VM module and points to the module associated to the program. Finally we replace the original edges with edges towards or from the entry/exit ports of the program module.

6.6 Evaluation

In this section, we present the results of running a prototype implementation of the Proactive Integrity Methodology on the web application system presented in Section 6.2.1.

6.6.1 Prototype System

The prototype has an Eclipse front-end that provides access to: (1) parsers to build system-wide HSM data flow graphs for XSM/Flask [26], SELinux [111], Labeled IPsec [61], and iptables with the Secmark extension [97]; (2) rule bases and manual interfaces to collect security constraints to construct information flow analysis models; (3) the Lemon graph library [115] to compute graph-cuts; and (4) a module for generating DIFC-Flume policies. All of our programs are written in C or Prolog: 4303 lines of code for the Bison based parser, 4808 in C/C++, and 405 in Prolog.

The prototype system uses the size of a system’s attack surface to estimate the cost of mediation. An attack surface is the set of interfaces accessible to an adversary in a deployment [54]. When a node receives an input that would cause an information flow error, it must mediate that input, and the cost of mediating that input depends on how many interfaces may receive that data. The number of interfaces accessible to an attack surface may be estimated using static or runtime analysis. A MAC policy authorizes any interface in a running program to access any authorized objects, so a static approach would include all program interfaces (e.g., system calls). However, not
Figure 6.11: Representation of interfaces. Program $s_1$ has two interfaces, $int_1$ and $int_2$, to access objects of labels $t_a$, $t_b$, and $t_c$. Level $l_1$ propagates to $s_1$ via its interface $int_1$. Every interface may receive untrusted data, so we use a runtime analysis to record the mapping between the program interfaces and the labels of data they actually access. We note that static evaluation provides an upper-bound cost estimate\(^9\), runtime evaluation provides a lower-bound cost estimate. One experiment will explore the difference between static and runtime estimates to examine how much the mediation cost of generic policies may be reduced for a deployment.

### 6.6.2 Experimental Setup

We evaluate a version of the example system described in Section 6.2.1. The focus is on the web server host, which runs Xen VM system 4.1, consisting of domain 0 privileged VM, web server VM including an Apache web server and web application, and MySQL database VM. Also, there is a separate user VM that is not part of the web application (i.e., they are mutually distrustful). Each VM runs the Ubuntu 2.6.31-23-generic kernel. The Xen hypervisor enforces XSM/Flask policies [26], and each of the VMs enforce SELinux policies [111]. While all of the OS policies are SELinux, they are independent in the sense that each policy supports a distinct set of applications and these policies do not refer to the interactions among VMs. Also, each VM runs an iptables firewall with the Secmark extension [97] to govern network communications. We assume that secure communication (e.g., IPsec or SSL) is used to protect any channel that carries data between hosts on a labeled channel. Finally, the other hosts are clients of the web application and an external network server for processing DNS requests. All clients are assumed to be untrusted. In addition we explore results for two variations of this system. The first one integrates a trusted DNS server, while the second one explores how the system changes if administrators label some resources based on their experience rather than letting the runtime infer the label for them.

\(^9\)We limit the number of static interfaces to those seen in the runtime trace to reflect potentially unnecessary access to objects in the deployment.
Static vs. Runtime System Representation. MAC policies are defined in terms of programs, but we want to mediate at a finer granularity where possible. To do so we use the concept of read-like interfaces. A read-like program interface is a system call in the program that causes a flow of data into the program, such as `read()` . We use the number of read-like interfaces in a program as an approximation of the effort required to protect that program by filtering each of these interfaces. A program with a higher number of read-like interfaces requires more effort to protect it. Thus, the goal is to determine the access of programs’ interfaces to objects that may be modified by an attacker. In other words, where possible we choose to mediate at a finer granularity: at the interfaces. We must extend the graph to represent program interfaces, and model the correspondence between an interface and its program.

Figure 6.11 illustrates the process. We add a node for each relevant interface associated with a subject. Each of these nodes has an edge \((i_v, v)\) to its subject node. Also, we substitute each edge \((u, v)\) with an edge \((u, i_v)\), where \(i_v\) is an interface of \(v\) that reads data of node \(u\).

