CERIAS Tech Report 2011-26
Accommodative Mandatory Access Control
by Jacques Daniel Thomas
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

Graduate School ETD Form 9 d
(Revise 1 /0) d d d d d

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared d

By d Jacquesm Daniel Thomasm

Entitled
Accom odative Mandatory Access Controlm

For the degree of d Doctor of Philosophym

Is approved by the final examining committee: d

Prof. Jan Vitekm d
Chair d

Prof.m Eugene Spaffordm d

Prof. Patrickm ugsterm d

Prof. Ninghui Lim d

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of d Purdue University's "Policy on Integrity in Research" and the use of copyrighted material. d

d d d d

Approved by Major Professor(s): Prof. Jan Vitekm d

d

Approved by: d Prof. Sunil Prabhakar /m Dr. William J. Gormanm 12/05/2011 d
d d d d d Head of the Graduate Program d d d d d Date d

Graduate School Form 20
(Revised 9/10)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:
Accommodative Mandatory Access Control

For the degree of Doctor of Philosophy

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Executive Memorandum No. C-22,* September 6, 1991, *Policy on Integrity in Research.**

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Jacques Daniel Thomas
Printed Name and Signature of Candidate

12/05/2011
Date (month/day/year)

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/c_22.html

ACCOMMODATIVE MANDATORY ACCESS CONTROL

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Jacques D. Thomas

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2011

Purdue University

West Lafayette, Indiana

ProQuest LLC. #
789 East Eisenhower Parkway #
P.O. Box 1346 #
Ann Arbor, MI 48106 - 1346 #

To Suzanne, Alice, Robert, and Daniel.

ACKNOWLEDGMENTS

Several researchers have had a distinct influence on my work, through the exchanges I have had with them. I am thankful for these exchanges that have helped me develop the ideas presented in this manuscript. Prof. Trent Jaeger kindly discouraged me from attempting a thesis on separation of duty when I was searching in that direction; later, he provided me with useful references on operating system security. These references, his book on operating system security [1], and his encouragements, have helped me assess the place where my work fits in the field. With Pascal Meunier, I had fruitful exchanges on the limits of administrative models and practical security primitives in the context of the ReAssure project at CERIAS. These conversations with Pascal help me stay motivated, and his questions on the usability of the administrative model I was designing stirred me in the direction of providing templates to factor the administrative policies. Ed Finkler, Nick Hirshberg, Pascal Meunier, Steve Plite, and Dan Trinkle have helped me refine the practical examples, based on their experience in administering multi-tenants systems. Prof. Mikhail Atallah has helped me with the computational geometry aspect of this work. I am thankful to Prof. Matt Bishop for his encouragements and his coverage of access control models in his book on computer security [2]. It is during Prof. Ninghui Li's research seminar on access control that I got started working on SELinux and Type Enforcement. The title of this dissertation was found while brainstorming with Prof. Patrick Eugster. Last but not least, Prof. Vitek has always been supportive of this endeavour, from my initial application to Purdue, to the completion of this manuscript. I am dearly thankful to him for his support over the years as a person, colleague, mentor, and academic advisor.

As a teaching assistant for operating system classes, I relied heavily on the support from the technical staff of the computer science department. I owe many thanks

to the members of the technical staff for all the system troubleshooting on which they have helped me over the years, as lab assignments would stop working from one semester to the next after seemingly innocuous system updates. I have fond memories of troubleshooting the system updates with Steve Plite, Dan Trinkle, and Prof. Gustavo Rodriguez-Rivera, as well as fond memories of keeping the old Xinu Lab on life support with Mike Motuliak, Brian Board, and Ron Castongia.

Within the Computer Science department, I enjoyed my interactions with –and learned a lot from– S3 Lab colleagues, CERIAS colleagues, the System Lab hackers (I miss the Mad Pizza lunches), Dr Gorman and the late Amy Ingram, and all the students and professors that I worked with as a TA.

Many persons contributed over the years to make my stay in Lafayette an enjoyable experience. I am thankful to all of them, including Angelo and Christina from La Village Food Mart, Dorothée Bouquet, Edie Cassell, and Nick Hirshberg. The following student associations have also been an important part of my social life: Friends of Europe, the Purdue Beat Society, and the Purdue Fencing Club.

Finally, I want to acknowledge the following people that were important along the long path that leads to eventually graduating with a PhD: Guy Batmale, Patrick Bolton, Bill Cheswick, Julien Fourcade, Claude Gicquet, Jean-Alain Godet, Dominique Guérillot, Florent Gusdorf, Bruno Kerouanton, François Muller, Alan Robbins, Vanessa Ruat, Christophe Schuhmann, Elvire Serres, and Haruko Takeuchi.

And, of course, I am extremely grateful to my family for their support and encouragements all along, even when this path lead me to the other side of the Atlantic.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Thomas, Jacques D. Ph.D., Purdue University, December 2011. Accommodative Mandatory Access Control. Major Professors: Jan Vitek and Patrick Eugster.

In operating system access control, there is a traditional divide between discretionary access control (DAC), on one side, and mandatory access control (MAC), on the other side. Compositions of MAC and DAC have been modeled and implemented as operating system access control mechanisms. With composition, two access control decisions (one for DAC and one for MAC) have to concur for an access request to be allowed. DAC is typically supported by coarse grained mechanisms, and it vulnerable to Trojan horse attacks, two limitations that are addressed by MAC. MAC mechanisms are therefore of interest to security-conscious users and application developers that want to confine applications they use or develop. MAC mechanisms, however, can only be configured by administrative users and as such can not be used by regular users. This dissertation explores how MAC mechanisms can be made available to regular users of an operating system. Our approach consists in extending the Type Enforcement MAC model with an administrative model. We call this approach *accommodative mandatory access control*.

1. INTRODUCTION

Building software of a non-trivial size is difficult and costly. Building that software so that it is reliable and secure is even more difficult and costly [3]. As a consequence, most of the software that we use on a daily basis on personal computers has flaws. Some of these flaws can be used to force the programs to behave in ways that the program's authors did not intend. An example of such a flaw is the drive-by-download attack against a web browser. In drive-by-download, additional software gets installed on a user's system as a result of visiting a web site that contains malicious content that tricks the user's web browser into installing software. This installation is performed without the user's intent and constitutes a violation of the integrity of the system. The software being installed can be, for instance, a keylogger that will record and report to a remote system the text being typed on the keyboard of the computer where it is installed. This remote reporting constitutes a breach of the confidentiality of the key strokes between the user and the program with which the user wishes to interact. When the recording captures the user's credentials to online services, this breach of confidentiality can be used to impersonate this user on these online services, potentially leading to identity theft.

In brief, software we use on a daily basis has flaws and thus can not be trusted to behave according to the intent with which we use it. Consequently, there should be means for users of personal computers to compartmentalize the applications they use. That is, there should be mechanisms to restrict an application's access to the resources offered by the underlying operating system, including the resources that belong to the same user on behalf of whom the application is currently running.

To support a safe use of untrusted applications, where applications are confined to a safe behavior that also preserves their usefulness, the *operating system* of a

personal computer should provide access control mechanisms that are *fine-grained*, *comprehensive*, *backwards-compatible* with existing applications, and *configurable by regular users* of the system. We will first explain how we categorize users and then justify these requirements.

Personal computers are not always owned by individuals. For instance, the personal computers deployed in a company are owned by that company. In this case, the company often designates *administrative users* that are responsible for the configuration of the personal computers that *regular users* work on. Typically, administrative users can change the configuration of access control mechanisms on the personal computer in ways that regular users can not, because regular users are not granted the permission to do so. *Superusers* are a special case of administrative users who have unrestricted access to the operating system and its resources. Now that we have clarified the two categories (regular and administrative) according to which we consider users of a personal computer, we justify the requirements already mentioned.

Permissions should be fine-grained for several reasons. The immediate reason is that we want an access control mechanism that supports the principle of least priviledge, and hence a precise confinement of applications. Coarse permissions run against this goal and additionally run against our goal of having an access control mechanism that regular users can configure. Indeed, if permissions can only be considered at a coarse level, then the *administrative permission* to grant a permission is necessarily coarse itself. If administrative permissions can not be granted at a fine granularity to regular users, then they will probably not be granted to regular users at all. Consider for instance the `setuid` facility on UNIX. It allows for programs to run under another identity than that of its invoker. This is a powerful confinement mechanism. The permission to configure this facility, however, is coarse. Either one can configure it, or not. There is no notion of who can configure which programs to run under which identity. This administrative permission is too coarse to be granted to regular users of the system, hence depriving them from the ability to use this confinement mechanism.

The access control mechanism should also (a) be provided as an operating system facility, and (b) be comprehensive. What we mean is that this mechanism should be integrated with the the underlying operating system in a way that it constitutes a reference monitor [4]. This is necessary to provide guaranteed enforcement of access controls, as intuition and empirical evidence show [5]. We are considering multi-user time sharing systems because they represent the main trend of operating systems found on personal computers. As we explain in the related work (see Chapter 2), our work remains relevant on other kinds of operating systems (e.g. single-user multi-programmed). By comprehensive (b), we mean that the access-control mechanism should offer full mediation of the interactions between a running process and both the underlying operating system and another process running atop the same operating system. In this work we do not consider the enforcement of distributed access control policies, policies that span several instances of an operating system.

Finally, we want an access control mechanism that is backwards-compatible with existing applications. For this reason, it should be possible to deploy this mechanism as an overlay, without changing the application programming interface exposed by the operating system. Since this work starts on the premise that developing non-trivial applications is difficult and costly, we do not think it is reasonable *in the general case* to require that applications be re-engineered in order to benefit from security enhancements of the underlying platform. Consequently, capability-based access control mechanisms are not appropriate for the use cases we are considering, because they change the system's programming interface and therefore require a re-engineering of the application being confined. We recognize that this re-engineering can be minimal in the case of applications that are already split in multiple specialized components, each running with a minimal set of permissions in its access control domain, a technique known as privilege separation [6]. However, such applications represent only a small fraction of the ones used on personal computers. Furthermore, it is either the case that these applications had to be re-engineered to run with privilege separation (e.g. `sshd`) or were recently designed from the ground up to run

that way (e.g. Google Chromium). We now present a summary of the existing work on access control that has directly guided the evolution of the work presented in this thesis. A more detailed survey of the related work is in Chapter 2.

1.1 State of the Art

The access control matrix model [7] was introduced by Lampson during the infancy of research in access-control models and mechanisms. To this day, all access control models can still be modeled in terms of an access control matrix, as we show in the related work (see Chapter 2). Here, we would like to clarify our terminology concerning access control, specially since the terminology used in defining an access control matrix varies depending on the authors and their goals. There are two main access matrix models. On one hand, Lampson's model [7] provides a unified representation of *implemented* mechanisms. On the other hand, the model from Harrison, Ruzzo, and Ullman [8] (the HRU model) is a simplified version of Lampson's model that is amenable to proving complexity results on policy analysis problems. As we are approaching the problem of access control from an implementor's perspective, Lampson's model fits our modeling needs better.

In Lampson's access control matrix model, processes run within a *domain* and permissions are attached to domains. The access requests of a process are allowed or denied based on the permissions of the domain in which the process runs. We have reproduced Lampson's example in Fig. 1.1. In this example, domain1 owns and controls itself as well as domain2. This means that domain1 can change the permissions granted to domain1 and domain2. The access matrix model will be useful in tying together our presentation of the related work on access control. There are two things that we would like to point out and discuss further about this model: it is not implementable in a straightforward fashion and was designed at a time where the need for mandatory access control was nascent but little work had yet been performed on the topic. We discuss these points below.

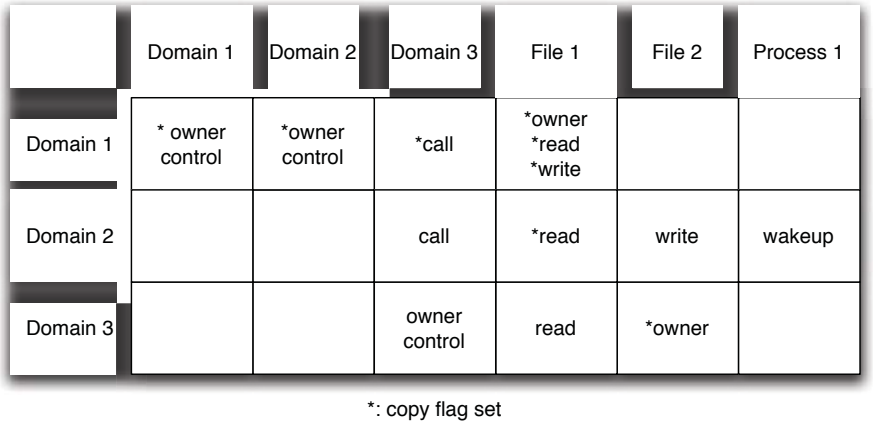


Figure 1.1.: Lampson's access control matrix model, reproduced from it original presentation [7].

While the access control matrix is a useful model, a straightforward implementation of this model is not practical: the matrix is large and dynamic. The matrix is large because it contains one element per pair of principal and resource. Also, the matrix is dynamic because the sets of principals and resources are not static: principals and resources can be added and removed from the system. By resource, we mean an entity that exists at the level of the operating system, because our focus is on access control provided by the operating system. A straighforward implementation would use a large amount of storage space to store the matrix. Additionally, this memory space would need to be compacted (respectively expanded) each time a resource or principal is removed from the system (respectively created in the system). However, the matrix is sparse and this property can be used to represent the matrix in compressed form. The matrix is sparse because accesses are restricted: a matrix that would not be sparse would represent a configuration where most principals have access to most resources; that is rarely the case. There are three general-purpose encodings of sparse matrices: by row, by column, and as a set of elements.

Two of these encodings have been used repeatedly by implementers: access control lists and capabilities. Access control lists (ACLs) consider the columns of the access control matrix. That is, for a given resource, which principals are granted which permission on it. Capabilities take a row-oriented approach and represent the permissions that a given principal has on resources of the system. The third encoding, as a set of elements, introduces a permissions lookup procedure that is more costly because the permissions are stored separately from the principal and the resource, in a centralized repository. Since permissions lookup need to be fast in order to preserve the performance of the system, caching has been introduced in system that use this matrix encoding. The family of systems derived from Flask [9], which our work extends, use this implementation strategy. Regardless of the encoding chosen to store the access matrix, there are two main approaches concerning the modifications that can be made to the matrix: discretionnary and mandatory access control.

In discretionnary access control (DAC), there are two important notions: resources are owned by principals, and principals are allowed to change the ACLs on resources they own. In other words, ACLs on a resource are left at the discretion of the resource owner. System resources are considered to be owned by system principals. Administrative users can control system principals in order to set the permissions on these resources. Lampson's access matrix, in its original exposition, describes DAC: a user can, through principals that he controls, grant access to any other users on any resources that he owns, by granting access to other user's principals. This approach presents problems: users have to be fully trusted to properly protect the resources that they own. This is not acceptable on a system that processes classified data, and specially state secrets, as the Ware report demonstrated [10]. Furthermore, DAC is susceptible to a form of attack called Trojan horse, where a program that looks useful and innocuous on the surface abuses the permissions granted to a user's principal. While the user, through a principal, is using the Trojan horse program, this program also uses the permissions granted to the principal to corrupt the system's integrity or leak data contained on the system. The drive-by-download attack we mentioned earlier can be considered a Trojan horse attack: the browser, while performing a useful function on the surface, displaying a web page, is actually corrupting the system's integrity by installing a rootkit. A variant of the drive-by download could post documents from the user's local hard drive on the Internet.

The confinement of programs with DAC is approximated in practice by having the confined programs run under a different identity. This identity has restricted access on the system, due to the ACL settings not granting much access to the principal running under that identity. Moreover, because the chosen identity owns very few resources, this limits the scope of ACL modifications that can be performed by a principal running under that identity, which limits the risk of privilege escalation. Privilege escalation is when a principal manages to broaden its allowed accesses on the system, beyond the set of permissions initially granted to it. To guarantee that programs run only under the confinement identity, they are set to change the identity

under which they run upon invocation. For instance, this mechanism is know as `setuid` on UNIX and `runas` on Microsoft Windows. These system facilities, however, lack a fine-grained permission model, and thus a fine-grained administrative model specifying which principal is allowed to configure which identity transitions.

In mandatory access control (MAC), additional controls are added on top of the discretionary ones. The goal of MAC is to prevent the permissions granted to a principal from being abused by a Trojan horse. The main idea of MAC is to make the determination of the domain in which a process runs based on criteria that are not entirely based on the identity of the user on behalf of whom the process runs.

The classic example of a MAC model is the Bell-LaPadula (BLP) model [11], which formalizes the handling procedures designed to preserve the confidentiality of classified documents [12, 13], with an explanation of the model in the context of an operating system. In the BLP model, security classifications are assigned to resources and security clearances are assigned to principals; classifications and clearances are drawn from the same set of labels. This set commonly comprises the following confidentiality labels, in increasing order of confidentiality: *public*, *confidential*, *secret*, and *top secret*. A principal is allowed access to a resource if and only if its security clearance is superior or equal to the classification of the resource. In addition to confidentiality labels, categories can be assigned to resources and principals in order to enforce a mandatory need-to-know policy. Examples of categories are *Navy*, *Army*, and *Nato*. With categories, the access to a resource by a principal requires that access be granted to that principal for each category that the resource is labeled with. The combination of confidentiality labels and categories results in a lattice-based classification of resources and principals; we discuss this lattice structure in more details in the related work (see Chapter 2). What is important to note here is that this lattice structure defines a partial order on principals and resources. The lattice and the partial order make BLP attractive; the lattice allows for simple graphical explanations of the access control model, while the partial order yields simple proofs of safety. By being simple, yet useful, the BLP model has been suc-

cessfully applied in computer systems. Transpositions of the BLP model to enforce integrity policies have been proposed, both by the authors of the BLP model and by Biba [14].

Some confinement problems, however, can not be described in terms of a partial order. The trusted labeler problem [15] is one such problem. In this case, the security goal is to guarantee that all documents printed from a trusted workstation will be properly labeled. In this context, a trusted workstation is a computer system trusted to properly enforce the BLP model, and proper labeling means that each page of a document coming out of the printer will bear the sensitivity and compartments of the documents in the header and footer of the page.

Type Enforcement (TE) is a MAC model that was created to address the trusted labeler problem, and more generally the class of problems known as high assurance pipelines. The key insight in the creation of TE was to recognize the non-hierarchical nature of the trusted labeler problem and, as a consequence, to create a non-hierarchical model to solve the problem. In TE, resources and principals have a type attached to them; a type is a label attached to an object. While this label is considered for access control decisions, it is not necessarily related to the internal structure of the object, contrary to the notion of type in programming languages. The type attached to a process determines the access control domain within which it runs; the types that are accessible from within a domain, and the operations that can be invoked on them, are declared in the Domain Definition Table (DDT). By configuring the DDT, it is possible to create arbitrary relations between domains. With TE the security of each domain can be analyzed and proven individually and the final proof of compliance can be assembled from these infividual proofs. This composability of proofs allows for successful divide and conquer approaches to proving security properties of software systems, including the trusted labeler. Moreover, if a software module is re-used across several software systems, its accompanying policy and proof module can also be re-used.

While the original presentation of TE does address the trusted labeler problem, it still shares a common limitation with BLP: all accesses are expressed in terms of read-like and write-like operations. We think this is a limitation because it is not possible to represent the diversity of interactions on a modern OS in terms of only read and write operations. For instance, the `bind` operation on a network socket is difficult to classify as a read or a write operation. This problem is solved by extending the original TE model with object classes, as shown in Flask [9]. Additionally, Flask demonstrates in practice the idea developed early on by the original authors of TE [16], namely that TE can be composed with BLP, some form of role based access control (RBAC, which we discuss in the related work), and identity-based access control.

Overall, the access control mechanisms provided by Flask satisfy all but one of our requirements: they are not configurable by regular users of the system. Focusing on type enforcement, this is the point that we want to address, so that regular users can protect themselves from flawed software. We outline our approach in the following section.

1.2 Accommodative Mandatory Access Control

We think that there is a middle ground between DAC and fine-grained MAC that has not been explored sufficiently. On one hand, DAC can be configured by regular users but is coarse grained and vulnerable to Trojan horse attacks. On the other hand, MAC is fine-grained and resilient to Trojan horse attacks, but it can not be configured *at all* by regular users. It would be nice if regular users had the ability to configure fine-grained MAC mechanisms in order to protect themselves from Trojan horse attacks like the drive-by download attack mentioned earlier. This ability, however, should not come at the cost of other aspects of the system's protection. For instance, regular users should still be prevented from tampering with parts of the security configuration that protect the system's integrity. In essence there should be an *administrative model* regulating how users can configure the fine-grained MAC

mechanisms. This is different from traditional MAC where the ability to modify the security configuration is granted exclusively to designated administrative users (the system's security administrators) and in an absolute manner (no restrictions on the modifications they can perform). With an administrative model, it is possible to split the administrative powers granted to administrative users; two direct benefits ensue. First, it is possible to provide a safety net to administrators. By granting them limited administrative permissions, one can limit the damage they could cause by mistake. Second, it is not necessary anymore to trust the administrators in absolute terms. Furthermore, it becomes possible to let regular users benefit from the presence of the MAC mechanisms and use them for the protection of their data, according to their own usage patterns. It is even possible to enable this scenario without additional configuration, by carefully defining the *default* scope of modifications that regular users can perform to the configuration of the MAC mechanisms. To summarize, a mandatory access control model extended with a fine-grained administrative model could support a wide-range of access control requirements, from DAC to MAC. We call this approach *accommodative mandatory access control*. Our thesis is the development of this approach.

1.3 Thesis Statement

It is possible to extend a MAC model and its operating system implementation so that regular users can reliably confine applications they use, in a way that is comprehensive, fine-grained, and backwards-compatible with existing applications, while preserving the mandatory nature of the access control configuration established by administrative users.

1.4 Approach and Contributions

Our approach to demonstrating our thesis is by construction, following the guiding principle that permissions have to be fine-grained at all levels. We have identified

three such levels which we describe later; first, we explain why permissions have to be fine-grained. There are two reasons why permissions should be fine grained. The first reason is that coarse-grained permissions prevent the application of the principle of least privilege (PoLP). The second reason is that coarse-grained permissions are are harder to grant as they require the grantee to be trusted for large sets of operations, which practically limits the delegation of permissions that can happen. Consequently, our approach consists in using or creating a fine-grained model at the three following levels. The first level is the base MAC access control model that we chose to extend. We have chosen to extend the TE model and its SELinux implementation, based on our survey of the related work. The second and third levels are respectively the administrative model for TE and the refinements to the coupling between a TE policy and the underlying system objects; we describe both in turn below.