To determine the interfaces, we leverage prior work [150] that developed a runtime logging method to determine for any given program the set of input interfaces (i.e., system calls to read data) such program uses and the MAC labels assigned to the input data. Any runtime approach is inherently incomplete, however, this approach is more refined to estimate the cost of mediation than considering all nodes to have the same number of interfaces as we need to defend input interfaces that receive untrusted data. For example, defending a program with 200 interfaces that read untrusted data should be more expensive that defending a program with 2 interfaces. Additionally, this approach gives us a hint about how a least privilege policy looks like as it registers the interfaces that are actually used, instead of all interfaces a program may use. The current runtime tool does not yet log interfaces for interpreted programs, so in our evaluation we assign them a cost higher than all the programs for which we have number of interfaces. In this way we favor selection of these nodes as mediators.

Specified Mediators. Administrators may specify mediators, programs that are trusted to provide a defined integrity level. These mediators may also be the result from a previous cut. We represent them as multiple nodes, each one providing one of the levels defined by its specification. Figure 6.12 shows an example. The first part of the mediator specification, \(F(s, t1) = \{(s, t1), (l1)\}\), means that mediator \(s\) will transmit only data with integrity level \(t1\) over the edge \((s, t1)\). To represent this specification, we add the node \(s\#l1\) and map it to a source that provides data with level \(l1\) over the edge
Figure 6.12: Representation of mediators for the mincut algorithm. We represent HSM mediators as mappings to nodes in the graph, one for each level the enforcer provides: \( l_1, s\#l_1 \) for \( t_1 \), and \( l_2, s\#l_2 \) for \( t_2 \).

<table>
<thead>
<tr>
<th>VM</th>
<th>Rules</th>
<th>Sub</th>
<th>Static O+I</th>
<th>Static E</th>
<th>Runtime O+I</th>
<th>Runtime E</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbsrv</td>
<td>293639</td>
<td>71</td>
<td>1974</td>
<td>17071</td>
<td>1904</td>
<td>7686</td>
</tr>
<tr>
<td>dom0</td>
<td>320554</td>
<td>79</td>
<td>2182</td>
<td>20575</td>
<td>2104</td>
<td>8812</td>
</tr>
<tr>
<td>uservm</td>
<td>312026</td>
<td>72</td>
<td>2355</td>
<td>18693</td>
<td>2284</td>
<td>9063</td>
</tr>
<tr>
<td>webserv</td>
<td>321798</td>
<td>77</td>
<td>2394</td>
<td>20752</td>
<td>2318</td>
<td>9544</td>
</tr>
<tr>
<td>VMM</td>
<td>573</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dnsrv</td>
<td>293403</td>
<td>69</td>
<td>2008</td>
<td>16861</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>total</td>
<td>1,541,993</td>
<td>394</td>
<td>10913</td>
<td>93952</td>
<td>8610</td>
<td>35105</td>
</tr>
</tbody>
</table>

Table 6.3: Problem Size: number of policy rules, subject labels (S), object labels and interfaces (O+I), and edges (E) for static and runtime cases.

that connects \( s \) to \( t_1 \).

**Costs.** We are interested on determining the set of mediators that minimizes developers’ and administrators’ efforts in order to fix a system that is not safe. We approximate this cost with the number of interfaces that a program must mediate, that is, the interfaces that receive data with lower integrity than the level the program is supposed to provide. We approximate the filtering cost for programs for which we do not have interface data registered based on their number of lines of code. The intuition behind this is that the longer the program the higher the number of system calls that it may execute, thus the number of interfaces. If we cannot determine the number of lines of code either, then we assign the highest possible value. This assignment should avoid cuts at these edges, except in cases for which there is no other solution.

6.6.3 Experiments and Results

We run three experiments. The first experiment, E1, computes the mediation cost for the web server host alone (i.e., not including the DNS server) using the static and dynamic attack surface estimate. This experiment provides a baseline for the expected mediation cost for the core system. The second part of the same experiment explores the lower-
<table>
<thead>
<tr>
<th>Sink</th>
<th>Static Web Server</th>
<th>Run Time Web Server</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sub</td>
<td>U Sub</td>
</tr>
<tr>
<td>kernel-dnssrv</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>dnsdata</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>kernel-dom0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>kernel-dbsrv</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>dbdata</td>
<td>27</td>
<td>3</td>
</tr>
<tr>
<td>kernel-uservm</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>total</td>
<td>124</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 6.4: Mediation costs (edges and subjects) by cut problem for the Web Server Case.