Developing our approach on TE has involved defining an administrative model for TE. To support the fine-grained granting of administrative permissions to users, the administrative model has to be able to finely characterize the permission that are granted. Therefore, we have designed an administrative model that relies on pattern matching to characterize the contents of the policy rules being modified. As our work addresses the lack of an administrative model for a MAC model, we have avoided a simple transfer of problem from the lack of an administrative model to the lack of model regulating changes to the administrative policy itself. We have done so by designing our administrative model to support a recursive nesting of administrative permissions. In other words, an administrative policy can be regulated by another administrative policy, and so on. This is a novel technique for constructing administrative models, for which we have also designed and implemented a prototype on SELinux.

From a theoretical standpoint, an administrative model for TE could be sufficient to claim our thesis as demonstrated. From a practical standpoint, however, that is not the case. Consider the following example on SELinux, where all files under a home directory are labeled with the same type (`user_home_dir_t`), and with a user who

wants to protect differently a given file located in her home directory. With just a TE administrative model, the policy implementation choices to treat these files differently would be to either relabel that file, or make a copy of the file and relabel the copy. Both choices present problems: relabeling the original file effectively nullifies all the rules that were granting access according to the original type of the file, and creating a copy naturally introduce the risk that the contents of the two copies will diverge over time. We have conceived a scheme, which we call overlay labeling, to address this problem. The idea of overlay labeling is to let users add extra tags to objects, and let users define access control rules based on these tags.

The overlay labeling of filesystem objects is an operation that is performed offline, once per overlay. Consequently, it does not affect the runtime performance of the access control enforcement. We show how this operation can be performed such that the semantics of the existing security policy are preserved. The overlay labeling of network packets is more complicated for two reasons. First, as each packet entering or leaving the system has to be labeled at runtime according to the existing policy, the network packets are not pre-existing resources that can be labeled offline, and therefore the efficiency of their labeling has a direct impact on the runtime performance. Second, the standard implementation of network packet labeling relies on packet classification, which determines a unique label for each packet. Overlay labels, however, require the ability to return multiple labels per objects. This is a fundamentally different and new problem. We show how this problem can be solved by using a datastructure from computational geometry, interval trees, which we generalize to multiple dimensions. We prove that, for the general case, our solution is optimal within a constant factor.

An additional contribution of this thesis is the unified presentation of the access control models that we survey in the related work section, where we extend the concept of the Extended Access Matrix [16] to encompass subject and object transitions. This unified presentation allows a characterisation of the differences between access control models that is more precise than what was available previ-

ously. For instance, we show that TE is strictly more expressive than RBAC, as RBAC can be emulated by TE, but RBAC can not emulate TE because it lacks automatic role transitions. We also show that TE can not enforce low-water mark policies because it does not support domain transitions on read and write operations.

1.5 Manuscript Organization

Our manuscript is organized as follows. After a survey of the related work in Chapter 2, we proceed to demonstrate our thesis by construction, based on TE. In Chapter 3, we formalize TE and define a core administrative model for it. This core administrative model supports a fine-grained delegation of permissions on the TE policy, which is a necessary step to demonstrating our thesis. In practice, however, this model is not sufficient by itself. Indeed, the coupling of the policy to concrete system resources –the *labeling* of resources– is external to a TE policy and thus not captured by our core TE administrative model. Policy statements added by regular users are constrained to the granularity of the labeling of system resources by administrative users. This is why we introduce the notion of overlay labeling in Chapter 4. The idea is to let regular users declare and attach additional labels on system resources, so that they can refine the system policy to cater their needs. At the same time, user actions are still limited by the mandatory policy. In Chapter 5, we evaluate our work with case studies and compare it to existing solutions. We conclude in Chapter 6.

2. RELATED WORK

Our work involves, and therefore relates to, issues both at the modeling level and at the implementation level. At the modeling level, our work relates to the existing work on access control models and administrative models. At the implementation level, our work relates to access control mechanisms implemented in operating systems. Consequently, our presentation of the related work will cover both related models and related implementations. This chapter is structured as follows. Section 2.1 covers the early history of the field of access control, clarifies the kind of operating systems that we are considering, and presents connex work that is not as closely related as the work presented in the subsequent sections. Section 2.2 covers related access control models, and Section 2.3 covers the compositions and emulations of access control models.The ability for a model to emulate another one is used as a means to compare the relative expressive power of access control models. Section 2.4 covers administrative models, Section 2.5 covers implementations of access control models found in operating systems, and Section 2.6 concludes this chapter.

2.1 Introduction

Before delving into the survey of the work specifically related to ours, we feel that that an extended introduction to this chapter can benefit our reader. This extended introduction will be split in three parts. First, we will recap the early history of research in computer systems access control. Then we will present how the access control matrix model can be extended to represent and compose the access control models that are presented in this chapter. Finally we will briefly mention work which is related just enough that it needs to be mentioned and remotely enough that it will not be surveyed further in the remaining of this chapter. We want to present this

work, nonetheless, to help our reader better evaluate what our work relates to, and how closely. We now recap the early history of the field.

2.1.1 Early History of Computer Access Control

Early on (and now over 40 years ago), the Department of Defense created the Task Force on Computer Systems Security, chaired by Willis Ware. The findings of the task force were summarized in a report [10] which pointed, for the first time, many elements of what is now common knowledge in the field of computer security. For example, the findings included the assessment that commodity operating systems were not providing protection mechanisms that were adequate for the processing of classified governement data, with mixed classification levels, on the same computing facility. The report also mentioned the use of trap doors to penetrate a system and characterized the evolution in operating systems functionality that lead to the apparition of the need for access control within operating systems. We have reproduced this characterization in Figure 2.1, where we explain how our work is relevant to OSes with these different characteristics.

The Ware report was followed by the Anderson report [4], which introduced the notion of a *reference monitor* (see Fig. 2.2) as the base architectural pattern used to enforce access control. The Anderson report was also the first one to mention *trojan horse* attacks on a computer system. Similarly to the mythical trojan horse, a trojan horse in a computer system is piece of software that betrays its appearance. While looking like a useful piece of software, a trojan horse does not do only what it looks like it should be doing. For instance, on smartphones, modern trojan horses that pose (ironically) as security software intercept the SMS messages sent by banks for multifactor authentication [17].

Around the same time, Lampson formalized the representation of the configuration of existing access control mechanisms in terms of a matrix, the *access control matrix* [7] (see also Figure 1.1 in Chapter 1). This model is important, as it was de-

signed to represent in a unified manner the access control mechanisms implemented by computer systems at the time. It is this unified representation that enables the common representation and therefore the comparison of access control models. When augmented with the notion of transitions that happen upon the execution of an operation, the access matrix model can be used to represent and compose many access control models, as explained by Boebert et al. [16]. In our presentation of access control models, we will use an access matrix representation to expose the different access control models related to our work. We elaborate on this idea in Section 2.1.2. As noted by Lampson in [7], a straightforward implementation of the access matrix as a matrix would not be efficient in terms of space, as the matrix is sparse. It is therefore proposed that the matrix be implemented with a storage either by columns or by rows. The storage by columns (by resources) is called access control lists (ACL); it corresponds to storing, with each resource, the set of subjects that are allowed to access it, and the operations that they are allowed to perform on it. The storage by rows (by subjects) is called capabilities; it corresponds to storing, for each subject, the set of resources to which it has access, and the operations permitted on these resources. In other words, these two implementations correspond to encoding the matrix either by rows or by columns. The problem posed by both implementations is that, by scattering the storage of the access configuration, they make it more difficult to gather the access control configuration and audit it. For example, consider the following audit scenario: on a system that uses ACL storage, an auditor wants to determine all the resources that a subject has access to. In order to make this determination, the ACL attached to each and every resource of the system will have to be retrieved. Conversely, a capability implementation makes it costly to determine all the subjects that have access to a given resource. As a result, some implementations use a sparse matrix encoding technique to compress the access matrix and keep it as a whole[1]. Keeping the matrix as a whole supports both of these audit scenarios

[1]We have observed the use of sparse matrix encoding for the domain definition table and domain transition tables in the source code of SELinux. There, each non-empty cell of the matrix is stored in a hash table that is indexed by the subject, the object, and the class of object. The class of object

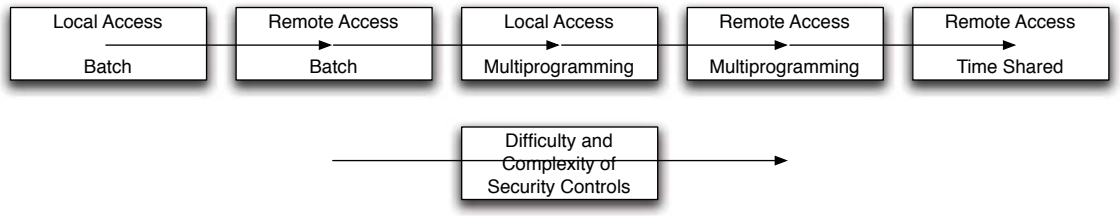


Figure 2.1.: Characterization of the difficulty of providing access control for OSes, based on their features, reproduced from the Ware report [10]. The need for access control appeared with the evolution of operating systems [21]; more specifically, it appeared with the introduction of non-volatile storage. Until that point, the computer system would be essentially stateless at the beginning of each computation. With non-volatile storage, there is the possibility that the next computation running on the system can access the data of a previous computation. As explained in the Ware report, the problem of access control can be solved with simple procedures for batch-processing systems by wiping the system clean between jobs, either by erasing the previous job's program and data or by dismounting the storage devices on which they reside. With multiprogramming, the executions of the computations are interleaved. This requires keeping several programs and their input and output data available to the system at the same time. Consequently, the previous simple workaround solution can not be applied. Instead, access control mechanisms have to be provided by the OS to isolate the computation, be it for integrity or for confidentiality reasons. Time sharing makes the access control harder to provide, by increasing the frequency at which the execution switches from one program to another, and by removing the control of the context switches from the programs (pre-emptive scheduling). Our work is in the context of timesharing systems.

efficiently, because the matrix contains all the information that is used to determine accesses. This information is a *protection state*.

An additional early paper that has had a lasting influence on the field was the paper by Salzer and Schroeder [18]. This paper was intended as a tutorial paper but ended up becoming a classic, mostly because of the 10 design principles that it contains [19]. The principle of separation of mechanims and policy, which was highlighted during the development of early extensible systems [20], is usually added to this list of principles.

---

is a refinement to access control rules that was introduced by Flask [9].

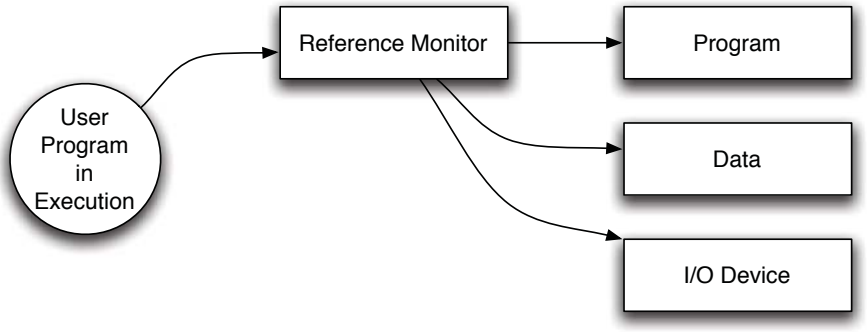


Figure 2.2.: A reference monitor, reproduced from the original presentation of the concept in the Anderson report [4]. A reference monitor is the architectural element responsible for the enforcement, in an access control model, of the "authorized access relationships between subjects and objects of a system. An implementation of the reference monitor concept is called a reference validation mechanism." An implementation of this concept requires that all interrelations of subjects and objects be mediated, and thus imposes the following three requirements on the design and implementation. First, the reference validation mechanism's integrity must be protected; otherwise, there are no guarantees as to what the validation actually validates. Second, the reference validation mechanism must be invoked for every access; this is also refered to as enforcing full mediation. Third, the reference validation mechanism must be an assured piece of software; in Anderson's words, it "*must be small enough to be subject to analysis and tests, the completeness of which can be assured*". Modern expositions and implementations of access control models [22,23] have refined the notion of reference monitor into three separate entities. The *policy decision point* decides of the fate of an access request, while the *policy enforcement point* is responsible for enforcing that decision. The policy decision point communicates with a *policy information point* to get the values it needs when evaluating the access request.

2.1.2 The Access Control Matrix

The access control matrix remains the model that is the most amenable to representing, in a uniform format, diverse access control models. We want to expand this idea that we have expressed earlier, both in the introduction of the thesis and in the introduction of this chapter; we do this in what follows.

First, we want to clarify why we chose to introduce the access control matrix by using the model originated by Lampson [7]. This model avoids the common confusion which consists in equating active entities (the entities that make access requests) with users. It does so by deliberately using a fairly neutral term for the active entities to which permissions are granted: *domain*, instead of *subject*. Indeed, the term "subject" is often equated to user identities as in [8] for instance.

As it turns out in the context of this thesis, which extends TE, the term domain unfortunately has a strong connotation as well[2]. Consequently, we will use the term *subject* to describe the active entities of the access control models that we present. This choice will make our exposition of the access control models conform with the usual terminology. We will, however, treat subjects as composite entities, similarly to the Extended Access Matrix (EAM) model [16]. By composite, we mean that subjects can possess a *set* of attributes, besides a user identity, that are used when evaluating access control decisions. In the BLP model [11], examples of these attributes are the clearance of the user identity, for the discretionnary access control, as well as the clearance and set of categories allowed to that user, for the mandatory access control. In our context, where we consider access control from the perspective of the operating system, a subject is a process. Consequently, these subject attributes will be attached to processes. In the same way that subjects are composite in order to support the composition of access control models, *objects* are composite.

In general, complex problems are easier to analyze and solve if they can be decomposed into simpler ones. This holds true for the composition of access controls

[2]The term "domain" has a well defined meaning in the context of type enforcement, as we show later in this section.

where several access control mechanisms, each implementing a separate access control model, may be used in conjunction to regulate accesses on a system. The EAM [16] is the embodiement of this decomposition of the exposition (and implementation) of access control models as separate access control matrices. The idea with the EAM is to describe the composition of access control models as the composition of a set of access control matrices, each representing one of the access control models being composed. The evaluation of an access control request is then performed individually on each of the access control matrices, passing the subject attributes that are relevant to each matrix's model. For example, the mandatory access control of MLS is only concerned with a subject's clearance and allowed categories, whereas the discretionary control of MLS is only concerned with the user's identity. An access request is granted if and only if each of the evaluation allows the access request. This is the reason why we insisted that subjects are composite, because they store attributes that are relevant to multiple access control models, and that we did not want to retain the name "domain", because this is precisely an attribute name that is used by domain and type enforcement (DTE) [24], a variant of TE (see Section 2.2.2 for a description of TE and its variants).

The EAM model covers only the part of the access control decision that is concerned with whether to allow an access request or not. Another part of access control models that is critical to the practical confinement of software is the change of type attribute settings that can happen when an access is allowed. An example of this change is the change of user identity associated that can be triggered by the `setuid` facility when a process replaces its binary program image by one constructed from a file that has `setuid` configured. In TE, these transitions are represented in terms of a matrix called the Domain Transiton Table (DTT). This matrix declares the domain in which a program should execute based on the domain of the calling program, either the same (not transition), or another one (domain transition). To keep with our goal of generality, the EAM needs to be extended with a set of *subject attribute transition tables*, one for each model that supports subject attribute transition. The subject at-

tribute transitions can happen on different operations: reading a file of low integrity can downgrade the integrity of a process (in the Biba low-water mark model [14]); executing a `setuid` program can change the identity of the running process.

Similarly to our insistance on having subjects be multi-dimensional, objects also need to be multi-dimensional since different access control models will consider different attributes of the objects. For instance, an access control model like MLS [11] considers two attributes on a file: the confidentiality label and the set of need-to-know mandatory categories. In keeping with the symmetry of the model, transitions can also happen on object attributes. For instance, a write to a file of high integrity by a process of low integrity will downgrade the integrity of the file (in the Biba low-water mark model); creating a file in a `setuid` directory will set the identity of that file to be the identity of the owner of the directory, instead of the user on behalf of whom the process is running. These transitions can also be represented in terms of a matrix, which would be called the *object attribute transition table*.

We provide one full representation of a subject attribute transition table, the domain transition table for the trusted labeler example by Boebert and Kain [15], in Figure 2.5c. Otherwise, we have condensed the representation of these tables by using arrows that overlap the access control matrix. The tail of the arrow is on the operation causing the transition, in the matrix cell corresponding to the subject and object of the access request. The head of the arrow is in the cell corresponding to the new subject attribute (for a subject transition) or the new object attribute (for an object transition). Consequently, subject transitions appear as vertical arrows, and object transitions appear as horizontal arrows. An example covering both subject and object transitions is the low-water mark policy by Biba, illustrated in Figure 2.4c.

2.1.3 Connex Work, Not Directly Related

Our work is related to access control models, which reason on the external actions of an application, and not on information-flow models, which reason on the

flow of information inside an application. To reason on the internal behavior of an application, information-flow models and their implementation require access to the source code of the application being analyzed. While getting access to the source code may not always be practical, the main limitation lies actually in the lack of models and techniques that scale to the size and complexity of common applications like a web browser. Some hybrid models that limit the scope of source-code analysis to the interfaces of a program are more tractable [25]. Examples of information-flow and language based techniques that we will not cover further include lattice information flow models and proof-carrying code. Lattice information flow models have been designed [26–28] to express what it means for a program to enforce a security policy. These works have focused on lattice information flow models, with the goal of preserving confidentiality. We are aware of two compilers that help guarantee the enforcement of information flow policies, by analyzing and instrumenting programs at compile time [29–33]. Besides a secure email client [34], few applications have been built with these languages as they are hard to work with. There is work in integrating these approaches with OS-based security, so that the policy enforced by an application and the policy enforced by the operating system are tightly coupled. For instance, [35] works on integrating application-enforced confidentiality policies with their OS-enforced counterpart. In earlier work [25], some of the same authors worked on integrating integrity policies to enforce a simplified version of the Clark-Wilson model of integrity [36], which they named CW-Lite. Proof-carrying code [37, 38] was developed to solve the performance problem associated with the sandboxing of mobile code, without giving up on the safety guarantees provided by sandboxes. Typical properties proved in PCC are memory safety and the termination of simple programs. The main limitation of PCC is the size of the proofs, which causes storage, transmission, and generation issues for any sizable program. Additionally, we do not cover inference-control models [39, 40] for two reasons. First, our work is on operating system security, where data is considered unstructured. Inference-control applies to structured data, and namely to database management systems. Second,

inference-control targets confidentiality policies, whereas the main focus of our work is integrity.

We have introduced the base concepts of access control, with a recap of the early history of research in computer systems access control. We have also briefly introduced research trends that are related to our work, but more remotely than what we will now present. In the rest of this chapter, we present the work on access control that is directly related to ours.

2.2 Access Control Models

An access control model defines how access requests are evaluated against the protection state. That is, when an access is attempted, how is the access request represented, and does the protection state of the system allow this request or not ? This decision is made by comparing chararacteristics of the access request against the protection state of the system on which the access is being requested. At the level considered by access control models, an access request is represented as a 3-tuple $subject, operation, resource$. The *subject* is the active entity that is attempting to perform an *operation* on a *resource*. Subjects, operations, and resources are characterized differently depending on the access control model. In DAC for instance, subjects are fully characterized by the identity of the user on behalf of whom they execute[3]. A common example of resource identification is the path of a file, with the associated read, write, and execute operations. An access request is allowed if an only if[4] the protection state allows it. In full generality, the protection state of a system can be viewed as defining a set of allowed requests. The evaluation of an access control request then becomes a membership test: does the access control request belong to the set of allowed requests. Many techniques of set theory have been used in access

[3]Because subjects are fully characterized by their identity in DAC, DAC is also known as identity based access control (IBAC)

[4]If a request could be allowed without being allowed by the protection state, then there would be a violation of the definition of a reference monitor.

control models, to facilitate the definition of the set of allowed requests. For instance, regular expressions and constraints have been used to define access control rules in comprehension. Regular expressions support the definition by comprehension of a set of resources, by defining a language that contains the set of resources to which access should be allowed (or denied). Constraints can be used to restrict the set of accesses that are allowed, to simplify the reasoning on the safety of the system [8, 41]. The implementor, then, has to strike a balance between the expressive power built into the access control model and the implementability of that model. For instance, efficiently testing a stream of incoming access requests against a set of regular expressions can be costly [42]. Similarly, it may not be practical to store an infinite record of the accesses performed by a user, when enforcing history-based access control constraints like the Chinese Wall model [43].

A special kind of access control model is the one used to describe the accesses made against the protection state of the system. These models, which regulate who can modify which parts of the protection state, are called *administrative models*. An access control model that captures both regular access requests and administrative access requests can be viewed as reflexive.

With the distinction between access control models and administrative models established, we will now present access control models in the remainder of this section.

2.2.1 Discretionary Access Control

In DAC, subjects are fully characterized by the identity of the user on behalf of whom they execute. This means that a process running on behalf of a user will possess all the permissions granted to that user, as there is no other attributes attached to that subject. As a result, systems that rely exclusively on DAC are vulnerable to trojan horse attacks.

2.2.2 Mandatory Access Control

Until the early 1970s, it was not generally realized that two fundamentally different types of access control exist [44], namely DAC and MAC. Now that we have presented DAC, we present MAC models. We first present MAC models as they were originally exposed. Then, we present them in terms of an access matrix, with the transition extensions that we introduced in Section 2.1.2.

Multi-Level Security – Bell La Padula

The Anderson report [4], which introduced the concept of a reference monitor, is one of the earliest reports on studies to address the security needs of institutions that handle classified national security information, namely multilevel security. The findings of this report were that current systems and system development methods were inadequate for supporting multilevel security with high assurance. Two models of multi-level security have been subsequently proposed: the Bell-LaPadula policy model (BLP) [11], for confidentiality, and the Biba variants, where the multi-level rules are transposed to support integrity policies [14].

The BLP model [11] is a formalization of the security policy used with physical documents in institutions that handle classified national security information, as prescribed in Executive Orders 10501 [12] (when BLP was formulated) and 13526 [13] (the current one at the time of this writing).