<table>
<thead>
<tr>
<th>Sink</th>
<th>Static Trusted DNS</th>
<th>Run Time Trusted DNS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sub</td>
<td>U Sub</td>
</tr>
<tr>
<td>kernel-dnssrv</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>dnsdata</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>kernel-dom0</td>
<td>63</td>
<td>12</td>
</tr>
<tr>
<td>kernel-dbsrv</td>
<td>59</td>
<td>0</td>
</tr>
<tr>
<td>dbdata</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>kernel-uservm</td>
<td>51</td>
<td>1</td>
</tr>
<tr>
<td>kernel-websrv</td>
<td>55</td>
<td>3</td>
</tr>
<tr>
<td>webdata</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>total</td>
<td>284</td>
<td>73</td>
</tr>
</tbody>
</table>

Table 6.5: Mediation costs (edges and subjects) by cut problem adding a Trusted DNS Case.

<table>
<thead>
<tr>
<th>Sink</th>
<th>Static WebResources</th>
<th>Run Time WebResources</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sub</td>
<td>U Sub</td>
</tr>
<tr>
<td>kernel-dom0</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>kernel-dbsrv</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>dbdata</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>kernel-uservm</td>
<td>30</td>
<td>6</td>
</tr>
<tr>
<td>kernel-websrv</td>
<td>31</td>
<td>7</td>
</tr>
<tr>
<td>webdata</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>webtypes</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>total</td>
<td>132</td>
<td>54</td>
</tr>
</tbody>
</table>

Table 6.6: Mediation costs (edges and subjects) by cut problem identifying web resources.
Table 6.7: DIFC-Flume Dual Capabilities. Subjects are grouped per number of capabilities.

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Web Server</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>393</td>
</tr>
<tr>
<td>Runtime Web Server</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>262</td>
</tr>
<tr>
<td>Static Trusted DNS</td>
<td>15</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>32</td>
<td>11</td>
<td>557</td>
</tr>
<tr>
<td>Runtime Trusted DNS</td>
<td>10</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>14</td>
<td>7</td>
<td>302</td>
</tr>
<tr>
<td>Static Web Resources</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>32</td>
<td>0</td>
<td>453</td>
</tr>
<tr>
<td>Runtime Web Resources</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>18</td>
<td>0</td>
<td>281</td>
</tr>
</tbody>
</table>

Table 6.3 shows the size of the resulting problems for static and runtime estimates for the web server case. Note that we do not cut on VMM (inter-VM) edges, so no data is given for those. In general, it is possible to add inter-VM guards (e.g., cross-domain solutions), but we focus on the typical case where processes must defend themselves. The size of the problems does not change for the next experiments.

In the second experiment, E2, we add responsibility to the DNS server to uphold its integrity to determine the impact on the overall system. One question is what impact expanding the system to include the DNS server will have on the overall mediation cost, and another question is whether and how the mediation cost will shift to the DNS server.

In the third experiment, E3, we manually labeled some resources to compare the cut generated from a more expert labeling to the cut generated by the runtime analysis. We identified the resources that belong to the web server and assign them a particular integrity level, based on our experience rather than expecting the runtime to compute the label for them. For example, we expect http log files to be lower integrity than the web server itself, as not only the web server but any of the scripts it starts may write to them. Also, we expect some web configuration files to be the same integrity that the web server itself. We want to know how much the cut changes and whether there are additional interfaces that need to be mediated.

We found that the rules we defined to label standard resources in the system enable us to configure many, but not all, of the necessary security constraints for the example system. First, as described in Section 6.5.3, we need to identify the applications to protect in each VM or host. Second, we manually identify client-server relationships between VMs to guide the inference of the integrity lattice. For example, the MySQL
database serves requests for the web applications. While we aim for lattice and import rules that apply to many applications, there may be specializations that limit their value in other configurations. We will explore this in future work.