Each document is assigned a security classification, which is composed of a security level and a set of categories. Security levels are a set of strictly ordered symbols that represent the sensitivity of a document. In other words, security levels represent the damage that could ensue if the document was leaked. A typical set of sensitivity levels is, in order: { confidential, secret, top secret}. Categories represent the topics, projects, or organizations related to the document. Typical examples of categories are {NORAD, NATO, Army }. Users of the system are assigned clearances based on their trustworthiness. A clearance, like a security classification, is composed of a sensitivity

Clearance \ Classification	Confidential	Secret	Top Secret
Confidential	observe/modify	modify	modify
Secret	observe	observe/modify	modify
Top Secret	observe	observe	observe/modify

Figure 2.3.: The hierarchical part of the Bell La Padula model, with the security levels $Confidential \leq Secret \leq TopSecret$, encoded as an access control matrix

level and a set of categories. A clearance is said to *dominate* a security classification if the level of the clearance is superior or equal to the level of the classification, and if the set of categories of the classification is included in the set of categories of the classification. Based on this dominance relationship, the Bell La Padula model enforces the security goal of confidentiality with the following two rules:

- Simple security: a subject is allowed to observe only documents that are dominated by her clearance (no read-up).
- Star property: a subject is allowed to alter only objects whose security label dominate her clearance (no write-down).

More precisely, the categories are not needed to enforce confidentiality. Categories are used to enforce need-to-know security goals, where users should have access to documents only if that access is necessary for them to perform their intended duties within the organization. The hierarchical aspect of the Bell La Padula model can easily (and concisely) be encoded as an access control matrix, as shown in Figure 2.3.

Biba

Following the work of Bell and La Padula, Biba [14] proposed several mandatory access control models to protect the integrity of data. In a fashion similar to Bell-

LaPadula, these models rely on hierarchical security levels that are assigned to objects (resp. users) of the system as classifications (resp. clearances). The main difference is that these security levels represent *integrity* levels. The Biba report offers several hierarchical models of integrity [14] that we present in turn in this section. The *strict integrity policy* is the direct transposition of the Bell La Padula model, from confidentiality to integrity protection, with static labels. According to Biba, the strict integrity policy can be considered the "complement" or "dual" of the [BlP model].[5] The following rules are the result of the transposition:

- A subject can only observe objects whose integrity level dominate her own integrity level.
- A subject can only modify objects whose integrity level are dominated by her integrity level.

These rules prevent high integrity subjects from being corrupted by the observation of low integrity data and they prevent high integrity objects from being tampered with by low integrity subjects. We have represented this model in Figure 2.4b. Other models are also proposed in the same report (Figure 2.4c presents these models together):

- low-water mark on objects: an object's integrity level is the lowest of the integrity levels of all subjects that modified it. This policy has the problem that it does not prevent the modification of high integrity objects. It just records that such a modification happened by keeping a "low-water mark" on the object.
- low-water mark on subject: if a subject observes an object whose integrity level is lower, then the subject's integrity level is automatically lowered, to that of the object it observes.

[5] Actually, integrity at large has been considered to be the dual of confidentiality in [45]. This may be the origin of the popular belief that confidentiality and integrity can not be jointly achieved, as a result of being dual properties.

S: the set of subjets s

O: the set of objects o. (Note: in Biba's presentation, "*the intersection of S and O is the null set.*" As a consequence, direct interactions among subjects are not represented in this model, contrary to the original Bell-LaPadula model.)

I: the set of integrity levels. The report suggests that these levels can be compartmentalized (again, in a fashion similar to the Bell La Padula model) to separate different applications of the system.

il: $S \cup O \rightarrow I$, a function that returns the integrity level of each object and subject of the system. This function and the dominance relation $\leq$ on integrity levels define a lattice.

$\leq$: a subset of $I \times I$, the dominance relation on integrity levels.

min: $\wp(I) \rightarrow I$, a function that returns the greatest lower bound (meet) of the subset of I specified.

(a) Base definitions

Clearance \ Classification	Confidential	Secret	Top Secret
Confidential	observe/modify	observe	observe
Secret	modify	observe/modify	observe
Top Secret	modify	modify	observe/modify

(b) The strict integrity model, represented with three integrity levels as an access control matrix. From this figure, it is visible that the strict integrity policy is the dual of the Bell La Padula model for confidentiality (see Figure 2.3).

Clearance \ Classification	Confidential	Secret	Top Secret
Confidential	observe/modify	modify / observe	modify / observe
Secret	modify / observe	observe/modify	modify / observe
Top Secret	modify / observe	modify / observe	observe/modify

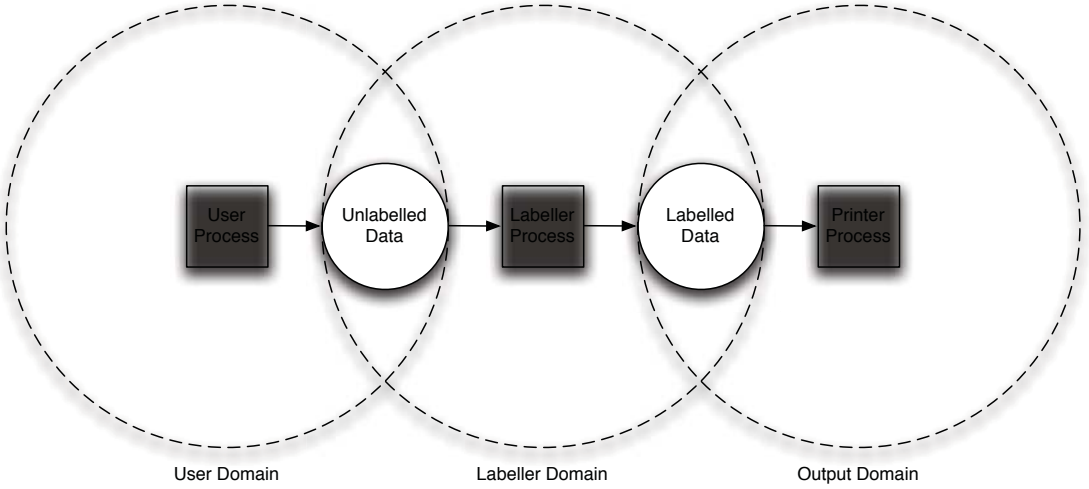
(c) The low-water mark model with floating labels for both subjects and objects, represented as an access control matrix. The downgrading (label transitions) for subjects take place upon observation of lower integrity objects (vertical transitions). The downgrading for objects takes place when they are written to by a subject of lower integrity (horizontal transitions). From this figure, it is visible that a system implementing floating labels runs the risk of downgrading the integrity of the whole system over time: all the transitions lead to lower integrity subjects and objects.

Figure 2.4.: The family of integrity models proposed by Biba in [14].

Type Enforcement

Type enforcement was created to support a use case that multi-level security can not address (and that was of interest) in the Secure Ada Target (SAT) [15]. Namely, the MLS integrity policies can not support the trusted labeler. The trusted labeler is an example of a non-hierarchical policy. In other words, the trusted labeler is an example of an application that can not be supported by a hierarchical policy. The trusted labeler is a mechanism which must guarantee that printed copies of classified documents bear their classification in the footers and headers of each page. This is to guarantee that information can not be leaked simply by printing it and walking away with it. With the headers and footers bearing the document's classification on each page, security personnel can prevent the physical copies of classified documents from being physically exfiltrated. The SAT implementation effort was aiming for the A1 certification level of the U.S. Department Defense Trusted Computer System Evaluation Criteria (TCSEC) [46]. As a consequence, enforcement of the security policy had to be formally proven on the system implementation. Following the same reasoning as [47], SAT was built in a modular fashion, with modular and composable proofs of compliance. Type enforcement is what enabled the composition of proofs of compliance.

We now present Type Enforcement, based on the trusted labeler example. The trusted labeler is an example of a high assurance pipeline that can be graphically represented as depicted in Figure 2.5a. A high assurance pipeline is a sequence of processing steps for which one can prove that, if data is output in the last step, the data has been processed by each of the previous processing steps, in order. Other examples of high assurance pipelines are commonly found in network guards, where data must be encrypted before being sent over non-trusted networks, and network access control must be enforced in a guaranteed fashion. The encryption module and the access control module are components of assured pipelines in network guards, in the same way that the trusted labeler was part of an assured pipeline in the SAT platform.



(a) "*Unverified and potentially hostile programs are encapsulated in the User domain. The labeler module ../.. is encapsulated in the Labeler domain and is verified to properly translate internal labels to readable form, and place them in the correct positions in the data. The output module ../.. is encapsulated in the Output domain and is verified to not tamper with labels. None of the domains in the example invoke any form of privilege.*"

Domain \ Object Type	Unlabelled	Labelled
User	observe/modify	null
Labeller	observe	observe/modify
Output	null	observe

(b) The domain definition table for the trusted labeler

Domain \ Called domain	User	Labeller	Output
User	execute in same domain	transition to Labeller	
Labeller		execute in same domain	transition to Output
Output			execute in same domain

(c) The domain transition table for the trusted labeler

Figure 2.5.: The trusted labeler example, adapted from [15].

The TE model can be viewed as a hybrid of Lampson's matrix model and the floating label policies proposed by Biba. From Lampson's model, TE has retained the notion of granting permissions to domains, with domains being entities not directly connected to users. From Biba's model, TE has retained the notion of dynamically changing the security context of a subject, based on the actions it takes. Whereas Biba's low-water mark model would lower a subject's integrity level upon observation of low integrity data, TE can transition a process into another domain when it starts executing another program; this is called an *automatic type transition*. These automatic type transitions are similar to the `setuid` facility [48]. However, TE domains are a notion orthogonal to user identities. This allows for running programs under the super-user identity, but in a restrictive domain that actually curtails the broad privileges normally associated with the super-user identity. The notion of a TE domain is also unrelated to the notion of an MLS clearance. This allows for confining the trusted labeler in a way that:

- guarantees it is not bypassable,
- protects it from corruption by other subjects,
- confines it so that there is no need to trust the labeler for more than proper labeling.

These last two elements support a modular decomposition of the specification and proof that the labeler and its integration in the system properly enforce the mandatory labeling of printed documents. An additional benefit of type enforcement is that it can reduce the attack surface of programs, hence limiting the scope of the audit when assuring the code of security-relevant programs [25].

The original exposition of TE [15] contains a motivating example for TE, together with an extensive explanation of how significantly TE helps the assurance effort on an information system. The subsequent adaptation of TE to UNIX [24], Domain and Type Enforcement (DTE), is more intelligible these days. DTE was designed to substantially improve the security of UNIX systems while maintaining a high degree of

backward compatibility and avoiding increases in administrative overhead. The security improvement is achieved by overlaying DTE domains on the UNIX domains; the backward compatibility and low administrative overheads are achieved by automating the common scenarios. We explain these points in turn.

In the standard UNIX access control model, a process's rights on an object are determined based on the {user, group, others} permission bits of the object, which user and group the object belongs to, and the user and group attributes of the process performing the request. In TE, that process's rights depend on the type of the object it is trying to access and the domain in which the process is running. DTE overlays this access control on top of the UNIX access control. This supports a partitioning of the whole set of rights that are normally available to a process based on its system user identity. In a later article, Walker et al. [49] have shown that this partitioning of rights can, for instance, be used to confine system daemons that would otherwise run with full super-user privileges. This confinement improved the security of the system. With type enforcement, every system object has a type, and every process runs inside a domain. DTE limits the administrative overhead by providing mechanisms that can be used to automate the system's behavior. For instance, the labeling of filesystem objects is simplified by relying on the filesystem hierarchy to automatically type files and directories based on the type of their parent directory. For processes, they automatically inherit the type of their parent process, unless they execute the *entry point* of a domain [6], and that entry point is set to trigger an *automatic domain transition*. In that case the new process will –upon invocation of the `exec()` system call– be running in the domain whose entry point it just executed.

Using the same representation of transitions that we have used to represent the Biba models in terms of an access control matrix, we can represent TE as an access control matrix, as illustrated in Figure 2.6.

[6]This notion of entry point is similar to the one defined by Lampson in [50].

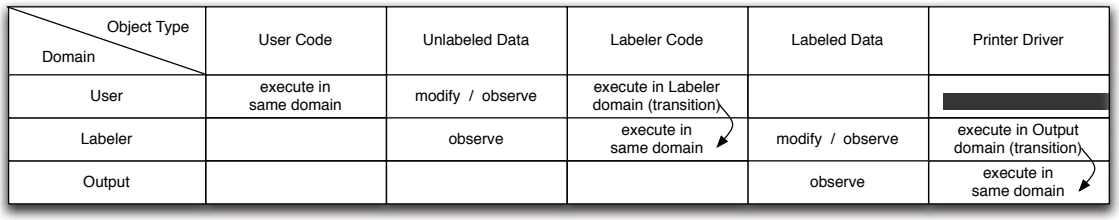


Figure 2.6.: The trusted labeler example, adapted from [15], represented in terms of an access control matrix

2.2.3 Role-Based Access Control

The main goal of Role-Based Access Control [51] (RBAC), is to simplify the administration of an automated security policy. This simplification is achieved by granting permissions to roles (instead of users), and assigning users to the roles they need in order to perform their duties, as illustrated in Figure 2.7. What makes RBAC attractive for security administration is the fact that duties performed by individuals within an organization are usually performed due to a responsibility assigned to the individual. Since these responsibilities are usually defined as functional roles, RBAC structures the security policy in a way that offers a natural mapping from a business organization chart to security administration. To further ease security administration, many extensions have been proposed to RBAC:

- Role hierarchies: In an attempt to model the hierarchy of functional roles that appear in a company's organization chart, RBAC has been extended with role hierarchies (see Figure 2.8). As pointed out in [51], there are three different semantics associated with role hierarchies.

- Temporal constraints: In an attempt to model worker shifts, temporal extensions to RBAC have been proposed: Temporal RBAC (TRBAC) [52], Generalized TRBAC (GTRBAC) [53], and their XML encoding, X-GTRBAC [54]. The interaction of temporal constraints and role hierarchies has been studied in [53].

Figure 2.7 is a reproduction of the Core RBAC model from the RBAC standard [55]. An important notion in this model is the notion of a session. Within a session, a user can activate roles, which in turn activate permissions, that can then be used to perform tasks on the system. Some have opposed to the inclusion of the notion of session as a part of the RBAC standard [57]; the origin of this notion can be traced to the database management system origins of RBAC [58]. The session is used to impose dynamic separation of duties constraints on subjects, to protect the integrity of the data being manipulated [59].

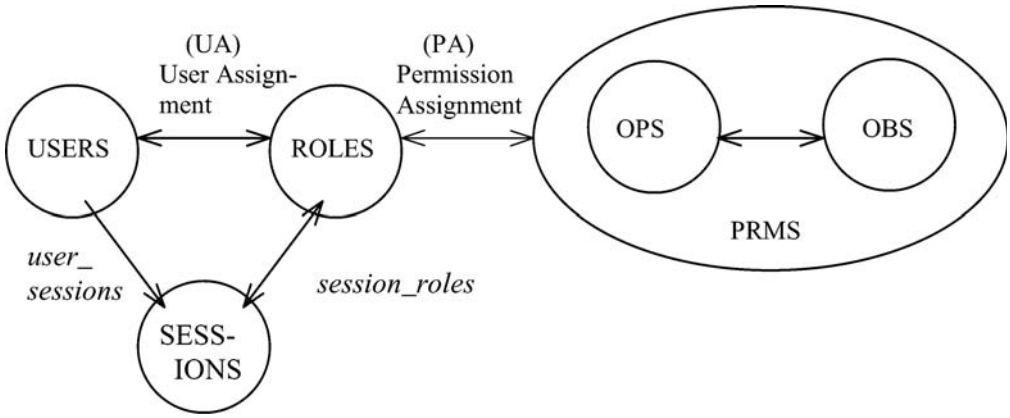


Figure 2.7.: The Core RBAC model, reproduced from the standard [55].

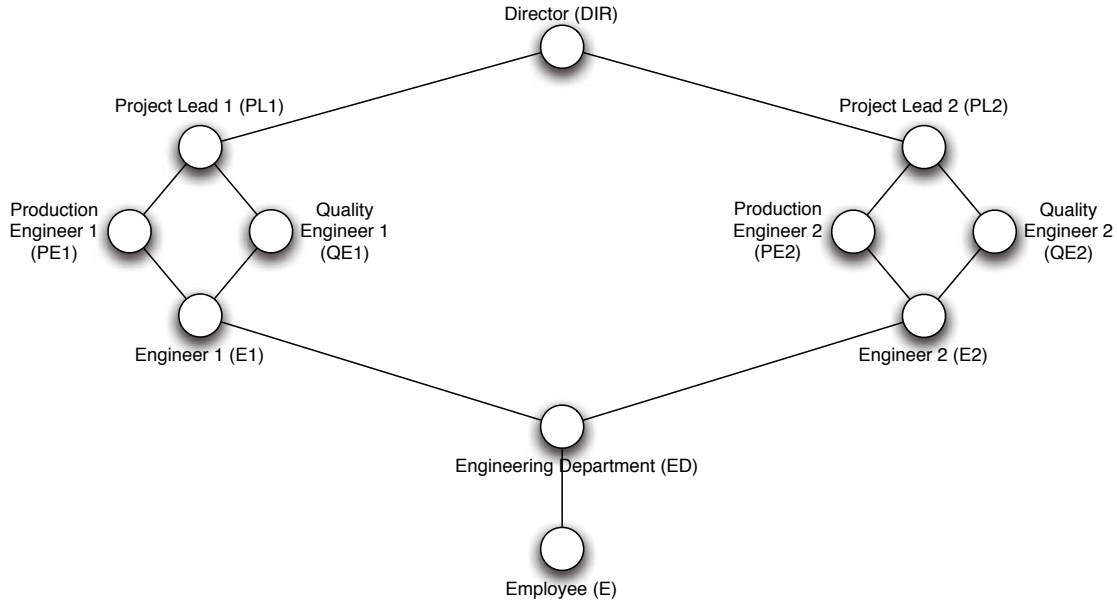


Figure 2.8.: The typical example used to illustrate role hierarchies, reproduced from [56]

We consider RBAC to be a mechanism for simplifying administration, and not an access control model per se. For the same reason, we consider that the question of whether RBAC can support discretionary or mandatory access control is not a valid question. The representation of RBAC as an access control matrix is straightforward: users are replaced by roles.

2.3 Emulation and Composition of Access Control Models

In this section, we complete our exposition of access control models by showing how a given access control model can be used to emulate another one. Then we present how native representations of access control models can be composed, without resorting to emulation.

2.3.1 Emulation

So far, for each access control model that we have presented, we have first reproduced its original exposition and then provided its representation in terms of an access control matrix. In other words, we have shown how that access control models could be *emulated* by an access control matrix. The models that use transitions, either on objects or on subjects, need to be extended with transition tables. Other emulations have been presented before. For instance, Pitelli [60] presents an emulation of the Bell-LaPadula using the HRU model, and Kuhn [61] presents an encoding of an RBAC policy in terms of an MLS policy. The motivation for emulating RBAC with MLS is that it allows re-using assured MLS systems to support RBAC policies without having to assure an RBAC implementation; the reverse mapping has also been performed Zhao and Chadwick [62], to support MLS policies on RBAC systems. We show in Section 3.2 that TE can emulate RBAC but not vice versa since RBAC does not possess a notion of one-way subject transition. Figure 2.9 presents a graphical summary of the model encodings referenced or introduced in this manuscript.

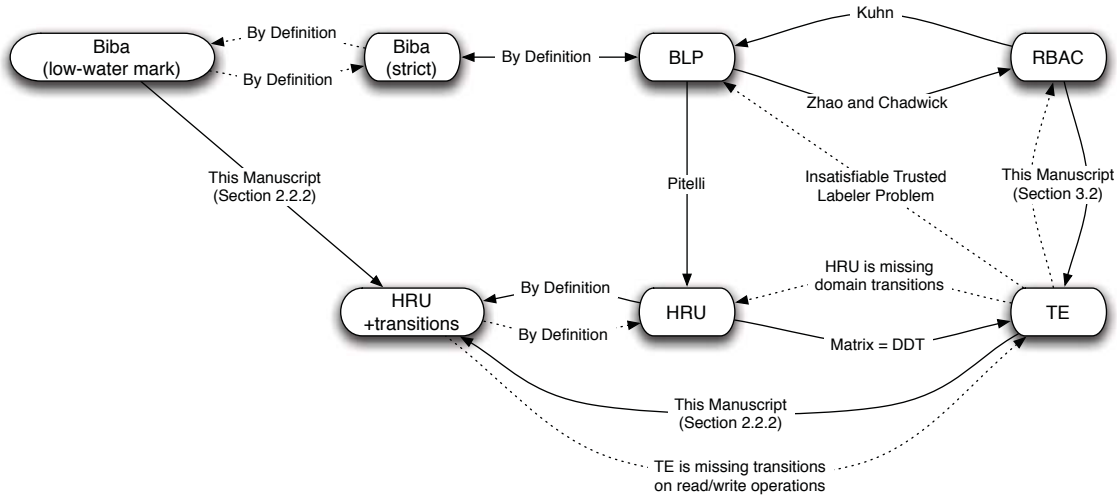


Figure 2.9.: Summary of the ability of the access control models presented in this related work to emulate one-another. Edges with a plain line represent a possible emulation, from the emulated model, to the emulating model. Edges with a dotted line represent impossible emulations. These results, when they are not a direct consequence of the definition of a model based on another one, are either provided in this manuscript or come from the following articles: Kuhn [61], Pitelli [60], Zhao and Chadwick [62], and the trusted labeler problem by Boebert and Kain [15].

A more general form of emulation consists in building a minimal access control model, on top of which other access control models can be built. This approach has been proposed in the Generalized Framework for Access Control (GFAC) [22,63] (and demonstrated in RSBAC [64]). Jajodia et al. [65] independently proposed a similar model. More recently, the Policy Machine effort at NIST is following that idea as well [66].

In the next section, we present cases where native model implementations are composed.

2.3.2 Model Composition

The emulations we described in the previous section are interesting from a theoretical standpoint. However, these emulations may not be desirable in practice: the emulation of a policy by another policy engine may be slower and, more importantly, the encoding required for the emulation is likely a verbose enumeration of the state space of the emulated policy[7]. By being verbose, this emulation is likely hard to analyze. To keep policies as tractable as possible, it is therefore desirable to compose policies *natively*, instead of picking a base policy and emulating the other ones based on it. Besides the previous section, we have so far introduced every security model in isolation from the other ones, to clarify and focus their exposition. Except for the access matrix model, which was historically the first model, all these models were actually presented composed with another access control model, starting from their original exposition. In this section, we present examples of access control models compositions that were described in the literature as native compositions.

The Bell-LaPadula model exposition [11] contains a description of the composition of the mandatory access control policy with a discretionary access control matrix model. The Biba report [14] also contains an explanation of integrating the mandatory access control model with a discretionary access control model, namely access control lists.

[7]Emulating RBAC on MLS can be justified by the cost of system assurance, which is then further amortized by implementing RBAC on a system *already* assured.

The original exposition of Type Enforcement [15] discusses how it is supposed to complement a Multi-Level Secure policy on the SAT platform: "*To enforce the mandatory access policy, the TOP compares security levels of the subject and of the object, and computes an initial set of access rights according to the algorithm defined in Section 4.1.1.4 of the TCSEC.*" This section of the U.S. Department of Defense's Trusted Computer System Evaluation Criteria [46] (the "Orange Book") contains a description of the Bell-LaPadula security model.

LOCK, the successor of SAT does however not have co-existing UNIX and TE policy enforcement. More precisely, since LOCK emulates UNIX over its type-enforced trusted computing base, TE can only be used to confine a whole UNIX emulation, and not just individual UNIX processes, according to [67]. The same authors present in [24] how, in their prototype, DTE complements UNIX security, at the process level.

The most complete example of simultaneous native composition of policies on an existing system that we are aware of is the one performed in Flask [9] (now in SELinux), which provides simultaneous support for UNIX discretionary access control, a form of RBAC, MLS, and TE. These models are composed natively by combining the information needed for their enforcement in the same security context. Conceptually, the security context contains fields for each of the security model implemented (practically, it is a colon-separated string), to form a 4-tuple like this (user, role, domain, mlslabel). The notion of role supported by Flask has been slightly adapted: users are still assigned to roles, but instead of assigning permissions to roles, one assigns domains to roles. For an access control request to be allowed, each access control model has to allow it, based on its configuration. This can be viewed as performing a lookup in 4 access control matrices at the same time, one for each model. RSBAC [64] and TrustedBSD [68] perform a similar native composition of access control models.

2.4 Administrative Models

In this section we revisit the access control models that we presented in section 2.2 and show, for each of them, the administrative models that are available,

2.4.1 The Access Control Matrix

The access control matrix [7, 8, 69] models subjects performing operations on objects and subjects, with the subjects considered as objects when an operation is performed on them. As such, the access control matrix model can represent administrative operations, and the permissions they depend on. A typical property of discretionary access control, for instance, is that the owner of a resource has the administrative permission to grant or revoke access on this resource to other subjects. The ownership of a resource is indicated by an extra flag, `owner` in the matrix cell that represents the permissions of the owner on the resource. While the information used to encode this administrative policy fits in the access control matrix, the rule that decides on the interpretation of this information is not modifiable. In other words, while the access control matrix can be used to represent the settings of arbitrary administrative models, the administrative models themselves fall outside of the scope captured by the matrix. As such, they can not be changed.

2.4.2 Mandatory Access Control

Mandatory access control models were designed under the assumption that a security officer would be tasked with determining and configuring the appropriate security policy for a system. For instance, "[Bell LaPadula] has no policies for the modification of access rights. As a matter of fact, [Bell LaPadula] was originally intended for systems where there is no change of security levels" [70]. Similarly, the models presented by Biba [14] have no administrative model either. Original expositions of TE [15] and its integration in UNIX systems [67, 71, 72] did not either present an

administrative model. The Policy Management Server (PMS) [73] for the SELinux implementation of TE is the only previous attempt at defining an administrative model for type enforcement. Its model, however, suffers from a major limitation. In PMS, the administrative permissions can only be specified in terms of, *either* the object types to which access can be granted, *or* the subject types to which access can be granted. In other words, there is no way to state administrative permissions in terms of both the subjects and the objects for which they allow permissions to be administered.

2.4.3 Role-based Access Control

Many models have been proposed to administer RBAC. Several of these models [56, 74, 75] rely on an existing hierarchy of user roles, which is then used to define the scope of modifications that administrators can perform on the policy. While these models are well defined, it seems that according to Anita Jones's definition of useful security models [76], they are not useful. In this definition, a security model is useful if it (quoting):

1. accurately and concisely expresses the essence of the phenomena of interest, and
2. tells a system designer or user something he did not know or understand without the model.

The surveys reported by Li and Mao [77] show that these models can not accurately represent the existing administrative practices. Consequently, these models fail on the first part of the above definition. To remedy this problem, Li and Mao [77] propose a principled approach to designing an administrative model for RBAC and show that the resulting model, UARBAC, reflects existing practices in the field. This principled approach has been helpful to us in designing our administrative model for TE (see next Chapter). We have also used this approach to evaluate our administrative model (see Chapter 5).

2.5 Operating System Access Controls

As we mentioned in the introduction of this chapter (see Figure 2.1), the desire for operating systems to offer access control mechanisms was introduced by the apparition of persistent data storage technologies. This desire became a necessity in order to take full advantage of multiprogramming (and later time sharing), without imposing restrictions on the programs that can be run at the same time.

Our goal, as stated in the introduction of this thesis, is to provide access controls that are backwards compatible with existing applications on personal computers. Nowadays, personal computers all run multi-user timesharing OS's[8] [78] that rely on virtual memory to isolate processes [79]. Consequently, our survey of the related work is focused on these OS's. We will use UNIX as a running example to provide a narrative to the security features present on these OS's, in a way that justifies our choice of experimental platform. While presenting these features, we provide references to the original work that these features stem from. We complete this exposition with security extensions to UNIX that do not fit this narrative, as well as security features from other multi-user timesharing OS's and research OS's.

2.5.1 Main Features and Limitations of UNIX Security

The base UNIX model of security relies on the identity of the user to perform access control. The identity of a user is composed of a login name and a set of user groups that the user belongs to. A central notion in UNIX access control is the ownership of resources. By default, when a resource is created, its owner is set to be the user that created it. Additionally, resources are considered owned by a group. By default, the group of a resource is set, at creation time, to the effective group[9] of the user

[8] By extension, we consider tablet computers and smartphones. Current OS's deployed on these platforms are iOS from Apple, Android from Google, and Windows Phone from Microsoft. The first two are multi-user operating systems.

[9] A user can be a member of several groups. Depending on the implementation, the semantics can vary here [80].

that created it. For some resources, the owner can configure their permissions. For instance, the permissions to access a file can be configured by their owner. The base permissions are read, write, and execute (`rwx` permission bits, originated by Daley and Neuman [81]). These permissions can be assigned to three sets of users, from specific to general: the owner of the file, the group that owns the file, and all the other users of the system (these are the `user`, `group`, `others` categories). There are several problems with the base access control model embedded in the original UNIX. Most of these problems have been addressed with successive extensions to the base model. We present these problems and their solutions in turn, and finally discuss one of the currently unaddressed problems, which this research addresses. Namely, super-user privileges are required to configure all the interesting security features of UNIX.

UNIX access control was originally specified in terms of *owners* and *non-owners* [48, 82]. This was later extended to include the notion of groups. Groups allow the sharing of resources among a work group. For instance, it is common practice to create a UNIX group for a team of developers that will share access to a code repository. By making all the files and directories of the repository readable and writable by the group, and giving no permissions to the others, it is possible to privately share the repository within the group. Creating user groups and managing their membership, however, requires superuser privileges. So, even with the group extension, it is still not possible for a regular user to individually specify several users that should have access to a file she owns. This problem, contrary to the ones we present below, has a solution that is available to regular users: POSIX access control lists [83] enable a user to define which users have which kind of access on files she owns, on a file-by-file and user-by-user basis. File access control, nevertheless has issues relating to the granularity of file access permissions, which we describe next.

Early on, it was recognized that the `read` and `write` permissions are too broad in certain contexts. Consider, for example, the file that is used to store encrypted

passwords of users (`/etc/passwd`[10]). Users need a way to write in this file in order to be able to change their password. However, a direct `write` access right to the password file would allow a user to change *any* password in the password file. This would ruin the system security. The `setuid` facility [48] was created to solve a similar problem with accounting files, where the `read` permission was too coarse to restrict users to reading only their accounting data. The `setuid` facility relies on an extra permission bit on executable files, the `setuid` bit. If that bit is set on an executable file, the process resulting from the loading of that file will run under the identity of the owner of the file. This feature allows a user to create programs, that are executable by other users and run under her identity, to mediate access to her data. For instance, a user can change his password (stored in the password file, which is owned by `root`), by running the `passwd` program, which is owned by `root` and has the `setuid` bit set. As we can see, the `setuid` facility was designed to assist in the deployment of access mediation with application-level semantics. It was not designed to confine arbitrary applications: a process, whose identity was set because of the `setuid` bit being set on its program file, can actually revert to its original process identity (the identity of the user that invoked the program). However, if one wants to confine an un-cooperative process by using `setuid`, one can write a wrapper program that will not only change the effective user identity (`euid`) of the process, but also change its real user identifity. Having to write such a wrapper, and many other subtleties make `setuid` usage a delicate exercise [84]. As a matter of fact, confining un-cooperative programs is different from the original `setuid` design goal of allowing a user to set up mediated access to his data. Furthermore, confining a program does actually require creating a new account, under which identity the program will be run. Only a superuser can create accounts on UNIX.

A common example of how a cooperating process is confined by running under a specific identity is the way the printing daemon (`lpd`), is run under a dedicated

[10]On systems with the shadow passwords package, the password file is split between the `/etc/passwd` and `/etc/shadow` files, and encrypted passwords are stored in `/etc/shadow`

identity (`lp`). The goal of such a setting is to avoid running the printing daemon under the super-user identity, in order to limit the damage that can be caused by exploiting a flaw in the printing daemon. UNIX, however, has permissive settings by default [85]. As a result the `lp` user has, by default, access to all the common executables of the system, including shells and potentially compilers. It is desirable to limit this access to prevent privilege escalation [6]. This can be performed by limiting the access to the filesystem that is granted to the printing daemon.

A commonly documented way of limiting filesystem access is to use `chroot`, which changes the directory that a process sees as the root directory of the filesystem. However, `chroot` was not designed as a security feature. It was reportedly designed as a means of testing the compilation of BSD 4.2 [86] (by changing the root of the filesystem, it was possible to clearly establish and guarantee the source code dependencies of the system). `chroot` is therefore not reliable as a security feature, which it wasn't designed to be in the first place. For instance, `chroot` can not be used to confine processes that run as the superuser. The rationale for this weakness is simple: "If you have the ability to use chroot() you are root. If you are root you can walk happily out of any chroot by a thousand other means" [87]. For instance, the `mknod()` system call could be used by a `root` process to create device files and then access the system's hard drives directly [88].

To enforce a proper confinement of processes, including processes running as `root`, `chroot` has to be supplemented with restrictions on the invocation of system calls. This is what the `jail` facility [86] provides. `jail`, like `chroot`, requires super user privileges to be administered. What we also consider to be a problem is that `jail` is a system virtualization technique: `jail` achieves its goal of simplicity by relying on the simple policy of fully isolating the jails. Virtualization, while it solves the problem of preventing untrusted process from accessing the host system, does not solve the problem of mediating interactions between applications [89].

System call interposition has been proposed and implemented in many projects (e.g. Janus [90] and systrace [91]) as a means of providing a flexible access control

mechanism, which can be configured by regular users. This mechanism, although very promising in terms of flexibility, is actually very hard to get right [92]. An example of its weakness is its susceptibility to race conditions, as explained in [91] and practically demonstrated in [93]. Although system call interposition (or even library interposition) seems like an attractive mechanism to support fine-grained access control in userspace, without requiring superuser privileges to configure the access control, this mechanism can not be relied on. It is an instance of "Fortresses built on sand" [94]: even when a coding error does not make its implementation directly vulnerable, its integration in the system will [5].

Besides the granularity of the read and write permissions, another problem with UNIX permissions are their filesystem orientation: since UNIX follows the philosophy that everything is a file, the focus of its access control has been the protection of files, and the protection of all the system abstractions that can be interacted with as if they were files (e.g. disk device, shared memory, and filesystem directory). Unfortunately, this leaves out many interactions that are not covered by the file abstraction. Network operations like connecting a socket, for instance, do not have mappings to the `read`, `write`, and `execute` operations.

Moreover, while `setuid` helps in partitioning the system into subsystems that run under different identities, it remains hard to assure a system whose security rests on proper `setuid` settings [80, 95]. A good part of this problem can be explained by Dennis Ritchie's saying about UNIX: "It was not designed from the start to be secure. It was designed with the necessary characteristics to make security serviceable" [85]. More formally, the problem in assuring `setuid` subsystems is that their security can not be efficiently modeled. Instead, all the file settings, all the potential inputs, and the code of the subsystem have to be analyzed [80].

Type Enforcement (TE) [15] offers a solution to all of the above problems (except the need to be superuser to configure it) while being compatible with UNIX semantics, as successfully demonstrated in [67, 71, 72]. TE solves the problem of the inadequacy of file permissions for other objects: each class of system objects can have different

permissions. For instance, there is a class for network sockets which has `connect` and `bind` permissions, in addition to the `read` and `write` permissions. TE also solves the problem of the modeling of the system to assess its security. In TE, processes run in a domain, which is a notion orthogonal to the UNIX notion of user identity. TE has a deny by default policy which facilitates the reasoning on the system: from a security standpoint, it is therefore easy to tell precisely what the allowed accesses are. In addition to domains, TE provides the notion of domain transitions. Domain transitions are very similar to the change of effective identity that `setuid` causes. The major difference is that TE domains are easier to reason about when assessing the security of the system, because one does not need to look at the permissions of each system object. Instead, system objects are abstracted in terms of types. The only trick is the special meaning of the word "type" when used in the context of TE, where a type is just a label attached to an object to indicate its security relevance (for instance, `shadow_t` is used to label the `shadow` password file in SELinux). What one normally thinks of as a type is called a "class".

TE is an efficient mean of establishing assurance on a system, by modularizing the proofs of correctness, so that proofs of correctness for small elements of the system can be produced by humans, and then composed by humans as well. This is necessary for the social process of proofs to function correctly [47,96]. Also, modern integration of TE in UNIX systems ([68,72,97]) provide a fine granularity of control

Last but not least, SELinux supports labeled networking [98, 99]. Labeled networking consist in using packet filtering criteria [100] to determine the TE type with which a given network packet should be labeled. Labeling network packets enables the specification of TE access control rules on them, which in turns blends the network aspect of the protection state with the rest of the protection state. Typically, this is not the case: the access control rules for network traffic are written directly as part of the configuration of the packet filtering facility, which complicates the assessment of whether the network access controls for an application are appropriate. With TE, this assessment can be done in two divided (and therefore simpler) steps: validating

the labeling configuration, and then validating the accesses granted on these types. TE is therefore an attractive solution to practical UNIX security problems, except that it lacks an administrative model. This is what our research has produced.

2.5.2 Additional UNIX Security Extensions

In the preceding presentation of UNIX security, we have introduced the main components of UNIX security in a manner that justifies the existence of our research. In the following, we correct this bias by presenting other security extensions to UNIX without which our coverage of the related work would not be complete.

Variants of the Bell LaPadula Model

The Bell LaPadula model of security has been integrated in "trusted" versions of several commercial unices, including Trusted AIX (IBM) [101], Trusted Xenix (Trusted Information Systems) [102], and Trusted Solaris (Sun Microsystems) [103]. IX [104] was a research effort to explore the implications of supporting a variant of the Multi Level Security model with UNIX as a base system. IX also supported a variant of the Biba model.

Variants of the Biba Model

Variants of the low water mark policy proposed by Biba [14] have been implemented on Linux, including LOMAC [105] and UMIP [106]. IX [104] did also implement a variant of the Biba model.

Ad-hoc Models

Many ad-hoc extensions to the UNIX security have been proposed. PinUP [107] offers an enhancement on the protection offered by POSIX access control lists: it is

possible to also restrict which application has access to a file. PinUP is implemented as a Linux security module. AppArmor [108] adds mandatory access control on Linux, by defining application profiles. AppArmor profiles can be viewed as a refinement on the `setuid` facility, for two reasons. First, a confinement profile is applied based only on the application being accessed. This is similar to the way `setuid` attaches to a binary the new identity to transition a process to. Second, profiles are mostly focused on filesystem access restrictions. Like `setuid` and UNIX, AppArmor is limited in its ability to enforce fine-grained permissions on the filesystem by not being able to differentiate among the many kinds of objects that can live in the namespace of the filesystem. By applying the profile based on the access path to the application, instead of the information attached to the file's inode, AppArmor can enforce a different policy for the same application, depending on which path it is accessed from (in the case of an inode linked by multiple directory entries). This can either be viewed as a feature or a security flaw. Keeping with the UNIX tradition, AppArmor offers very coarse controls on the network communication of an application: it can restrict which kinds of network connections are allowed (e.g. TCP or UDP), but none of their other characteristics (e.g. port numbers and addresses). AppArmor offers some support for transitions between profiles, with its ability to require a new process to execute under a profiles (the '`p`' and '`P`' access modes). This support is also very similar to `setuid`, in the sense that only one profile can exist per program, contrary to SELinux, where the same program can be the entrypoint to different domains, depending on the source domain (the domain of the process executing the entrypoint).

Linux Security Modules

The Linux Security Module infrastructure (LSM) [109] provides a interface that allows Linux loadable kernel module to extend the access controls performed by the standard kernel. LSM exposes a set of sites in the kernel code where access control decisions are performed. A module that uses LSM extends the security mechanisms of

Linux by registering callbacks for some of these decision points. Many access control developments on Linux have used this interface, including UMIP [106], AppArmor [108], PinUp [107], and SELinux [110].

2.5.3 Other Systems

We now present the security features of other mainstream operating systems.

Microsoft Windows

The discretionary access control model of Microsoft Windows [111] uses access control lists attached to "securable objects". A mandatory access control model called Mandatory Integrity Control (MIC) [112] can be layered on these discretionary controls, and provides a protection against network attacks similar to that of UMIP [106]. Additionally, the administration of the discretionary permissions can be simplified by resorting to the RBAC features that the platform supports [113].

Apple OS X

OS X, from Apple, is a BSD variant of UNIX. As such, it supports the UNIX discretionary access control model and `setuid`. The integration of TE in the TrustedBSD [68] project has been ported to the open source version of OS X, Darwin, and is named SEDarwin [114]. Since version 10.5 codenamed Leopard and released in 2007, OS X has been extended with a sandboxing feature, called "sandbox" that allows overlaying coarse-grained mandatory access control on an application at launch time [115]. This sanboxing is used to secure the network time protocol (NTP) daemon and the document indexer used by the local document search feature.

OpenVMS

OpenVMS from Hewlet Packard (and originally VAX/VMS from Digital Equipment Corporation) is a multi user timesharing operating system that relies on virtual memory to isolate processes from one another. OpenVMS offers the following security features [116]. Protected objects, which are defined as "passive repositories that either contain or receive information", have their security-relevant attributes grouped in a "security profile": the `owner` attribute, used to determine who has discretionary administrative control on the object; the "protection code", which defined for broad groups of users (system, owner, group, and world) the accesses that they have on the object (similarly to SELinux, OpenVMS uses object classes and class-specific operations); an access control list, containing access control entries, which are similar in spirit to (and predate) the POSIX access control lists [83]. OpenVMS also supports the implementation of protected subsystems, which rely subject identity transitions, similar to UNIX `setuid`.

2.5.4 Research Operating Systems Security

While not solved on mainstream UNIX systems, the problems mentioned in our presentation of UNIX security have been solved in research operating systems. These solutions, however, have not been transferred yet to mainstream operating systems. The cost of migrating (and porting) existing applications to these research systems, as well as the limited hardware supported by most of these systems seems to have been dominant factors in preventing their adoption [117].

Capabilities

Capability-based operating systems solve all the granularity and delegation problems described above. Moreover, they solve efficiently the Confused Deputy problem [118]. This has been argued strongly in [119], and demonstrated in practice by

several systems. Two recent examples of capability-based OSes are Asbestos [120] and HiStar [121]. Both are built towards enforcing information flow in a distributed manner. The enforcement is distributed in the sense that the flow is not centrally decided. Instead, users of the system can decide on some of the information flow policy.

Programming Languages Techniques

While traditional OS security relies on a combination of security features provided by the hardware in order to guarantee the integrity of a kernel, Singularity [122] relies only on programming languages techniques to preserve the integrity of its kernel and enforce the mediation of inter-process interaction. Two techniques are relied upon. First, all the code loaded by the operating system is verified for type safety. This guarantees that a program can not perform arbitrary memory references. Second, a global system invariant is enforced: no process can contain a direct reference to an object that is owned by another process. This invariant guarantees that the system will mediate all all inter-process interaction. To an extent, this approach is very similar to the approach used in capabilities-based systems which were able to enforce confinement of processes without resorting to memory protection [123, 124].

2.6 Conclusion

We have presented access control models, and their implementations, that relate to our work. In this presentation, we have shown the reasons that lead us to choose this research path, namely extending TE on SELinux with an administrative model, in order to support our thesis. The reasons were the backwards compatibility of TE with existing applications, the fine granularity of its permissions, and the comprehensiveness of the access controls offered by SELinux, inclusive of network traffic.

3. ADMINISTRATIVE MODEL FOR TYPE ENFORCEMENT

There is little guidance in existing work on how one should go when designing an administrative model for an access control model. A necessary first step, which we present in this chapter, is to enable controlled modifications of the policy. Additionally, the granularity at which accesses are controlled should be as fine as possible, to avoid constraining arbitrarily the possible delegations. The reasoning behind this approach was the following. An administrative model with controls that would be coarse would most likely prevent scenarios from being supported by that model. From a design perspective, a coarse administrative grain would also spoil some of the effort that was put in implementing fine-grained access control in SELinux. A question that naturally arises when considering fine-grained access control is whether a fine granularity of control will result in a significant performance overhead or, at least, a significantly superior overhead. At the time scale of a running system, however, changes of the security configuration are extremely rare. As a result, moderate performance of our administrative model would have been acceptable. Instead, our performance evaluations (see Chapter 5) show that our system is an order of magnitude faster than current approaches when performing the small edits that are required when fine-tuning and debugging a security policy.

In this chapter, we first present the specific variant of Type Enforcement implemented in SELinux. We present a set of formal semantics to ease the reasoning and understanding of its behavior, together with the security policy language used to express the configuration of these mechanisms. Then, we present the administrative model that we designed to control modifications of the policy. We present its semantics, concrete syntax, and how we integrated its implementation in the system.