Results. We guided our analysis of the results by looking for answers to the questions we identified in Section 6.2.1. Below we use our model to answer these questions. Tables 6.4, 6.5, 6.6, and 6.7 show the results for the experiments.

(Q1) What is the minimum cost of protecting a deployment?
(Q2) How far are we from functional policy?
(Q3) How do different systems compare? What is the system behavior if I install a program I do not trust? Or a new component?
(Q4) What is the policy that identifies the interfaces where integrity protection is needed?

(Q1) For the first experiment, E1, the static approach requires a cut of 124 subjects where 50 are unique, and 7,460 edges where 2,018 are unique (Table 6.4). As the effort to mediate a program should apply across deployments to some extent, we modified the MediationResolution program to reuse subjects from prior (and higher integrity) cuts by default. Tables 6.5 and 6.6 identify the minimum attack surface to protect for experiments E2 and E3, respectively. Some of the cuts involve filtering a significant number of interfaces. For example, the cut to protect the integrity of the kernel in dom0, kernel-dom0, for the static case in experiment one, involves 1069 interfaces, associated to 32 subjects. These numbers may seem high but they represent the effort that takes to provide filtering for all adversarial flows to the resources that must be protected.

(Q2) We evaluated the same systems using the runtime attack surface estimates, this estimate corresponds to the permissions that the applications actually used during an observation period. The number of edges are much fewer in every case. For example, for the first experiment (E1) the resulting cut is significantly smaller: 88 subjects where 35 are unique and 488 unique edges. The lower-bound to achieve integrity protection requires defense of about 500 program interfaces, which is a non-trivial effort. We found that the major cause of the difference between the static and dynamic cuts is due to attributed-based permissions. Attributes are an easy way to express aggregate access by associating a permission to a set of labels, but few of the aggregate permissions appear to be used in this deployment. Based on this experiment, we envision that measures need to be explored to refine the use of attributes for deployments. The same conclusion applies for the other two experiments, E2 and E3.

(Q3) We used the runtime interface mapping to evaluate the impact of including a
Table 6.8: Execution Times, all in seconds. HSM: Construction of the HSM model, GCM: Constructions of the Graph cut model, Cuts: running the MEDIATION RESOLUTION program, DIFC: Computing DIFC-Flume labels and capabilities.

Trusted DNS server in the distributed system evaluation (E2). While adding the DNS server had little impact on the mediation cost, it reduced the number of DIFC capabilities for the web server and database subjects as shown in Table 6.7. The subjects with 10 and 11 capabilities are in domain 0 and the DNS server now, and the 18 subjects with 3-5 capabilities in the static case have been reduced to 2-4 capabilities. Thus, while it is still necessary to provide mediation for the same number of interfaces, the capabilities that must be used is reduced.

We also evaluated the impact of manually labeling some resources (E3). We labeled resources that belong to the web server: web server configuration files with the same integrity of the web server and web server log files with a lower integrity. The mediation cost is higher than that of the baseline as access to the labeled resources must be mediated. The new subjects that must implement filtering interfaces (new subjects in the cut) are associated with http helper applications: `httpd_rotate_logs_t`, `httpd_suexec_t`, and `httpd_sys_script_t`. These subjects did not appear in the runtime analysis of the baseline case because they did not request access to the objects we labeled at any point during our observation period. The number of capabilities increased, but all the processes with a higher number of capabilities belong to dom0, as in the baseline case.

(Q4) The cut set defines the programs that need to implement integrity filters for a particular deployment to be safe. Our cut-method also identifies the particular interfaces associated to those programs that need to be mediated. Table 6.7 summarizes the results of the generation of DIFC-Flume policies, given the mapping we defined in Section 6.5.5 for each experiment. In all cases the subjects with higher number of capabilities belong to dom0. For the second experiment, E2, subjects in the DSN server also hold a high number of capabilities. This is expected as the trusted DSN provides high integrity data and needs to communicate with processes in the other VMs. Our results demonstrate that we can generate DIFC-Flume policy without additional manual specification.
**Performance.** Our experiments were run on a machine with a 2.3 GHz Intel Xeon with 8GB of RAM. Table 6.8 shows the execution time for the experiments by the cost of computing methodology tasks. The times reflect an average over 10 runs. Note that the HSM data flow graph only needs to be built once per analysis session, and applies for both static and runtime analyses. The execution times vary with the number of edges and the size of the cut, as expected. The time of computing the cuts is dominated by the number of labels in the lattice, since each corresponds to a problem that must be analyzed, and the computation of the minimum cut for each problem, as shown in Figure 6.7. The DIFC-Flume label computation is dominated by the transitive closure, as shown in Figure 6.8.