3.1 Modeling Type Enforcement

In this section, we first present a model for the core features of type enforcement: the accesses allowed within a domain, and the domain transitions. At the same time, we present how the notion of domain transitions is generalized as type transitions in SELinux. This generalization allows the treatment of domain transitions and default labeling of new objects within the same framework. Finally we present extensions of the TE model that are included in SELinux. One of these extensions, type attributes, is particularly useful as it supports both our comparison of TE and Core RBAC (see Section 3.2) and the overlay labeling that we develop in the next chapter (see Chapter 4). Our presentation of these features is such that it enables their composition, as illustrated in Figure 3.4.

3.1.1 Core Type Enforcement Model

"The foundation of any protection system is the idea of different protection environments or contexts. Depending on the context in which a process finds itself, it has certain powers; different contexts have different powers" [7]. Indeed, Type Enforcement (TE) is based on two sets of rules:

- *access vector* rules which specify, based on the type of a process, the operations that this process can perform on objects of the system.
- *type transition* rules which specify how types are assigned to new system objects. This includes typing process objects.

To properly define the access control enforced by the access control system, we first need to define what an access request is, and how it is presented to the access control system. A few preliminary definitions are required before we can define an access request.

Definition 3.1.1 (Class) *System resources are grouped in classes, which define the operations that instances of the resource support. For instance,* `read` *and* `write` *are*

valid operations for files as well as sockets, whereas the `connect` *operation is valid only on a socket object.*

Definition 3.1.2 (Object) *An object is an instance of a resource class.*

Definition 3.1.3 (Object manager) *An object manager is a component of the system that manages a given class (or several classes) of resources. For instance the virtual filesystem manages files, directories, and file links; the X server manages, among other things, the cursor, the selection, and drawable areas. Please note that this also illustrates that object managers can be either kernel-space components (the virtual filesystem) or user-space components (the X server).*

Definition 3.1.4 (Type) *Each object has a type attached to it; a type is a string.*[1] *For instance, regular user processes have the type* `user_t`*, while processes running on behalf of the system administrator have the type* `sysadm_t`*. These are examples of process types; an example of a file type is* `httpd_user_content_t`*, the type attached to the files of the webpage of a user.*

In TE, an object is never considered directly. Instead, an object is considered through its type and class. The class is used to group objects by resource kind, while the type is used to group objects by security domain.

The model we are defining is an abstraction of the behavior of SELinux, where the notions of domain and types have been merged. As a result, automatic domain transitions and the default labeling of new objects have been unified as a single primitive: the automatic labeling of new objects. Consequently, there is no direct notion of a *domain* in SELinux. Instead, a type is considered a domain if it has the `domain` attribute. Attributes are presented in more details in Section 3.1.2.

[1]Contrary to Bell-LaPadula labels, there is no partial order on TE types

Syntax

In the following definitions, let T be the set of types, C be the set of object classes, O be the set of operations that can be performed on objects, I be the set of objects (instances), $\Gamma \subset I \times T$ be the relation that maps an object to its type, and $\Xi \subset I \times C$ be the relation that maps an object to its class. Type attributes can be used interchangeably with types in all places but one (the "new type" field of a type transition rule). As they play an important role in our modeling, we introduce from the beggining syntactic placeholders that can accept either types or type attributes. We call them typoids and represent them individually as θ and their set as Θ.

Definition 3.1.5 (Access request) *An access request is a 5-tuple of the form* $r(\mathtt{p}, \theta, \theta', \mathtt{c}, \mathtt{o}, \mathtt{i})$*, where* $\mathtt{p} \in I$ *is the process, of type* $\theta \in T$ *(the source type), attempting to perform operation* $\mathtt{o} \in O$ *on an instance* $\mathtt{i} \in I$ *of class* $\mathtt{c} \in C$ *and of type* $\theta' \in T$ *(the target type). In some rules, the value of some of these fields is irrelevant. We will indicate that by using a "don't care" character ('_') instead of providing a value for the field. There is a special case for the instance (*$\mathtt{i}$*) and target type (*θ'*) fields: the creation of new objects. In this case, both of these fields refer to the parent object used in the creation (when the creation requires a parent). For instance, when creating new objects on the filesystem (files, directories, named pipes, etc.), the parent object is the directory where the new object is created.*

Definition 3.1.6 (TE access vector rule) *An access vector rule is a 4-tuple of the form* $a(\theta, \theta', \mathtt{c}, \mathtt{o})$*, where* $\theta \in \Theta$ *(the source type) is the type of the process attempting to perform an operation;* $\theta' \in \Theta$ *(the target type) is the type of the object on which the operation is attempted;* $\mathtt{c} \in C$ *is the class of the object on which the operation is attempted;* $\mathtt{o} \in O$ *is the operation being attempted. In early versions of SELinux, any of the fields of an access vector rule could be wildcarded to indicate that its value was irrelevant to the specification of the policy. This is not the case anymore, and our model reflect this fact. Type attributes have been introduced, to enable the designation*

of sets of objects in access vector rules. This is reflected here by the fact that the source and target type of the access vector rule can both take a typoid, which can be either a type or a type attribute.

Definition 3.1.7 (TE type transition rule) *A type transition rule is a 4-tuple of the form* $(\theta_c, \theta_r, \mathtt{c}, \mathtt{t}_n)$*, where* $\theta_c \in \Theta$ *(the current type) is the current type of the object;* $\theta_r \in \Theta$ *(the related type) is the type of a related object;* $\mathtt{c} \in C$ *is the class of the related object;* $\mathtt{t}_n \in T$ *(the new type) is the type the object will have after the transition. As with access vector rules, the source type and target types can be replaced by an attribute, hence our usage of typoids in the definitions of these fields. The new type, however, can not be replaced by a type attribute. Indeed, a type attribute designates a* set *of types, whereas a type transition must specify the* single *type that the object will bear after the type transition.*

Semantics

In our exposition of the semantics, we are using *stuck semantics* to simplify the representation of the dynamics of the system. The access control model is viewed as receiving a stream of access requests, represented by the list $\mathtt{R}$, out of which the first element, ρ, is picked for evaluation. With stuck semantics, only the allowed state transitions, resulting from the successful evaluation of allowed (valid) access requests, are represented. Denied (invalid) state transitions are implicitly represented by their omission. As a result, if the stream of access requests contains an invalid request, the system will enter a stuck state. For example (see Figure 3.2), an attempt to execute a file (assuming the requesting process has the `execute` but not the `execute_no_trans` permission on the file) will fail if either there is no automatic type transition that matches the request or if the matching type transition is not explicitly allowed. Both failures are implicitly represented in the semantics.

In a nutshell, access vector rules specify the accesses that will be allowed on the system, while type transition rules specify how newly created objects will be typed.

TE-eval

Syntax

access_vector_rule :=	$a(\theta, \theta', c, o)$
type_transition_rule :=	$n(\theta, \theta', c, t'')$
access_request :=	$r(p, t, t', c, o, i)$
R :=	$\rho R \mid \varnothing$
θ :=	$t \mid \alpha$
T :=	$\{t\}$
A :=	$\{\alpha\}$
Θ :=	$\{\theta\}$
Γ :=	$\{(i, t)\}$
Ξ :=	$\{(i, c)\}$
Ω :=	$\{o\}$
Ψ :=	$\{a\} \cup \{n\}$

Metavariables

t	*type*
α	*type attribute*
θ	*typoid*
c	*class*
o	*operation*
i, j, p, x	*instances* (j is a new filesystem object; p is a process; x is a security context)
ρ	access_request
a	access_vector_rule
n	type_transition_rule
Ω	*transition operations*
Γ	*object to type mapping*
Ψ	*policy*

Semantics

$$\frac{\Omega = \varnothing \qquad a(s, t, c, o) \in \Psi}{\Psi \vdash r(\_, s, t, c, o, \_)} \text{ (TE-ACCESS-BASE)}$$

$$\frac{r = r(p, s, t, c, o, i) \qquad o \notin \Omega \qquad \Psi \vdash r}{\Psi, rR \to \Psi, R} \text{ (EVAL)}$$

Figure 3.1.: TE semantics for simple accesses. Our model follows the semantics of recent version of SELinux (at least past Linux kernel 2.6.30) where wildcards are not supported in rules anymore, but types attributes can be used instead. We describe type attributes in Section 3.1.2, with the other extensions that SELinux contributed to TE.
We are using an explicit set of transition operations, Ω, which will allow us to extend this base access model with transitions in the next two semantics: transitions on subjects (see Figure 3.2) and transitions on objects (see Figure 3.3).

TE-domain-transitions

Syntax

$$
\begin{array}{rl}
\texttt{access\_vector\_rule} := & a(\theta, \theta', \texttt{c}, \texttt{o}) \\
\texttt{type\_transition\_rule} := & n(\theta, \theta', \texttt{c}, \texttt{t}'') \\
\texttt{access\_request} := & r(\texttt{p}, \texttt{t}, \texttt{t}', \texttt{c}, \texttt{o}, \texttt{i}) \\
\texttt{R} := & \rho\texttt{R} \mid \varnothing \\
\theta := & \texttt{t} \mid \alpha \\
\texttt{T} := & \{\texttt{t}\} \\
\texttt{A} := & \{\alpha\} \\
\Theta := & \{\theta\} \\
\Gamma := & \{(\texttt{i}, \texttt{t})\} \\
\Xi := & \{(\texttt{i}, \texttt{c})\} \\
\Omega := & \{\texttt{o}\} \\
\Psi := & \{\texttt{a}\} \cup \{\texttt{n}\}
\end{array}
$$

Metavariables

$\texttt{t}$	*type*
α	*type attribute*
θ	*typoid*
$\texttt{c}$	*class*
$\texttt{o}$	*operation*
$\texttt{i}, \texttt{j}, \texttt{p}, \texttt{x}$	*instances* ($\texttt{j}$ is a new filesystem object; $\texttt{p}$ is a process; $\texttt{x}$ is a security context)
ρ	$\texttt{access\_request}$
$\texttt{a}$	$\texttt{access\_vector\_rule}$
$\texttt{n}$	$\texttt{type\_transition\_rule}$
Ω	*transition operations*
Γ	*object to type mapping*
Ψ	*policy*

Semantics

$$\Omega = \{\texttt{execute}, \texttt{setcurrent}\}$$

$$\frac{\begin{array}{c}\texttt{r} = r(\_, \texttt{t}, \texttt{t}', \texttt{c}, \texttt{o}, \_) \\ \texttt{o} \notin \Omega \\ \Psi \vdash \texttt{r}\end{array}}{\Psi, \Gamma, \rho\texttt{R} \rightarrow \Psi, \Gamma, \texttt{R}} \quad (\textsc{Eval})$$

$$\frac{\begin{array}{c}\texttt{r} = r(\_, \texttt{t}, \texttt{t}', \texttt{file}, \texttt{execute}, \_) \\ \Psi \vdash \texttt{r} \\ \nexists \texttt{t}'' \textit{ s.t. } (\texttt{t}, \texttt{t}', \texttt{process}, \texttt{t}'') \in \Psi \\ a(\texttt{t}, \texttt{t}', \texttt{file}, \texttt{execute\_no\_trans}) \in \Psi\end{array}}{\Psi, \Gamma, \rho\texttt{R} \rightarrow \Psi, \Gamma, \texttt{R}}$$
(Eval-exec-simple)

$$\frac{\begin{array}{c}\texttt{r} = r(\texttt{p}, \texttt{t}, \texttt{t}', \texttt{file}, \texttt{execute}, \_) \\ \Psi \vdash \texttt{r} \\ \exists! \texttt{t}'' \textit{ s.t. } n(\texttt{t}, \texttt{t}', \texttt{process}, \texttt{t}'') \in \Psi \\ a(\texttt{t}, \texttt{t}', \texttt{process}, \texttt{transition}) \in \Psi \\ a(\texttt{t}, \texttt{t}', \texttt{file}, \texttt{entry\_point}) \in \Psi\end{array}}{\Psi, \Gamma, \texttt{rR} \rightarrow \Psi, (\Gamma \backslash \{(\texttt{p}, \texttt{t})\}) \cup \{(\texttt{p}, \texttt{t}'')\}, \texttt{R}}$$
(Eval-exec-trans)

$$\frac{\begin{array}{c}\texttt{r} = r(\texttt{p}, \texttt{t}, \texttt{t}', \texttt{process}, \texttt{setcurrent}, \texttt{x}) \\ \Psi \vdash \texttt{r} \\ \textit{context2type}(\texttt{x}) = \texttt{t}'' \\ a(\texttt{t}, \texttt{t}, \texttt{process}, \texttt{setcurrent}) \in \Psi \\ a(\texttt{t}, \texttt{t}'', \texttt{process}, \texttt{dyntransition}) \in \Psi\end{array}}{\Psi, \Gamma, \texttt{rR} \rightarrow \Psi, (\Gamma \backslash \{(\texttt{p}, \texttt{t})\}) \cup \{(\texttt{p}, \texttt{t}'')\}, \texttt{R}}$$
(Eval-exec-dyn-trans)

Figure 3.2.: TE semantics for domain transitions (type transitions on process objects, when they start executing a new binary): TE-domain-transitions. In SELinux, domain transitions are implemented as type transitions on process objects. The transition is triggered when the policy contains a type transition rule that matches the request by a process, of type `s`, to execute a file of type `t`. The details of a successful transition upon `exec()` are given in rule Eval-exec-trans. Another kind of transition which does not require executing another binary, a dynamic transition, is possible. Dynamic transitions are represented in rule Eval-exec-dyntrans. The details of an allowed call to `exec()`, without transition, are given in rule Eval-exec-simple.

TE-object-transitions

Syntax

access_vector_rule :=	$a(\theta, \theta', c, o)$
type_transition_rule :=	$n(\theta, \theta', c, t'')$
access_request :=	$r(p, t, t', c, o, i)$
R :=	$\rho R \mid \varnothing$
θ :=	$t \mid \alpha$
T :=	$\{t\}$
A :=	$\{\alpha\}$
Θ :=	$\{\theta\}$
Γ :=	$\{(i, t)\}$
Ξ :=	$\{(i, c)\}$
Ω :=	$\{o\}$
Ψ :=	$\{a\} \cup \{n\}$

Metavariables

t	*type*
α	*type attribute*
θ	*typoid*
c	*class*
o	*operation*
i, j, p, x	*instances* (j is a new filesystem object; p is a process; x is a security context)
ρ	access_request
a	access_vector_rule
n	type_transition_rule
Ω	*transition operations*
Γ	*object to type mapping*
Ψ	*policy*

Semantics

$$\frac{\begin{array}{c}\Omega = \{\texttt{create}\} \\ \texttt{fs\_objects} = \\ \{\texttt{dir}, \texttt{file}, \texttt{link\_file}, \texttt{socket\_file}, \texttt{fifo\_file}\}\end{array}}{}$$

$$\frac{\begin{array}{c}r = r(t, t', c, o, i) \\ o \notin \Omega \\ \Psi \vdash \rho\end{array}}{\Psi, \Gamma, \rho R \rightarrow \Psi, \Gamma, R} \quad \text{(Eval)}$$

$$\frac{\begin{array}{c}r = r(\_, t, t', c, \texttt{create}, i) \\ c \in \texttt{fs\_objects} \\ (i, \texttt{dir}) \in \Xi \\ (i, t') \in \Gamma \\ \nexists t'' s.t.\ n(t, t', c, t'') \in \Psi \\ a(t, t', \texttt{dir}, \{\texttt{search}, \texttt{write}, \texttt{add\_name}\}) \in \Psi \\ a(t, t, c, \{\texttt{create}, \texttt{link}, \texttt{write}\}) \in \Psi \\ a(t, \texttt{fs\_t}, \texttt{filesystem}, \{\texttt{associate}\}) \in \Psi\end{array}}{\Psi, \Gamma, \rho R \rightarrow \Psi, \Gamma \cup \{(j, t)\}, R}$$

(Eval-create-simple)

$$\frac{\begin{array}{c}r = r(\_, t, t', c, \texttt{create}, i) \\ c \in \texttt{fs\_objects} \\ (i, \texttt{dir}) \in \Xi \\ (i, t') \in \Gamma \\ \exists! t'' s.t.\ n(t, t', c, t'') \in \Psi \\ a(t, t', \texttt{dir}, \{\texttt{search}, \texttt{write}, \texttt{add\_name}\}) \in \Psi \\ a(t, t'', c, \{\texttt{create}, \texttt{link}, \texttt{write}\}) \in \Psi \\ a(t'', \texttt{fs\_t}, \texttt{filesystem}, \{\texttt{associate}\}) \in \Psi\end{array}}{\Psi, \Gamma, \rho R \rightarrow \Psi, \Gamma \cup \{(j, t'')\}, R}$$

(Eval-create-trans)

Figure 3.3.: TE semantics for filesystem type transitions: TE-object-transitions.

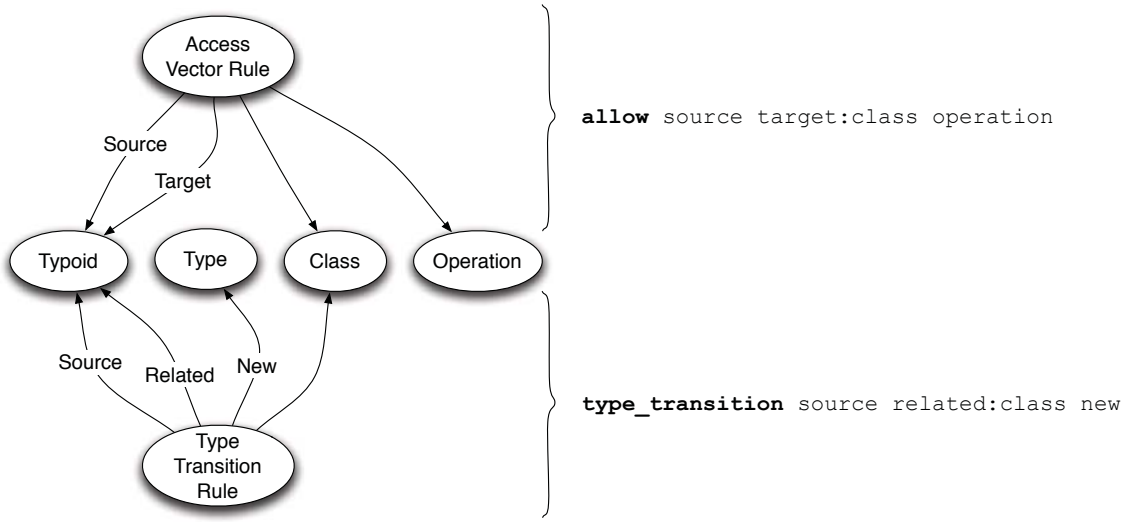


Figure 3.4.: Concrete syntax of the base TE model

Formal semantics are provided in Figure 3.1 for simple accesses, in Figure 3.2 for domain transitions, and in Figure 3.3 for the automatic labeling of filesystem objects. An informal description of the semantics follows.

Semantics 3.1.1.1 (Authorizing accesses) *An access request is authorized if and only if the policy contains an access vector rule that matches the request. Simple semantics for the matching are described by the rule* TE-ACCESS-BASE *in Figure 3.1; semantics that accept type attributes in the access vector rules are described by the set of of* TE-ACCESS-* *rules in Figure 3.5. Example*[2]*:*

```
allow chkpwd_t shadow_t:file { getattr open read };
```

The above example specifies that a process running in the `user_chkpwd_t` *domain (the domain used to validate user passwords) can perform the operations* `getattr` *and* `read` *on files (objects of class* `file`*) of type* `shadow_t`*. In other words, this example specifies that programs running in the* `user_chkpwd_t` *domain can read* `/etc/shadow`*.*

Semantics 3.1.1.2 (Automatic labeling of new objects) *When a process creates a new object, the default behavior of the system is to label this new object with the type of the process that created it. It is always so, unless a type transition rule specifies otherwise. There are two interpretations of a type transition rule* $(\mathtt{t_c}, \mathtt{t_r}, \mathtt{c}, \mathtt{t_n})$*:*

- **Filesystem object labeling**[3]*: This interpretation is used to automatically attach a specific type to objects created on the filesystem. The meaning of the fields is then the following:* $\mathtt{t_c}$ *is the type of the process creating the filesystem object;* $\mathtt{t_r}$ *is the type of the directory in which the file is created;* `c` *is the class of object being created;* $\mathtt{t_n}$ *is the type that will be assigned to the object (provided such a transition is allowed by an access vector rule).*

[2]This example is written using the concrete syntax used to write rule in the SELinux policy language. This concrete syntax is illustrated in Figure 3.4.

[3]by filesystem object, we mean an object that is accessible through the filesystem namespace, e.g. a file, a directory, a UNIX socket, a device, etc.

Example:

```
type_transition passwd_t tmp_t:file passwd_tmp_t
```

The above example specifies that when a process running in the `passwd_t` *domain (the domain of the* `passwd` *program) creates an object of class* `file` *(a file) in a directory of type* `tmp_t`*, then that file should be labeled with the type* `passwd_tmp_t`*. In other words, this example specifies that when files are created in the directory* `/tmp` *by a process that runs in the* `passwd_t` *domain, these files should be labeled as temporary password files, of type* `passwd_tmp_t`*.*

- **Domain transition***: This interpretation is used to automatically attach a new type to a process which, after a successful call to* `exec()`*, starts executing a new program. In other words, this interpretation of transition rules is used to automatically place processes in specific domains, which depend on the type attached to the process (before the transition) and the type attached to the file being executed.* $\texttt{t}_\texttt{c}$ *is the type of the process that is calling* `exec()`*, the class* `c` *of object being created is* `process` *and* $\texttt{t}_\texttt{r}$ *is the type of the executable file used as an argument to the* `exec` *system call.*

  Example:

  ```
  type_transition init_t apache_exec_t:process apache_t
  ```

  The above example specifies that when a process running in the `init_t` *domain (the domain of the* `init` *daemon) starts executing code based on an executable file of type* `apache_exec_t` *(the type of the executable file for the apache web server), then this process should automatically be transitioned to the domain* `apache_t` *(the domain of the apache web server daemon). In other words, this rule specifies that when the* `init` *daemon starts the apache web server, the web server is automatically placed in its own confinement domain which is* `apache_t`

Remark 1 *Since Type Enforcement follows the principle of full mediation and has a default policy of denying accesses, type transition rules need to have matching access*

vector rules for the specified labeling to happen. Type transitions are otherwise denied by default, as any other operation that is not explicitly allowed.

For an automatic domain transition to be allowed, several permissions are actually required, as represented in the rule EVAL-EXEC-TRANS *(Figure 3.2): the type transition from the current process type to the target type of the type transition needs to be allowed, and the type of the program to which the transition is attached must also be an authorized entry point into the target domain. Please note that executing programs without a domain transition requires the specific* `exec_no_trans` *permission, which is separate from the* `execute` *permission, as represented in the rule* EVAL-EXEC-SIMPLE *(Figure 3.2)*

While type transitions for filesystem objects are triggered by the addition of filesystem objects in a directory, several permissions are required for the addition of the object in the directory to be allowed, besides the permission to add a name in the directory (`add_name`). The permission to `search` the directory is required, as a search of the directory is required upon creation of the file to prevent the creation of two entries in the directory with the same name. The permission to `write` the modified directory is also required to save the modified directory object.

3.1.2 Type Enforcement extensions in SELinux

In SELinux, Type Enforcement has received several extensions. Types can be labeled with type attributes, to factor policy rules. Types can also be aliased, to facilitate backwards compatible evolutions of the security policy. Finally, SELinux supports a version of role-based access control adapted to TE, where a role definition constrains the set of domains yjsy subjects can enter when they are members of that role. In this section, we present how these features fit in the model of TE developed in the previous sections.

Type Attributes

To represent type attributes, we need to extend the configuration of the system with a set of attributes A and two relations to map attributes to types and types to attributes: $type2attr : T \to \mathcal{P}(A)$ and $attr2type : A \to \mathcal{P}(T)$

Definition 3.1.8 (Type Attribute) *A type attribute is a string attached to a type. Type attributes can be used instead of types in access vector rules.*

Semantics 3.1.2.1 (Type Attribute) *Type attributes are used to group types in the security policy. By writing access vector rules on type attributes, general aspects of the policy can be written once for all types that bear the same attribute. This can be used either to give the same access from different domains (e.g. write to the system log socket), or to give the same domain access to different types (e.g.* `logrotate` *can rotate the logs of different daemons).*

Type Aliases

From a modeling perspective, we represent type aliases as types, and we extend the configuration of the system with a relation that maps a type to its aliases: $type2aliases : T \to \mathcal{P}(T)$

Definition 3.1.9 (Type Alias) *A type alias defines a secondary name for an existing type. As such, a type alias can be used wherever a type is expected.*

Semantics 3.1.2.2 (Type Alias) *Type aliases, as indicated by their name, are just aliases. They simply and only provide an alternative name to an already existing type. We have not made a separate figure to represent the semantics of type aliasing. From a modeling perspective, the use of type aliases when enforcing access control is the same as the use of type attributes described in Figure 3.5. The semantics for type aliases are obtained from these semantics by having α be a type alias variable and by substituting type2aliases for type2attributes in the rules.*

TE-type-attributes

Syntax

access_vector_rule :=	$a(\theta, \theta', \mathtt{c}, \mathtt{o})$
type_transition_rule :=	$n(\theta, \theta', \mathtt{c}, \mathtt{t}'')$
access_request :=	$r(\mathtt{p}, \mathtt{t}, \mathtt{t}', \mathtt{c}, \mathtt{o}, \mathtt{i})$
R :=	$\rho\mathtt{R} \mid \varnothing$
θ :=	$\mathtt{t} \;\; \alpha$
T :=	$\{\mathtt{t}\}$
A :=	$\{\alpha\}$
Θ :=	$\{\theta\}$
Γ :=	$\{(\mathtt{i}, \mathtt{t})\}$
Ξ :=	$\{(\mathtt{i}, \mathtt{c})\}$
Ω :=	$\{\mathtt{o}\}$
Ψ :=	$\{\mathtt{a}\} \cup \{\mathtt{n}\}$

Metavariables

t	*type*
α	*type attribute*
θ	*typoid*
c	*class*
o	*operation*
i, j, p, x	*instances*
	(j is a new filesystem object; p is a process; x is a security context)
ρ	access_request
a	access_vector_rule
n	type_transition_rule
Ω	*transition operations*
Γ	*object to type mapping*
Ψ	*policy*

Semantics

$$\frac{\Omega = \varnothing}{\Psi \vdash r(\_, \mathtt{t}, \mathtt{t}', \mathtt{c}, \mathtt{o}, \_)}$$

$$\frac{\mathtt{r} = r(\_, \mathtt{t}, \mathtt{t}', \mathtt{c}, \mathtt{o}, \_) \quad \mathtt{o} \notin \Omega \quad \Psi \vdash \rho}{\Psi, \rho\mathtt{R} \to \Psi, \mathtt{R}} \text{ (EVAL)}$$

$$\frac{a(\mathtt{t}, \mathtt{t}', \mathtt{c}, \mathtt{o}) \in \Psi}{\Psi \vdash r(\_, \mathtt{t}, \mathtt{t}', \mathtt{c}, \mathtt{o}, \_)} \text{ (TE-ACCESS-BASE)}$$

$$\frac{a(\alpha, \mathtt{t}', \mathtt{c}, \mathtt{o}) \in \Psi \quad \alpha \in type2attr(\mathtt{t})}{\Psi \vdash r(\_, \mathtt{t}, \mathtt{t}', \mathtt{c}, \mathtt{o}, \_)} \text{ (TE-ACCESS-ATTR1)}$$

$$\frac{a(\mathtt{t}, \alpha', \mathtt{c}, \mathtt{o}) \in \Psi \quad \alpha' \in type2attr(\mathtt{t}')}{\Psi \vdash r(\_, \mathtt{t}, \mathtt{t}', \mathtt{c}, \mathtt{o}, \_)} \text{ (TE-ACCESS-ATTR2)}$$

$$\frac{a(\alpha, \alpha', \mathtt{c}, \mathtt{o}) \in \Psi \quad \alpha \in type2attr(\mathtt{t}) \quad \alpha' \in type2attr(\mathtt{t}')}{\Psi \vdash r(\_, \mathtt{t}, \mathtt{t}', \mathtt{c}, \mathtt{o}, \_)} \text{ (TE-ACCESS-ATTR3)}$$

Figure 3.5.: TE semantics for accesses with type attributes: TE-type-attributes. Type attributes are described in Section 3.1.2. Type attributes are used to factor policy rules.

Roles

To represent roles, we need to extend the configuration of the system with a set of of roles R, a relation to map roles to the domain they are allowed into: $role2types : R \to \mathcal{P}(T)$, and relation that maps processes to their current role: $currentrole : P \to R$.

Definition 3.1.10 (Role) *A role is a set of types.*

Semantics 3.1.2.3 (Role) *The role is one of the components of the security context that SELinux considers when evaluating an access control request. As in role-based access control, roles in SELinux define a set of functions that a user can perform. SELinux being built around Type Enforcement, these functions are defined as TE domains.*

Roles impose an additional constraint on domain transitions. For a domain transition to be allowed, the new type needs to be an element of the set of types defined by the current role of the process. This can be encoded by adding the extra requirement

$$\mathtt{t}'' \in role2types(currentrole(\mathtt{p}))$$

to the rules EVAL-EXEC-TRANS *and* EVAL-EXEC-DYN-TRANS *in Figure 3.2.*

Conditional Rules

Conditional rules, sometimes referred to as "booleans" because of the booleans that condition their activation, are an extension that was added to SELinux to support a simple policy configuration mechanism. The idea is that some security decisions can be formulated in a "checkbox" binary style. Booleans are a representation of these decisions, and boolean expressions that can encompass several of these decisions are used to select blocks of the policy accordingly. For instance, the `ping` program that is used to check network connectivity relies on ICMP sockets to perform some network

diagnostics. Under the regular UNIX access control model, the use of ICMP sockets is considered a privileged operation, and therefore is reserved to the superuser. When relying only on base UNIX security, the decision to let regular users use `ping` with ICMP sockets is performed by using the `setuid` facility to have `ping` run automatically as root, or not.

With the way Type Enforcement is integrated in SELinux, if regular users are to use `ping`, then the program still needs to be installed with `root` as the owner and the `setuid` bit set. Conditional rules support changing at runtime (as opposed to installation time) the decision of whether to let regular users use `ping`. In the policy shipped with RedHat Fedora Core 10, the ability for regular users to use ping is guarded by the `user_ping` boolean.

Definition 3.1.11 (Conditional Rule) *A conditional rule is a rule that is guarded by a boolean expression*

Semantics 3.1.2.4 (Conditional Rule) *The semantics of conditional rules are the same as the semantics of regular rules, except that conditional rules are considered in the process of evaluating access requests only if their guard evaluates to true. This can be modeled by adding booleans (and their values) to the policy* Ψ *in our model and extending rules so that they contain a boolean expression that indicates their validity (unconditional rules contain the special boolean* `true` *that always evaluates to the value "true"). The evaluation process is then rewritten so that rules are considered only if the boolean expression they contain evaluates to the value "true".*

Bounded Types

Bounded types are a generalization of hierarchical types, which are currently supported as bounded types. The idea of bounded types is to constrain the permissions that a type can exert. If a type `t` is bounded by type $\mathtt{t_b}$, then a subject of type `t` can not exert more permissions than the permissions available to a subject of type $\mathtt{t_b}$. Bounded types can be represented by a relation $boundingtype : T \rightarrow T \cup \top$. In

that relation, types are either bounded by another type, or not. For a given type $\mathtt{t}$, this last case is represented by inserting the $(\mathtt{t}, \top)$ in the relation.

Definition 3.1.12 (Bounded Type) *A bounded type is a type whose maximum effective permissions as a subject type are bounded by the effective permissions of another type.*

Semantics 3.1.2.5 (Bounded Type) *Bounded types introduce recursion in the evaluation of permissions. For a bounded type, an access request is allowed only if the corresponding permission is granted to both the type and its bounding type. The bounding type may itself be bounded, hence a recursive evaluation of permissions is needed. The recursion is defined as follow:*

$$\frac{}{\Psi \vdash_{bound} r(\_, \top, \mathtt{t}, \mathtt{c}, \mathtt{o}, \_)} \quad (\text{TE-BOUND-BASE})$$

$$\frac{\Psi \vdash r(\_, \mathtt{s}, \mathtt{t}, \mathtt{c}, \mathtt{o}, \_) \qquad \Psi \vdash_{bound} r(\_, boundingtype(\mathtt{s}), \mathtt{t}, \mathtt{c}, \mathtt{o}, \_)}{\Psi \vdash_{bound} r(\_, \mathtt{s}, \mathtt{t}, \mathtt{c}, \mathtt{o}, \_)} \quad (\text{TE-BOUND-BOUNDED})$$

3.1.3 Summary

In this section, we have presented TE formally, with formal semantics and the concrete syntax that is used in SELinux. To recap, we have illustrated in Figure 3.4 how the different semantics that we have presented can be composed, to form what we consider the core features of TE, TE-CORE. In the following section, we will compare these features to those of Core RBAC.

3.2 Comparative Modeling of RBAC

In this section we model RBAC using the same formalism that we used to model TE. The goal is to clarify what RBAC provides with respect to the requirements of

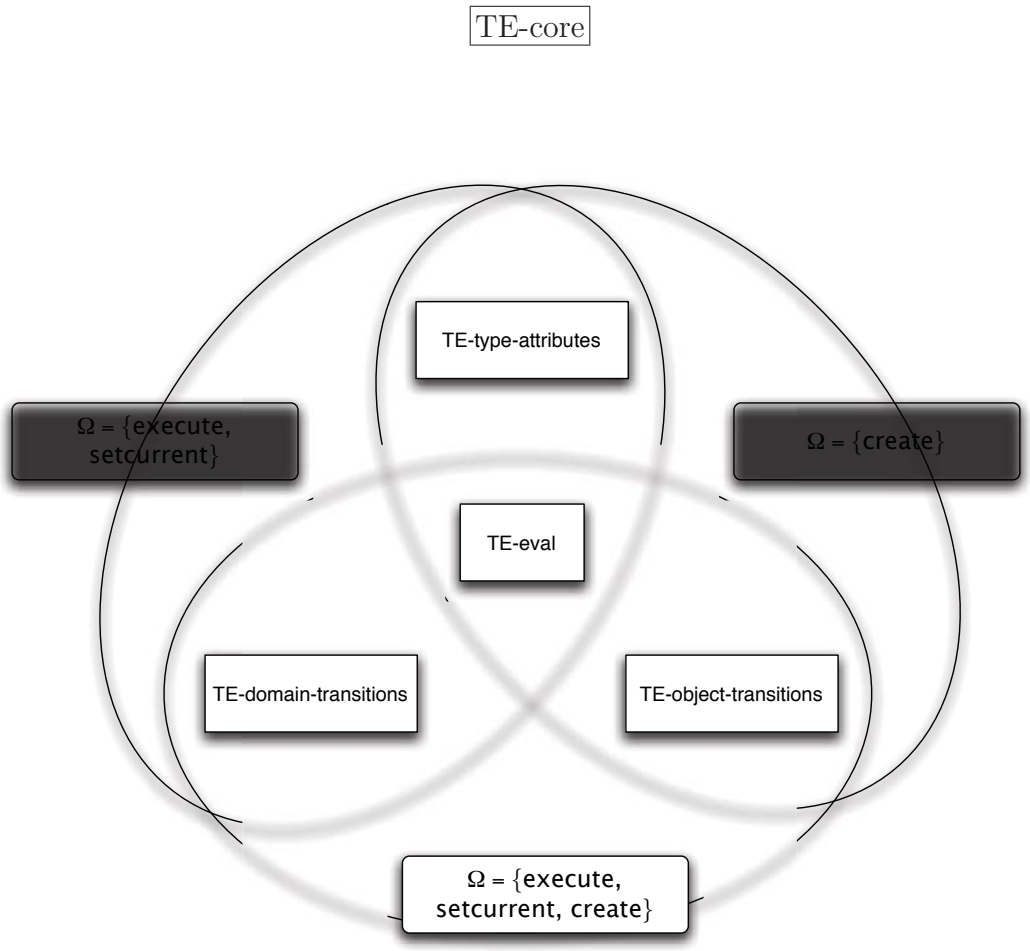


Figure 3.6.: Compositions of the core features of TE: TE-core. The features of TE that we have presented previously, TE-eval (see Figure 3.1), TE-domain-transitions (see Figure 3.2), TE-object-transitions (see Figure 3.3), and TE-type-attributes (see Figure 3.5), can be composed as illustrated here. TE-eval is the nucleus of this set of features, on which the other features can be added. The only thing that is necessary for this composition is to extend the set of operations for which the base evaluation rule (Eval) does not apply. When introducing domain transitions with TE-domain-transitions, the operations on which domain transitions can be triggered (`execute` and `setcurrent`) have to be handled by the evaluation that is specific to these operations. Similarly, introducing transitions on filesystem objects creation in TE-object-transitions requires that the `create` operation be handled as a special case. We have represented the set of special-cased operations Ω for each features that we presented. When composing features, the union of the Ω sets of each feature has to be used, so that all the evaluations that are supposed to have a side effect are properly redirected to the evaluation rule that supports them.

the problem we set to solve (see Chapter 1). We first model RBAC as defined in the NIST standard [55]. Then we proceed to exhibit a mapping of RBAC on top of TE. Finally we show that the reverse mapping is not achievable. We discuss why the reverse mapping is not possible and the impact that this has on our requirements.

3.2.1 Modeling RBAC

We model Core RBAC as it is defined in the NIST standard [55], but using the same formalism that we use to model the core part of TE. Similarly to the TE semantics, we use a stuck semantics evaluation[4]. There are three cases in this evaluation (see Figure 3.7): a regular access request, a role activation, and a role deactivation.

In this model, we did not include administrative operations that can be considered part of the core RBAC model. These excluded operations include the creation and destruction of session, user, roles, and permissions. The creation of objects is not included in our model either, which conforms to our claim of modeling core RBAC: core RBAC does not model object creation either. We now proceed to constructing a mapping from RBAC configurations to TE configurations.

3.2.2 Mapping RBAC to TE

Since RBAC has such a prominent place in the existing access control literature, it is natural to wonder how RBAC and TE compare to each other. We chose to compare them by showing how one model can express the other one. In this section, we show a mapping from Core RBAC (as modeled in Figure 3.7), to TE-core (as modeled in Figure 3.6), where TE is used to emulate RBAC. In other words this mapping is such that an access is allowed in TE, by the TE configuration, if and only if it would have been allowed in RBAC by the original RBAC configuration.

[4]We explain what stuck semantics are at the beginning of Section 3.1.1.

Syntax

$$
\begin{array}{rl}
\mathit{permission\_assignment} := & p(\mathtt{r}, \mathtt{ob}, \mathtt{op}) \\
\mathit{role\_activation} := & \mathit{act}(\mathtt{s}, \mathtt{u}, \mathtt{r}) \\
\mathit{role\_assignment} := & \mathit{ass}(\mathtt{r}, \mathtt{u}) \\
\mathit{access\_request} := & r(\mathtt{s}, \mathtt{ob}, \mathtt{op}) \\
\mathit{Act} := & \{\mathtt{act}\} \\
\mathit{Ass} := & \{\mathtt{ass}\} \\
P := & \{\pi\} \\
\Psi := & \mathit{Act} \cup \mathit{Ass} \cup P
\end{array}
$$

Metavariables

s	*session*
u	*user*
r	*role*
op	*operation*
ob	*object*
π	permission_assignment
act	role_activation
ass	role_assignment
Ψ	*policy*

Semantics

$$\frac{\rho = r(\mathtt{s}, \mathtt{ob}, \mathtt{op}) \quad \mathtt{op} \notin \{\mathtt{activate}, \mathtt{deactivate}\} \quad \Psi \vdash \rho}{\Psi, \rho\mathtt{R} \to \Psi, \mathtt{R}}\ \text{(RBAC-Eval-base)}$$

$$\frac{\rho = r(\mathtt{s}, \mathtt{ob}, \mathtt{op}) \quad \mathtt{ob} = \mathtt{r} \quad \mathtt{op} = \mathtt{deactivate} \quad \mathit{ass}(\mathtt{r}, \mathtt{u}) \in \mathit{Ass} \quad \mathit{act}(\mathtt{s}, \mathtt{u}, \mathtt{r}) \in \mathit{Act}}{\Psi, \rho\mathtt{R} \to \Psi \setminus \{\mathit{act}(\mathtt{s}, \mathtt{u}, \mathtt{r})\}, \mathtt{R}}\ \text{(RBAC-eval-deactivate)}$$

$$\frac{\rho = r(\mathtt{s}, \mathtt{ob}, \mathtt{op}) \quad \mathtt{ob} = \mathtt{r} \quad \mathtt{op} = \mathtt{activate} \quad \mathit{ass}(\mathtt{r}, \mathtt{u}) \in \mathit{Ass} \quad \mathit{act}(\mathtt{s}, \mathtt{u}, \mathtt{r}) \notin \mathit{Act}}{\Psi, \rho\mathtt{R} \to \Psi \cup \{\mathit{act}(\mathtt{s}, \mathtt{u}, \mathtt{r})\}, \mathtt{R}}\ \text{(RBAC-eval-activate)}$$

$$\frac{p(\mathtt{r}, \mathtt{ob}, \mathtt{op}) \in P \quad \mathit{act}(\mathtt{s}, \mathtt{u}, \mathtt{r}) \in \mathit{Act}}{\Psi \vdash r(\mathtt{s}, \mathtt{ob}, \mathtt{op})}\ \text{(RBAC-access)}$$

Figure 3.7.: Semantics for Core RBAC, as defined in the NIST standard [55]. These semantics model access checks (RBAC-access), role activations (RBAC-eval-activate), and role deactivations (RBAC-eval-deactivate).

To obtain a mapping from one access control model (RBAC) to the other one (TE), we need to produce a mapping from the different parts of the source model to the different parts of the target model. The parts that have to be mapped are the objects, their operations, the access control domains and their attributes, the permissions granted to the domains, and the state transitions.

For the mapping of objects and operations, we map each RBAC object to a TE type and the operations are mapped directly.

$$\mathtt{ob} \rightarrow \mathtt{t}_{\mathtt{ob}}$$

$$\mathtt{op} \rightarrow \mathtt{o}_{\mathtt{op}}$$

The permissions granted to an RBAC session are granted based on the active roles of that session. We start by mapping roles and their permissions to types and their permissions. More precisely, we map the roles to type attributes. using type attributes allows us to factor the mapping of permissions as shown below. Also, please note that the object class is not specified in the mapped TE rule, to conform to the RBAC standard[5].

$$p(\mathtt{r}_i, \mathtt{ob}, \mathtt{op}) \in P \rightarrow a(\alpha_i, \mathtt{t}_{\mathtt{ob}}, \_, \mathtt{o}_{\mathtt{op}}) \in \Psi$$

Now that we have a mapping of role permissions to type alias permissions, we proceed to construct the TE subject types that mirror the RBAC sesssion states. First, we need a mapping from each possible session state to a type. This is achieved by enumerating the possible session states. To that extent we first define what we mean by the state $\sigma(\mathtt{s})$ of a session $\mathtt{s}$. Based on this definition, we can define an

[5]An RBAC model with object classes, such as UARBAC [77], would result in the TE object class being used in the resulting TE access vector rules.

auxilliary function $mid(\sigma(\mathtt{s}))$ that takes a session state $\sigma(\mathtt{s})$ and generates a mapping type identifier for it. With n_r roles, these definitions and the mapping are as follows.

$$\sigma(\mathtt{s}) = \{act(\mathtt{s},\mathtt{u},\mathtt{r})\} \text{ s.t. } act(\mathtt{s},\mathtt{u},\mathtt{r}) \in Act$$
$$mid(\sigma(\mathtt{s})) = (\mathtt{u}, b_1..b_i..b_{n_r} \text{ bits s.t. } b_i = 1 \Leftrightarrow act(\mathtt{s},\mathtt{u},r_i) \in \sigma(\mathtt{s}))$$
$$\sigma(\mathtt{s}) \rightarrow \mathtt{t}_{mid(\sigma(\mathtt{s}))}$$

With this mapping from session states to types established, we need to make sure that the mapping types have permissions that correspond to the permissions which would be granted to a session in that state. This is achieved by attaching to each type that represents a session state, the set of type aliases that correspond to the roles that are active in the session state being represented.

$$\alpha_i \in type2attr(\mathtt{t}_{mid(\sigma(\mathtt{s}))}) \Leftrightarrow b_i = 1$$

With this permission mapping we have a mapping that grants accesses to resources in TE that correspond to the accesses that would be granted in RBAC. A second type of permissions does also need to be mapped: the permissions to activate or deactivate a role. These permissions need to be mapped to two TE permissions: the permission for the running process to request a type transition, and the permission for that type transition to take place. The TE permissions corresponding to the RBAC role

activations are obtained by enumerating all the valid pairs of session states that belong to the same user, and generating for each such pair the required access vector rules:

$$\forall(\mathtt{s},\mathtt{s}') \text{ s.t.}$$
$$\sigma(\mathtt{s}) = (\mathtt{u}, b_1..b_i..b_{n_r}) \text{ s.t. } \forall\ 1 \le j \le n_r, b_j = 1 \Rightarrow (\mathtt{u},\mathtt{r}_i) \in Ass$$
$$\text{and } \sigma(\mathtt{s}') = (\mathtt{u}, b'_1..b'_i..b'_{n_r}) \text{ s.t. } \forall\ 1 \le j \le n_r, b'_j = 1 \Rightarrow (\mathtt{u},\mathtt{r}_i) \in Ass$$
$$\text{and } hammingDistance(b_1..b_i..b_{n_r}, b'_1..b'_i..b'_{n_r}) = 1$$
$$\text{then}$$
$$a(\mathtt{t}_{mid(\sigma(\mathtt{s}))}, \mathtt{t}_{mid(\sigma(\mathtt{s}))}, \mathtt{process}, \mathtt{setcurrent}) \in \Psi^{TE}$$
$$a(\mathtt{t}_{mid(\sigma(\mathtt{s}'))}, \mathtt{t}_{mid(\sigma(\mathtt{s}'))}, \mathtt{process}, \mathtt{setcurrent}) \in \Psi^{TE}$$
$$a(\mathtt{t}_{mid(\sigma(\mathtt{s}))}, \mathtt{t}_{mid(\sigma(\mathtt{s}'))}, \mathtt{process}, \mathtt{dyntransition}) \in \Psi^{TE}$$
$$a(\mathtt{t}_{mid(\sigma(\mathtt{s}'))}, \mathtt{t}_{mid(\sigma(\mathtt{s}))}, \mathtt{process}, \mathtt{dyntransition}) \in \Psi^{TE}$$

With this mapping established, we proceed to demonstrate that an access will be allowed in the TE configuration image if and only if it would have been allowed in the RBAC configuration source.