We note that optimizations are possible that partition the system into pieces that may be processed independently, as introduced in Section 6.5.3. For example, since the user VM is supposed to isolated from the rest of the system by the privileged VM, its inputs are independent of the rest of the system. However, the privileged VM still must be able to protect itself and others from user VM outputs. Thus, the user VM mediation would be computed first, then the rest of the system. The insight is that the use of mediation to protect another component may also be useful to create such partitions. But we have not implemented these improvements yet, they are part of our future work.

### 6.7 Conclusion

We proposed the Proactive Integrity methodology to help OS Distributors and systems administrators configure systems according to integrity requirements in their target deployments. Our methodology enables policy developers to determine a near-minimal mediation cost for defending their systems. We do so by designing a heuristics to infer automatically security goals and leveraging the Hierarchical State Machine (HSM) model to compose multiple independently-developed policies into a concise yet expressive representation. We use the model to evaluate safety of represented systems. We also leverage the minimum-cut graph problem to compute a near optimal solution that aims to minimize the effort that administrators and developers need to make to fix non-compliant systems.

Our heuristics to infer security goals are inherently limited for specialized deployments. Currently we enable administrators to modify the security goals we generate, but we need to provide administrators a tool to ease that task considering the size of the security goals that may emerge in distributed systems. Also, we proposed some possible
optimizations of our implementation to make it more scalable and efficient. We plan on continuing our exploration in this area.

Finally, we proved that a secure configuration in our model is equivalent to a secure configuration in the DIFC-Flume model. Based on the equivalence, we developed a method to generate DIFC-Flume policies automatically.

In summary, we developed a prototype that demonstrates the practicality of a comprehensive policy design, and the advantages of the methodology to evaluate target systems and explore how modifications of the system, such as adding a new application or virtual machine, affect the integrity of the system. The end result is a system-wide information flow policy that is compliant with administrator’s security goals and minimizes the attack surface for particular deployments.
Chapter 7

Conclusions and Future Work

7.1 Conclusions

We developed services to help administrators configure and deploy distributed MAC systems with independently-developed components, namely, programs, operating systems, and possibly virtualized environments, to meet security goals, i.e., to defend security-relevant resources from external attacks.

One of the driving ideas behind this thesis is to reduce the burden that administrators must assume to configure MAC systems. We consider our results to be appropriate steps in that direction. The configuration task is challenging because of the number of components, the multiple ways in which they may interact, and even in some cases the complexity of each component. Briefly, our services help administrators by mostly-automatically building an analytical model of all the components in a distributed MAC system and their interactions, using the model to evaluate a security goal, and resolving systems that do not meet an established security goal.

We represent the problem of evaluating whether a MAC system, single or composite, meets a security goal as a compliance problem. Informally, a compliance problem evaluates whether a system policy (a policy that describes a system’s behavior) meets the requirements defined by a goal policy (a policy that describes accepted behavior). Notice that we assume that each MAC component has an enforcement mechanism that guarantees that the system only behaves within the boundaries defined by the MAC policy. We represent the system and the goal policies with directed graphs and define compliance based on that representation. The arrows in the case of the system policy represent the information flows that the system policy allows. The arrows in the case of
the goal policy represent the direction in which information can legally flow. Additionally, we define a partial mapping function, $map$, to relate nodes in the system policy to nodes in the goal policy, i.e., to identify security relevant resources. Informally, a system is compliant with a security goal, given a mapping function $map$, if for all pair of nodes that represent security relevant resources in the system policy, $n_1, n_2$, if there is a path from $n_1$ to $n_2$ in the system policy then there is a path from $map(n_1)$ to $map(n_2)$ in the goal policy.

Although the compliance problem has been studied before, we addressed two unsolved challenges: how to automate the construction of specifications and the generation of resolutions for non-compliant systems.