We first prove that$\Psi^{\ RBAC} \vdash req(\mathtt{s},\mathtt{ob},\mathtt{op}) \Rightarrow \Psi^{TE} \vdash r(_, \mathtt{t}_{mid(\sigma(s))}, \mathtt{t}_{ob}, _, \mathtt{o}_{op}, _)$ If an access $req(\mathtt{s},\mathtt{ob},\mathtt{op})$ is allowed in the source configuration, then by definition of our mapping, there will be

1. a type $\mathtt{t}_{\mathtt{ob}}$ representing object `ob`
2. an operation $\mathtt{o}_{\mathtt{op}}$ representing operation `op`
3. a type $\mathtt{t}_{mid(\sigma(\mathtt{s}))}$ to represent the state of session `s`
4. at least one role with the necessary permission that is active in session `s`. Let $\mathtt{r}_i$ be that role (i.e. $p(\mathtt{r}_i, \mathtt{t}_{ob}, \mathtt{o}_{op}) \in P$)
5. a type attribute α_i representing role $\mathtt{r}_i$

6. an access vector rule $a(\alpha_i, \mathtt{t}_{ob}, \_, \mathtt{o}_{op})$ granting the corresponding permission to α_i

7. a binding from this type attribute α_i to the type $\mathtt{t}_{mid(\sigma(\mathtt{s}))}$ that represents the session state: $\alpha_i \in type2attr(\mathtt{t}_{mid(\sigma(\mathtt{s}))})$

By application of TE-ACCESS-ATTR1 (see Figure 3.5),$\Psi\ ^{TE} \vdash r(\_, \mathtt{t}_{mid(\sigma(s))}, \mathtt{t}_{ob}, \_, \mathtt{op}, \_)$. Proving the converse is done by observing that, by construction, our mapping from an RBAC configuration to a TE configuration is bijective: each element of the TE configuration exists if and only if there is a corresponding element in the RBAC configuration, and the accesses and domain transitions allowed in TE strictly reflect the accesses and role activations allowed in RBAC. We have therefore exhibited a mapping from RBAC to TE such that the allowed behavior in the TE model is a strict emulation of the allowed behavior in the TE model. □

3.2.3 Mapping TE to RBAC

In this section we argue that a mapping from TE to Core RBAC can not be constructed, by outlining its tentative construction. Similarly to the mapping in the other direction, the following elements have to be mapped: the types, the object classes, their operations, the access control domains and their attributes, the permissions granted to the domains, and the domain transitions. We start by mapping types to roles.

$$\mathtt{t_i} \rightarrow \mathtt{r}_{\mathtt{t_i}}$$

When we map the RBAC permissions to TE access vector rules, there needs to be one permission assignment for each object that is labeled with the target type. This

is because the RBAC standard does not specify a way to group resources, contrary to the seminal work by Baldwin [58].

$$a(\mathtt{t}_i, \mathtt{t}_j, \mathtt{c}, \mathtt{o}) \in \Psi^{TE} \rightarrow \forall \mathtt{ob}_k \in \mathtt{t}_j, p(\mathtt{r}_{t_i}, \mathtt{ob}_k, \mathtt{op}_o) \in P$$

As in the mapping from RBAC to TE, we have not mapped object classes but remark that this extension can be performed easily, as shown in UARBAC [77].

The domain transitions are the part of the mapping that poses problem. The problem is twofold, based on the fact that the semantics of role activation are not amenable to emulating the semantics of domain transitions. First, domain transitions are atomic and the set of permissions available to a domain after the transition can be completely disjoint from the set of permissions available to the domain before the transition. This disjointness is not supported atomically by RBAC, which offer only a role activation (which adds permissions) or a role deactivation (which removes permissions) as atomic primitives. In other words, RBAC is missing a role transition which would replace an active role by another one[6]. Second, TE domain transitions are unidirectional: the permission for a process to transfer from one domain to another is directional. The fact that a process is allowed to transition from a domain to another one does not imply the authorization to transition back. In RBAC, however, domain transitions are all reversible: the user to role assignment determines the set of roles that a user can activate *and* deactivate at will. For these reasons, we consider that Core RBAC is not capable of emulating TE. Core RBAC can be extended, and then probably be able to emulate TE; that is beyond what the standard covers.

3.2.4 Summary

In order to compare RBAC and TE, we have provided a model of Core RBAC using the same formalism that is used in the rest of this chapter to model TE. In

[6]Flask, on the other hand, offers only role transitions and forces the set of active roles to be a singleton.

this formalism, we have constructed a mapping from Core RBAC to TE-core which shows that TE is capable, in theory, of emulating Core RBAC. In practice, though, this mapping would only work for RBAC configurations with a small number of roles. Indeed, for n_u users and n_r roles, our mapping generates up to $n_u \times 2^{n_r}$ domains which are connected by up to $n_u \times 2^{n_r} \times n_r$ transitions. This quickly becomes intractable as the numbers of roles and users grow. This limitation can be avoided altogether by composing TE with RBAC, as we showed in our survey of the related work (see Section 2.3.2). This composition of access control models is how the family of systems derived from Flask offers access control *mechanisms* that are expressive *in practice*.

The mapping of TE-core to Core RBAC can not be achieved because Core RBAC lacks unidirectional role transitions, which are an integral part of TE and a key feature to confine applications. An analogy with the real world should make our point clearer: when a criminal is convicted and jailed, that person can not decide on her own to leave the jail. Neither should a regular process be allowed to change its confinement at will, regardless of the user on behalf of whom the process runs. We are sure that Core RBAC can be extended to support unidirectional role transitions. For instance, the work by Nyanchama and Osborn on MAC atop RBAC [125] showed how Core RBAC (before it was standardized) can be extended to provide acyclic information flow. Once again, we think that model composition is the solution that makes sense in practice.

We have compared TE and RBAC and observed that RBAC does not offer unidirectional domain transitions. Since these unidirectional transitions are essential to the confinement of applications, the remainder of this thesis focuses on TE. In the next section, we proceed to extend TE so that it contains its own administrative model.

3.3 Extending TE to Contain Its Administrative Model

In Section 3.1, we presented the core features of TE and its extensions, as they are embodied in SELinux. We now present how TE can be extended to support recursive policy statements. That is, we present an extension of the TE model (as it was modeled above) in which a TE policy can contain statements that regulate modifications that can be made to the same policy in which they are contained. This extension is exposed in two steps. First, we explain the necessary reification of policy statements (see Section 3.3.1). New objects classes and their operations are added to the model and the evaluation semantics are modified accordingly. We show that this first step is a necessary but not sufficient extension for TE to contain its own administrative model with support for fine-grained delegation of administrative permissions. Consequently, we then expose an additional extension, with consists in pattern-matching the policy statements in order to address this limitation (see Section 3.3.2).

3.3.1 Recursive Policy Statements

We now describe how the TE model can be extended to support simple recursive policy statements. These recursive policy statements express under which conditions the policy can be modified, and hence support the definition of an administrative policy.

Syntax

At the syntax level, supporting recursive policy statements requires adding two classes of objects to represent the policy statements we presented in the TE model (see Section 3.1): the `av_rule` class to represent access vector rules and the `tr_rule` class to represent transition rules.

These classes support two operations that enable administration of the security policy: `insert` for the creation of a policy statement, and `remove` for the deletion of a policy statement.

Semantics

This simple recursive extension of TE does not change the structure of access requests, access vector rules, or type transition rules, so the semantics of the evaluation of an access request according to the policy of the system remain unchanged, except for the case of access requests on access vector rules. Indeed, allowing to delete or create an access vector rule has the side effect of modifying the policy, as shown in the rules EVAL-INSERT and EVAL-REMOVE (Figure 3.8). Supporting this extension to the semantics requires a modification of the implementation of the access control mechanisms so that the two newly introduced `av_rule` and `tr_rule` object classes can be represented and manipulated. We discuss this extension at the end of this chapter (see Section 3.4).

With the addition of the `av_rule` and `tr_rule` object classes, it is now possible to write access vector rules like the following.

```
allow webmaster_t webapp_t:av_rule { insert remove };
```

This rule specifies that a webmaster can create and delete access vector rules that have the type `webapp_t`.

Limits

The recursive extension that we explained does actually suffer from a serious limitation: there is no clear intuition on how to represent the security relevance of a given `av_rule` or `tr_rule` object through a sinlge type. For instance, there are at least two different ways to type `av_rule` objects: use the source type or use the target type. As we explain below, none of these alternatives is satisfactory.

TE-recursive

Syntax

$$
\begin{array}{rl}
\texttt{access\_vector\_rule} := & a(\texttt{s},\texttt{t},\texttt{c},\texttt{o}) \\
\texttt{type\_transition\_rule} := & n(\texttt{s},\texttt{t},\texttt{c},\texttt{s}') \\
\texttt{access\_request} := & r(\texttt{s},\texttt{t},\texttt{c},\texttt{o},\texttt{i}) \\
\texttt{R} := & \texttt{rR} \mid \varnothing \\
\Psi := & \{\texttt{a}\} \cup \{\texttt{n}\}
\end{array}
$$

Metavariables

$\texttt{s},\texttt{t}$	*types*
$\texttt{c}$	*class*
$\texttt{o}$	*operation*
$\texttt{i}$	*instance*
$\texttt{r}$	`access_request`
$\texttt{a}$	`access_vector_rule`
$\texttt{n}$	`type_transition_rule`
Ψ	*policy*

Semantics

$$\frac{\begin{array}{c}\texttt{r} = r(\texttt{s},\texttt{t},\texttt{c},\texttt{o},\texttt{i}) \\ \Psi \vdash \texttt{r} \\ c \notin \{\texttt{av\_rule},\texttt{tr\_rule}\} \\ o \notin \{\texttt{execute},\texttt{create},\texttt{setcurrent}\}\end{array}}{\Psi,\texttt{rR} \to \Psi,\texttt{R}} \text{ (Eval-base)}$$

$$\frac{\begin{array}{c}\texttt{r} = r(\texttt{s},\texttt{t},\texttt{c},\texttt{o},\texttt{i}) \\ \Psi \vdash \texttt{r} \\ c \in \{\texttt{av\_rule},\texttt{tr\_rule}\} \\ o = \texttt{insert}\end{array}}{\Psi,\texttt{rR} \to \Psi \cup \{i\},\texttt{R}} \text{ (Eval-insert)}$$

$$\frac{\begin{array}{c}\texttt{r} = r(\texttt{s},\texttt{t},\texttt{c},\texttt{o},\texttt{i}) \\ \Psi \vdash \texttt{r} \\ c \in \{\texttt{av\_rule},\texttt{tr\_rule}\} \\ o = \texttt{remove}\end{array}}{\Psi,\texttt{rR} \to \Psi \setminus \{i\},\texttt{R}} \text{ (Eval-remove)}$$

$$\frac{\begin{array}{c}\texttt{r} = r(\texttt{s},\texttt{t},\texttt{c},\texttt{o},\texttt{i}) \\ a(\texttt{s}',\texttt{t}',\texttt{c}',\texttt{o}') \in \Psi \\ \mathit{match\_type}(\texttt{s},\texttt{s}') \\ \mathit{match\_type}(\texttt{t},\texttt{t}') \\ \mathit{match\_class}(\texttt{c},\texttt{c}') \\ \mathit{match\_op}(\texttt{o},\texttt{o}')\end{array}}{\Psi \vdash \texttt{r}} \text{ (TE-base)}$$

$$\begin{array}{c}
\mathit{match_type}(\texttt{t},\texttt{t}) \text{ (Match-T-eq)} \\
\mathit{match_type}(\texttt{t},_) \text{ (Match-T-wild)} \\
\mathit{match_type}(\texttt{c},\texttt{c}) \text{ (Match-C-eq)} \\
\mathit{match_type}(\texttt{c},_) \text{ (Match-C-wild)} \\
\mathit{match_type}(\texttt{o},\texttt{o}) \text{ (Match-O-eq)} \\
\mathit{match_type}(\texttt{o},_) \text{ (Match-O-wild)}
\end{array}$$

Figure 3.8.: Semantics for the recursive TE model. Please note that the base rule for the evaluation of access requests is now (Eval-base). It excludes the cases of `execute` and `create` operations. These operations are still allowed according to the semantics defined in Figures 3.2 and 3.3. Additionaly, the Eval-base rule now excludes operations on the `av_rule` and `tr_rule` classes of objects, since the `insert` and `remove` operations on these classes have the side effect of modifying the policy. These modifications are reflected in the Eval-insert and Eval-remove rules.

If we use the source type, meaning that an `av_rule` object will have the same type as the source type it contains, then the previous example (see Section 3.3.1) can be interpreted as meaning "the webmaster has administrative rights to add and remove permissions to the web application domain". However, such a statement presents a serious limitation: it does not specify *which* permissions the webmaster can add or remove to the web application domain. A malicious webmaster could use this lack of specification to get full privilege access on the machine.

If we use the target type, meaning that an `av_rule` object will have the same type as the target type it contains, then the same example can be interpreted as meaning "the webmaster has administrative rights to add or remove permissions to perform operations on objects of type `webapp_t`." However, yet again, such a statement presents a serious limitation: it does not specify *who* the webmaster can grant these permissions to. Moreover, under these semantics it is hard to identify who is granted administrative rights on the web application domain or, for that matter, who is given administrative rights to any security domain.

A similar limitation of Type Enforcement was also pointed out by Spencer et al. [9] in the context of filesystem access control, when considering the relabeling of files (the operation that changes the security type attached to a file). In this case, the decision depends on more than two types: three types have to be considered, as the decision depends on the subject requesting the relabeling, the current type of the file, and the desired type of the file. Fortunately in this case, the decision rule can be properly encoded by three separate permissions: a permission for the subject to relabel from the original type, a permission for the subject to relabel to the desired type, and a permission for the original type of the file to be changed to the desired type. This solution, however, can not be adapted to our problem.

Another alternative, which would use types to enumerate the possible combinations of source types and target types would not be practical as it would introduce a proliferation of types. While this solution could work in theory, it would not work in practice (similarly to the emulation of RBAC with TE which we showed in Sec-

tion 3.2.2). We have therefore extended the simple recursive model with support for recursive pattern matching on rules, which we present below.

3.3.2 Pattern Matching Policy Statements

To remedy the limitations that were presented above, we have extended the recursive TE model with support for pattern matching the contents of permission and transition objects (defined below). This feature enables the definition of fine-grained administrative policies.

Definition 3.3.1 (Permission object) *An object which is an instance of the* `av_rule` *class is called a permission object, for brevity, as it does represent a permission in the policy.*

Definition 3.3.2 (Transition object) *An object which is an instance of the* `tr_rule` *class is called a transition object, for brevity, as it does represent a type transition in the policy.*

Syntax

Definition 3.3.3 (TE-Pattern access vector rule) *A TE-Pattern access vector rule has the same overall structure as an access vector rule from the core TE model (see Section 3.1.1): it is a 4-tuple of the form* $(\mathtt{s}, \mathtt{t}, \mathtt{p}, \mathtt{o})$ *where* `s`*,* `t`*, and* `o` *can take the same values that they would in an access vector rule. The difference is with the* `p` *field which can contain either an object class specification (as in a regular access vector rule), or a policy statement pattern, to constrain the content of permission or transition objects that can be manipulated.*

Definition 3.3.4 (Permission pattern) *A permission pattern is a recursive structure with the same fields as a TE-Pattern access vector rule. In the base case, the* `p` *field of the pattern contains a class specification. In the recursive case, the* `p` *field*

contains another permission pattern. Any of the fields of a permission pattern can be wildcarded.

Definition 3.3.5 (Transition pattern) *A transition pattern is a structure with the same fields as a TE type transition rule. Any of the fields of a transition pattern can be wildcarded.*

Semantics

The semantics are summarized in Figure 3.9.

Semantics 3.3.2.1 (Authorizing accesses) *As previously, an access request is authorized if and only if the policy contains an access vector rule that matches the request; the main extension to the semantics is the addition of the rules that support the recursive pattern-matching of permission objects:* TE-PATTERN, PERM-PATT-NESTED, *and* PATTERN-NESTED-REC *(fig. 3.9).*

With this extended recursive model, it is now possible to specify precisely that the webmaster can manage all permissions that grant access to objects of type `webapp_t`*:*

```
allow webmaster_t _:av_rule(_,webapp_t,_,_) { create delete };
```

Please note that in the above rule we omitted to specify the type of the permission objects that the webmaster can manipulate; such a specification is not useful, as pointed out in Section 3.3.1.

Semantics 3.3.2.2 (Automatic labeling of new objects) *The semantics for type transition rules remain unchanged.*

Proof of termination

We now proceed to demonstrate that the evaluation of an access request in the extended recursive model does still terminate in finite time, despite its recursive

TE-recursive-pattern

Syntax

access_vector_rule :=	$a(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{o})$ $a(\mathtt{s},\mathtt{t},\mathtt{p},\mathtt{o})$	
type_transition_rule :=	$n(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{s}')$	
access_vector_pattern :=	$avp(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{o})$ $avp(\mathtt{s},\mathtt{t},\mathtt{p},\mathtt{o})$	
type_trans_pattern :=	$ttp(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{s}')$	
rule_pattern :=	access_vector_pattern	
	type_trans_pattern	
access_request :=	$r(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{o},\mathtt{i})$	
R :=	rR $\mid\varnothing$	
Ψ :=	$\{\mathtt{a}\}\cup\{\mathtt{n}\}$	

Metavariables

s, t	*types*
c	*class*
o	*operation*
i	*instance*
r	access_request
a	access_vector_rule
n	type_transition_rule
p	rule_pattern
Ψ	*policy*

Semantics

$$\frac{\mathtt{r}=r(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{o},\mathtt{i}) \quad \Psi\vdash\mathtt{r} \quad c\notin\{\mathtt{av\_rule},\mathtt{tr\_rule}\}}{\Psi,\mathtt{rR}\to\Psi,\mathtt{R}}\ (\textsc{Eval-base})$$

$$\frac{\mathtt{r}=r(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{o},\mathtt{i}) \quad \Psi\vdash\mathtt{r} \quad c\in\{\mathtt{av\_rule},\mathtt{tr\_rule}\} \quad o=\mathtt{insert}}{\Psi,\mathtt{rR}\to\Psi\cup\{i\},\mathtt{R}}\ (\textsc{Eval-insert})$$

$$\frac{\mathtt{r}=r(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{o},\mathtt{i}) \quad \Psi\vdash\mathtt{r} \quad c\in\{\mathtt{av\_rule},\mathtt{tr\_rule}\} \quad o=\mathtt{remove}}{\Psi,\mathtt{rR}\to\Psi\setminus\{i\},\mathtt{R}}\ (\textsc{Eval-remove})$$

$$\frac{\mathtt{r}=r(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{o},\mathtt{i}) \quad c\notin\{\mathtt{av\_rule},\mathtt{tr\_rule}\} \quad a(\mathtt{s}',\mathtt{t}',\mathtt{c}',\mathtt{o}')\in\Psi \quad is\_in(avp(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{o}),avp(\mathtt{s}',\mathtt{t}',\mathtt{c}',\mathtt{o}'))}{\Psi\vdash\mathtt{r}}\ (\textsc{TE-base})$$

$$\frac{\mathtt{r}=r(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{o},\mathtt{i}) \quad c\in\{\mathtt{av\_rule},\mathtt{tr\_rule}\} \quad \exists\mathtt{p}\ s.t.\ (a(\mathtt{s},\mathtt{t},\mathtt{p},\mathtt{o})\in\Psi \wedge match(\mathtt{i},\mathtt{p}))}{\Psi\vdash\mathtt{r}}\ (\textsc{TE-pattern})$$

$$\frac{\mathtt{i}=n(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{s}') \quad is\_in(ttp(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{o}),\mathtt{p})}{match(\mathtt{i},\mathtt{p})}\ (\textsc{Patt-ttr-base})$$

$$\frac{\mathtt{i}=a(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{o}) \quad is\_in(avp(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{o}),\mathtt{p})}{match(\mathtt{i},\mathtt{p})}\ (\textsc{Patt-avr-base})$$

$$\frac{\mathtt{i}=a(\mathtt{s},\mathtt{t},\mathtt{p}',\mathtt{o}) \quad is\_in(avp(\mathtt{s},\mathtt{t},\mathtt{p}',\mathtt{o}),\mathtt{p})}{match(\mathtt{i},\mathtt{p})}\ (\textsc{Patt-avr-nested})$$

Figure 3.9.: Semantics for the extended reflexive TE model (1/2)

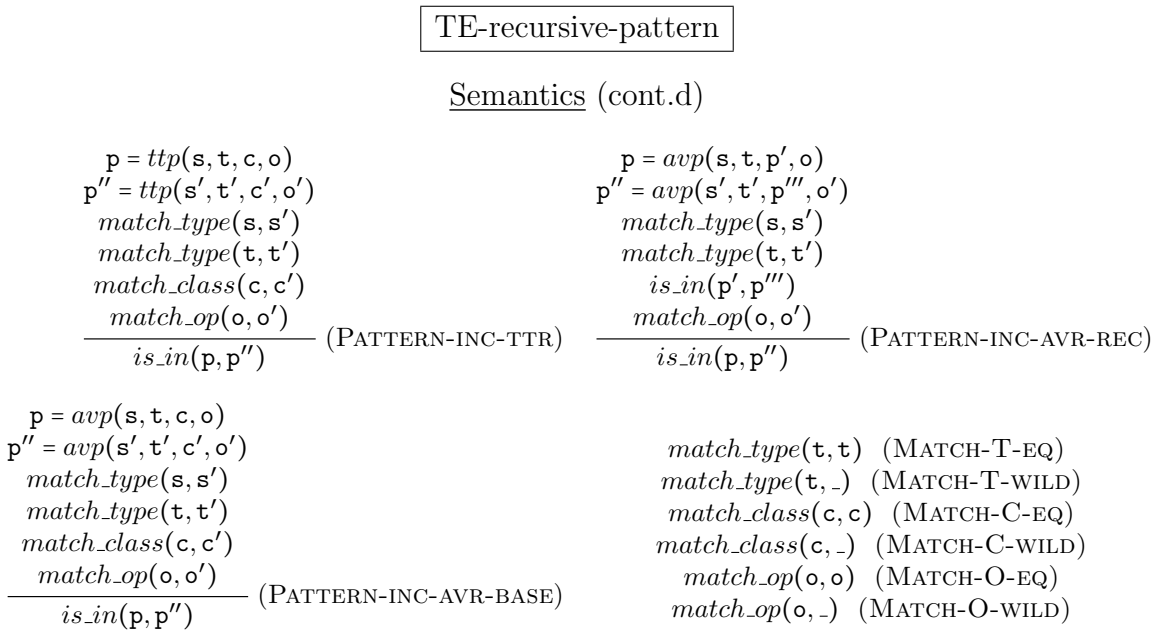


Figure 3.9.: Semantics for the extended reflexive TE model (2/2)

nature. Showing this property is important, as otherwise the administrative extension would not guarantee that modifications to the policy can be made in a timely fashion. Moreover, it would potentially be possible to use these extensions for denial of service attacks, by loading the system with non-terminating policy administration jobs.