- To automate the construction of the specification of the problem we first leverage the hierarchical graph representation of the hierarchical state machine (HSM) model, which allows the representation of multiple components, exposes their layering, and multiple interaction mechanisms. Second, we developed heuristics to infer security goals and mapping functions.

A reasonable question is how to evaluate whether the mapping function and the security goal we infer for a given system are a good approximation of the actual requirements of the system. We found that there are two types of components to consider: standard and specialized. Standard components have previously developed policies and well-known functional requirements. For example, standard Linux services. Specialized components are developed or customized to meet specific functional requirements. Our heuristics can properly represent the requirements of the standard components. On the other hand, specialized components, like in-house developed programs, have their own semantics, and we cannot create rules to infer their requirements. However, we know that our goal must have at least one security level per specialized component. Notice that our work favors standard installations because they make it possible to automate configuration tasks.

- To automate the generation of resolutions for non-compliant distributed MAC systems we leverage the technique to solve the minimum-cut problem in directed graphs. However, our problem is an instance of the weighted cut-conjunction problem, which is proven to be NP-hard. Thus we implement a greedy approach that computes minimum-cut sets per resource with security constraints, and unions the solutions.
Another question to consider is what the appropriate cost function should be to best approximate the actual cost of mediation. We explored several cost functions, choosing one that approximates the effort that programmers would need to make by the number of input interfaces that a program uses to read untrusted resources. This approach assumes that the cost of implementing procedures to sanitize inputs is the same for all input interfaces, however, further refinement may be possible.

Finally, we compare the cost of protecting a system deployed with currently available MAC policies against the cost of protecting a system that aims for least privilege. We approximate the latter with a representation that only includes information flows that actually happen when programs are run with their corresponding testing suites. Our results are incomplete because they are based on a run time analysis, but they give us a hint about the difference in the values. We found that there is a large gap: the cost of mediating for the former representation is about 5 times larger than the cost of mediating for the system that approximates the latter. This result suggest that tightening of the available MAC policies is possible.

Our approach demonstrates that it is possible to use already available information with little input from administrators to automate currently manual tasks, thus easing the burden of configuration and deployment on system administrators. We hope that our approach provides other researchers with a vision towards developing services that make the construction of systems that meet security requirements more practical.

### 7.2 Future Work

Considering the advantages of MAC systems to enforce security requirements our goal is to keep exploring the development of services to make them more practical to deploy. There are several options to explore.

- Our approach demonstrates that it is possible to use already available information to automate the specification of compliance problems. We believe we can use the same approach, asking administrators to manually define a small amount on information, to automate other tasks such as the generation of policy for a new component in an already evaluated distributed MAC system.

- Also, we studied some possible optimizations of our implementation to make it scalable and efficient. We plan on continuing our exploration in this area so we can effectively represent, evaluate, and explore larger deployments.
• In addition, we want to explore how to use our services to tighten an original set of MAC policies while keeping necessary functionality. Currently, functional requirements are not explicit, so we want to explore the feasibility of automatically generating functional requirements. Based on this approach we could adjust the original policies, probably downloaded from standard distributors, to better approximate least privilege for a particular deployment.

• Finally, we believe our approach can help administrators of utility computing environments build secure configurations. Utility computing systems run pre-configured instances on known utility platforms. Therefore, they could use our approach to evaluate and adjust their configurations so their systems satisfy a defined set of security requirements. Later, administrators can simply deploy previously evaluated configurations as needed, in the same way they do now, but with security guarantees. Notice that utility computing validates our argument of favoring installations with standard components when possible as they make it possible to automate many configuration tasks.
Bibliography


Vita

Sandra Julieta Rueda Rodríguez

Education

- Ph.D., Computer Science and Engineering. August 2011.
  The Pennsylvania State University. University Park, PA, USA

  Los Andes University, Bogotá, Colombia

  Los Andes University, Bogotá, Colombia

Awards


Experience

- Research Assistant
  Department of Computer Science and Engineering.
  The Pennsylvania State University.

- Instructor
  Department of Systems and Computing Engineering.

- Information Technology and Systems Administrator
  Vision Tech Colombia and Latin Net S.A. (Colombian Companies)