Definition 3.3.6 (Depth of a permission pattern) *A permission pattern where the* `p` *field contains a class specification is said to have depth 0; otherwise the depth of a pattern is equal to the depth of the pattern it contains, plus one.*

Lemma 3.3.2.1 *The nesting of patterns can not form a cycle.*

Proof [Proof of Lemma 3.3.2.1] Since there is no way to name patterns, there is no way to reference patterns, and hence no way to form a cycle with the nesting of patterns. ■

Corollary 3.3.2.2 (Corollary of Lemma 3.3.2.1) *The depth of a pattern is finite.*

Theorem 3.3.2.3 *The evaluation of an access request terminates in a finite number of evaluation steps.*

Proof [Proof of Theorem 3.3.2.3]

Let $\mathtt{r} = r(\mathtt{s}, \mathtt{t}, \mathtt{c}, \mathtt{o}, i)$ be the access request being evaluated (granted or denied). Let `a` be the access vector rule against which the access request is being evaluated.

There are two cases:

1. $\mathtt{a} = a(\mathtt{s}', \mathtt{t}', \mathtt{c}', \mathtt{o}')$

   In this case, the evaluation proceeds through the rule TE-BASE, which triggers one evaluation of the rule PATTERN-INC-AVR-BASE, which terminates after matching the elements of `a` and `r` one by one.

2. $\mathtt{a} = a(\mathtt{s}', \mathtt{t}', \mathtt{p}', \mathtt{o}')$

   We prove this case by induction on the depth of the pattern contained in an access vector rule.

- Base case: $\mathtt{p}'$ is a pattern of depth 0.

  This means that we have either $\mathtt{p} = p(\mathtt{s}'', \mathtt{t}'', \mathtt{c}'', \mathtt{o}'')$ or $\mathtt{p} = p(\mathtt{s}'', \mathtt{t}'', \_, \mathtt{o}'')$ Either cases will be evaluated through the rules TE-Pattern, Patt-avr-nested, and if $\mathtt{i}$ is a permission, then Pattern-inc-avr-base, which terminates after matching the elements of $\mathtt{i}$ and $\mathtt{p}$ one by one.

- Induction hypothesis: if $\mathtt{p}'$ is a pattern of depth $(n-1)$ $(n \geq 1)$, the evaluation terminates.

- Induction: if $\mathtt{p}'$ is a pattern of depth n, it contains by definition a pattern of depth $(n-1)$. The evaluation of the access request will be reduced, after application of the rules TE-Pattern, Patt-avr-nested and Pattern-inc-avr-rec, to the same evaluation that would be performed for a pattern of depth $(n-1)$, which terminates.

■

The result of this theorem is strengthened by the fact that, in practice, we do not expect a policy to contain patterns of a depth exceeding two: depth zero corresponds to base permissions, depth two to administrative permissions (which constitute the administrative policy), and depth three to administrative permissions on the administrative policy. Beyond depth three, we lose the security intuition on the meaning of permissions.

3.3.3 Administrative Templates

With the previous extension, it is now possible to precisely define which administrative permissions are granted to which user of the system. This precision, however, comes at a price: many administrative rules are necessary to grant administrative privileges. For instance, 451 type enforcement access vectors are used in the definition of the domain of the webalizer application (a log analysis application). The same

number of administrative rules is required to grant a user the administrative right to create a similar domain.

It is a direct consequence of the fine granularity of the administrative permissions that, for each permission that is to be granted, there needs to be an administrative permission to authorize the granting of this permission. Having an administrative policy whose size is comparable to the size of the underlying policy is not an attractive perspective, specially in the case of TE, where the underlying policy can be large.

Our approach to mitigating this problem consists in using parts of the policy as templates for other parts of the policy, in a fashion similar to the use of bounded types (see Section 3.1.2). Bounded types are used to define the access boundary of a type based on the access boundary of another type. We propose to use the access boundary of a type to define administrative boundaries. For instance, consider a webmaster that is allowed to deploy and configure log analysis tools. One could say that the webmaster is allowed to grant, to a specific domain, all the permissions granted to the webalizer domain. By doing so, all the efforts put into the definition of the webalizer domain can be reused to define similar boundaries for similar applications.

We have introduced two kinds of templates, for which we provide definitions and semantics below. The first kind of template, based on all the permissions of a domain, is an administrative domain template. The second kind of template, based on the permissions that a domain has on a resource of a given (TE) type, is called an administrative resource template.

Using Administrative Templates

Definition 3.3.7 (Administrative domain template) *An administrative domain template is a 3-tuple* $(\mathtt{s}, \mathtt{d}_{\mathtt{ref}}, \mathtt{d}_{\mathtt{target}})$*, where* $\mathtt{d}_{\mathtt{ref}}$ *is the domain used as a reference for the template,* $\mathtt{d}_{\mathtt{target}}$ *is the target domain to which the template can be applied, and* $\mathtt{s}$ *is the subject that can apply part of the template.*

Semantics 3.3.3.1 (Administrative domain template) *Administrative templates offer an additional means of granting administrative access to the policy.*

Consequently, the semantics of the extended reflexive model (see Figure 3.9) are augmented, as illustrated in Figure 3.10. These semantics indicate that a subject can can add permission to a domain $\texttt{d}_{\texttt{target}}$ *if:*

- *There is a domain* $\texttt{d}_{\texttt{ref}}$ *that possesses this permission*
- *There is an administrative template that indicates that* s *can grant permissions from domain* $\texttt{d}_{\texttt{ref}}$ *to domain* $\texttt{d}_{\texttt{target}}$.

The granting of administrative permissions according to administrative domain templates is embodied by the rule ADMIN-DTMPL *(see Figure 3.10).*

We use the following concrete syntax to represent an administrative domain template. This syntax is deliberately chosen to be similar to the definition of type boundaries, as the concepts are similar.

```
admin_domain_template admin referencedomain targetdomain1\
                                          [targetdomain2 ...]
```

Definition 3.3.8 (Administrative resource template) *An administrative resource template is a 5-tuple* $(\texttt{s}, \texttt{d}_{\texttt{ref}}, \texttt{t}_{\texttt{ref}}, \texttt{d}_{\texttt{target}}, \texttt{t}_{\texttt{target}})$*, where* $\texttt{d}_{\texttt{ref}}$ *and* $\texttt{t}_{\texttt{ref}}$ *are the reference domain and target type for the template, and* $\texttt{d}_{\texttt{target}}$ *and* $\texttt{t}_{\texttt{target}}$ *are the domain and target type to which the template is applied, and* s *is the subject to which the template administrative permissions are granted.*

Semantics 3.3.3.2 (Administrative resource template) *Administrative resource templates offer an additional means of granting administrative access to the policy. Consequently, the semantics of the extended reflexive model (see Figure 3.9) are augmented, as illustrated in Figure 3.10. These semantics indicate that a subject can can add a permission to a domain* $\texttt{d}_{\texttt{target}}$ *if:*

- *There is a domain* $\texttt{d}_{\texttt{ref}}$ *that possesses this permission on a type* $\texttt{t}_{\texttt{ref}}$
- *There is an administrative resource template that indicates that* s *can grant permissions that domain* $\texttt{d}_{\texttt{ref}}$ *has on type* $\texttt{t}_{\texttt{ref}}$ *to domain* $\texttt{d}_{\texttt{target}}$ *on type* $\texttt{d}_{\texttt{target}}$.

The granting of administrative permissions according to administrative domain templates is embodied by the rule ADMIN-RTMPL *(see Figure 3.10).*

We use the following concrete syntax to represent an administrative resource template.

```
admin_resource_template admin referencedomain referencetarget \
                                        targetdomain targettarget
```

Administering Administrative Templates

To support the formulation of precise administrative rules on the insertion and removal of administrative templates, the model needs to be extended so that administrative templates are manipulable objects, whose manipulation is regulated by administrative rules. This extension is similar to the extension that handles `tr_rule` objects; we present it below.

Definition 3.3.9 (Administrative template object) *Administrative domain (resp. resource) template rules are represented as instances of the* `admin_tmpl` *(resp.* `admin_rtmpl`*) class, and are called administrative domain (resp. resource) template objects when they are being manipulated by administrative operations.*

Semantics 3.3.3.3 (Administering administrative templates) *Similarly to* `av_rule` *and* `tr_rule` *objects,* `admin_dtmpl` *(resp.* `admin_rtmpl`*) objects support two operations:* `insert` *and* `remove`*. Similarly to* `tr_rule` *objects, the allowed manipulations of these objects are declared with rules that can contain simple (non-recursive) patterns that specify their allowed content. The detailed semantics are provided in Figure3.11.*

TE-admin-template

Syntax

access_vector_rule :=	$a(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{o})$ $a(\mathtt{s},\mathtt{t},\mathtt{p},\mathtt{o})$
type_transition_rule :=	$n(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{s}')$
admin_domain_rule :=	$adr(\mathtt{s},\mathtt{t},\mathtt{t}')$
admin_resource_rule :=	$arr(\mathtt{s},\mathtt{d},\mathtt{t},\mathtt{d}',\mathtt{t}')$
access_vector_pattern :=	$avp(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{o})$ $avp(\mathtt{s},\mathtt{t},\mathtt{p},\mathtt{o})$
type_trans_pattern :=	$ttp(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{s}')$
admin_domain_pattern :=	$adp(\mathtt{s},\mathtt{t},\mathtt{t}')$
admin_resource_pattern :=	$arp(\mathtt{s},\mathtt{d},\mathtt{t},\mathtt{d}',\mathtt{t}')$
rule_pattern :=	access_vector_pattern
	type_trans_pattern
	admin_domain_rule
	admin_resource_rule
access_request :=	$r(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{o},\mathtt{i})$
R :=	rR $\mid\varnothing$
Ψ :=	$\{\mathtt{a}\}\cup\{\mathtt{n}\}\cup\{\mathtt{b}\}\cup\{\mathtt{l}\}$

Metavariables

s, t, d	*types*
c	*class*
o	*operation*
i	*instance*
r	access_request
a	access_vector_rule
n	type_transition_rule
b	admin_domain_rule
l	admin_resource_rule
p	rule_pattern
Ψ	*policy*

Semantics

$$\frac{\begin{array}{c}\mathtt{r}=r(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{o},\mathtt{i})\\ \mathtt{o}\in\{\mathtt{insert},\mathtt{remove}\}\\ \mathtt{i}=a(\mathtt{d}',\mathtt{t}',\mathtt{c}',\mathtt{o}')\\ adr(\mathtt{s},\mathtt{d},\mathtt{d}')\in\Psi\\ a(\mathtt{d},\mathtt{t}',\mathtt{c}',\mathtt{o}')\in\Psi\end{array}}{\Psi\vdash\mathtt{r}}\qquad\text{(Admin-dtmpl)}$$

$$\frac{\begin{array}{c}\mathtt{r}=r(\mathtt{s},\mathtt{t},\mathtt{c},\mathtt{o},\mathtt{i})\\ \mathtt{o}\in\{\mathtt{insert},\mathtt{remove}\}\\ \mathtt{i}=a(\mathtt{d}',\mathtt{t}',\mathtt{c}',\mathtt{o}')\\ arr(\mathtt{s},\mathtt{d}_{\mathtt{ref}},\mathtt{t}_{\mathtt{ref}},\mathtt{d}_{\mathtt{target}},\mathtt{t}_{\mathtt{target}})\in\Psi\\ a(\mathtt{d}_{\mathtt{ref}},\mathtt{t}_{\mathtt{ref}},\mathtt{c}',\mathtt{o}')\in\Psi\end{array}}{\Psi\vdash\mathtt{r}}\qquad\text{(Admin-rtmpl)}$$

Figure 3.10.: Semantics for administrative templates: additional rule that allow administrative operations if there is an administrative template based on which the administrative operation can be allowed.

TE-admin-template-admin

Syntax

$$
\begin{array}{rcl}
\texttt{access\_vector\_rule} & ::= & a(\texttt{s},\texttt{t},\texttt{c},\texttt{o}) \;\; a(\texttt{s},\texttt{t},\texttt{p},\texttt{o}) \\
\texttt{permission\_pattern} & ::= & p(\texttt{s},\texttt{t},\texttt{c},\texttt{o}) \;\; p(\texttt{s},\texttt{t},\texttt{p},\texttt{o}) \\
\texttt{access\_request} & ::= & r(\texttt{s},\texttt{t},\texttt{c},\texttt{o},\texttt{i}) \\
\texttt{R} & ::= & \texttt{rR} \mid \varnothing \\
\Psi & ::= & \{\texttt{a}\}
\end{array}
$$

Metavariables

$\texttt{s},\texttt{t},\texttt{d}$	*types*
$\texttt{c}$	*class*
$\texttt{o}$	*operation*
$\texttt{i}$	*instance*
$\texttt{a}$	access_vector_rule
$\texttt{p},\texttt{p}',\texttt{p}''$	permission_pattern
$\texttt{r}$	access_request
Ψ	*policy*

Semantics

$$\frac{\texttt{i} = adr(\texttt{s},\texttt{t},\texttt{t}') \quad is\_in(adp(\texttt{s},\texttt{t},\texttt{t}'),\texttt{p})}{match(\texttt{i},\texttt{p})}\ (\textsc{Dpatt-bound-base})$$

$$\frac{\texttt{i} = arr(\texttt{s},\texttt{d},\texttt{t},\texttt{d}',\texttt{t}') \quad is\_in(arp(\texttt{s},\texttt{d},\texttt{t},\texttt{d}',\texttt{t}'),\texttt{p})}{match(\texttt{i},\texttt{p})}\ (\textsc{Rpatt-bound-base})$$

$$\frac{\begin{array}{c}\texttt{p} = arp(\texttt{s},\texttt{d},\texttt{t},\texttt{d}',\texttt{t}') \quad \texttt{p}'' = arp(\texttt{s}',\texttt{d}'',\texttt{t}'',\texttt{d}''',\texttt{t}''') \\ match\_type(\texttt{s},\texttt{s}') \quad match\_type(\texttt{d},\texttt{d}'') \\ match\_class(\texttt{d}',\texttt{d}''')match\_type(\texttt{t},\texttt{t}'') \quad match\_class(\texttt{t}',\texttt{t}''')\end{array}}{is\_in(\texttt{p},\texttt{p}'')}\ (\textsc{Rpattern-inc-bound})$$

$$\frac{\begin{array}{c}\texttt{p} = adp(\texttt{s},\texttt{t},\texttt{t}') \quad \texttt{p}'' = adp(\texttt{s}',\texttt{t}'',\texttt{c}''') \\ match\_type(\texttt{s},\texttt{s}') \quad match\_type(\texttt{t},\texttt{t}'') \quad match\_class(\texttt{t}',\texttt{t}''')\end{array}}{is\_in(\texttt{p},\texttt{p}'')}\ (\textsc{Dpattern-inc-bound})$$

Figure 3.11.: Semantics for the administration of administrative templates. While the semantics of Figure 3.10 cover the granting of permissions based on administrative templates, these semantics cover the how administrative templates themselves are manipulated. The above semantics are both a generalization of Figure 3.10 and an extension of the semantics from Figure 3.9 (where we introduced the pattern-matching extension).

Proof of Termination

The administrative templates are supported by the introductions of six rules. The Admin-dtmpl, Dpattern-inc-bound, Dpatt-bound-base, Admin-rtmpl, Rpattern-inc-bound, and Rpatt-bound-base rules introduce six new base cases. These base cases do not change the termination of the recursive evaluation of permissions; they just contribute additional base cases to the evaluation. Consequently, the results of the proof of termination still hold.

3.3.4 Summary

In this section, we have presented an administrative model for TE, which is constructed as an extension of TE. The construction has proceeded in two steps. First we have reified TE policy elements, exposing them as objects that are visible at the TE level of the model, so that their manipulation can be subjected to TE access controls. We have shown that this simple extension has obvious limitations. Then, we have shown how these limitations can be addressed by introducing recursive pattern matching on the policy constructs. This pattern matching supports the precise expression of fine-grained administrative permissions. Furthermore, it supports an administrative policy on the administrative policy itself (and so on), hence avoiding a fixed administrative policy on the administrative policy. Finally, we have introduced administrative templates to support the factorization of the administrative policy, by using existing permissions on a given domain as blueprints for administrative permissions on another domain. For both the recursive pattern matching and the administrative template, we have shown that the evaluation of access control decisions always terminates in a finite number of steps.

3.4 Implementation

We have integrated the administrative extensions to TE (presented above) in a prototype on SELinux. In this section, we describe this integration. It has involved deciding on the interface through which to expose the reified elements of the TE policy, so that they can be manipulated. We have chosen to use a virtual filesystem, which we present in Section3.4.1. Another important aspect of the implementation is integrating the implementation of this new interface with the rest of the system in a way that preserves the security guarantees of the system. We present this integration in Section 3.4.2, where we argue that this integration constitues an extension of the trusted computing (TCB) base of the system, in the form of a TCB subsystem. As such, we show that this integration does not weaken the security guarantees of the system, per se.

3.4.1 Interface

We have chosen to expose the policy in terms of a virtual filesystem. That is, the elements of the policy are exposed as virtual files and directories, whose manipulations are regulated by the administrative policy. The administrative policy itself is also exposed inside the same virtual filesystem.

There are several reasons for our choice of using the filesystem interface to expose the policy. The UNIX filesystem API is well understood and supported by many utilities. As a result, standard command-line utilities can be used to interact with our interface (e.g. `ls` for listing policy elements). Being able to re-use file manipulation command line utilities, which have been stable for years, to administer the policy is a big win from an assurance perspective: no extra tool has to be developed to interact with the policy through the filesystem interface. Moreover, having the interface to the policy be textual eases the development and debugging, since the same file manipulation utilities can also be relied on during the development of the interface to test and diagnose the interface. This increases assurance that the system will behave according to its specification.

Yet another advantage of using a filesystem interface is that remote administration can be enabled by exporting the virtual filesystem over a network file system. The filesystem can then be remotely mounted at the administration point. Using a network filesystem for remote administration avoids the design, development, and debug of a protocol and its accompanying libraries and daemons. This is again an advantage from an assurance perspective. One important requirement that is placed on the network filesystem is that it preserves the type labels so that only authorized domains can access the administrative filesystem. This will be addressed by an upcoming extension to NFSv4. (There is currently an IETF draft and a Linux implementation of labeled extensions for NFSv4[7], which addresses this issue of transporting security labels over NFSv4).

Table 3.1: Filesystem layout, and mapping from policy modifications to filesystem operations

Policy modification	FS path	FS operation
add access vector rule *avr*	/avr/	`creat(`*avr*, *mode*`)`
remove access vector rule *avr*	/avr/	`unlink(`*avr*`)`
add type transition rule *ttr*	/ttr/	`creat(`*ttr*, *mode*`)`
remove type transition rule *ttr*	/ttr/	`unlink(`*ttr*`)`
add administrative rule *meta*	/meta/	`creat(`*meta*, *mode*`)`
remove administrative rule *meta*	/meta/	`unlink(`*meta*`)`
add conditional a.v.r. *avr* with guard *guard*	/cond/*guard*/	`creat(`*avr*, *mode*`)`
remove conditional a.v.r. *avr* with guard *guard*	/cond/*guard*/	`unlink(`*avr*`)`
add type *name*	/type/	`creat(`*name*, *mode*`)`
remove type *name*	/type/	`unlink(`*name*`)`
add attribute *name*	/attr/	`mkdir(`*name*, *mode*`)`
remove attribute *name*	/attr/	`remove(`*name*`)`
attach attribute *name1* to type *name2*	/attr/*name1*/	`creat(`*name2*, *mode*`)`
detach attribute *name1* from type *name2*	/attr/*name1*/	`unlink(`*name2*`)`
create type *name1* with attribute *name2*	/attr/*name2*/	`creat(`*name1*, *mode*`)`
remove type *name1* with attribute *name2*	/type/	`unlink(`*name1*`)`

[7] `http://www.ietf.org/internet-drafts/draft-quigley-nfsv4-sec-label-00.txt`

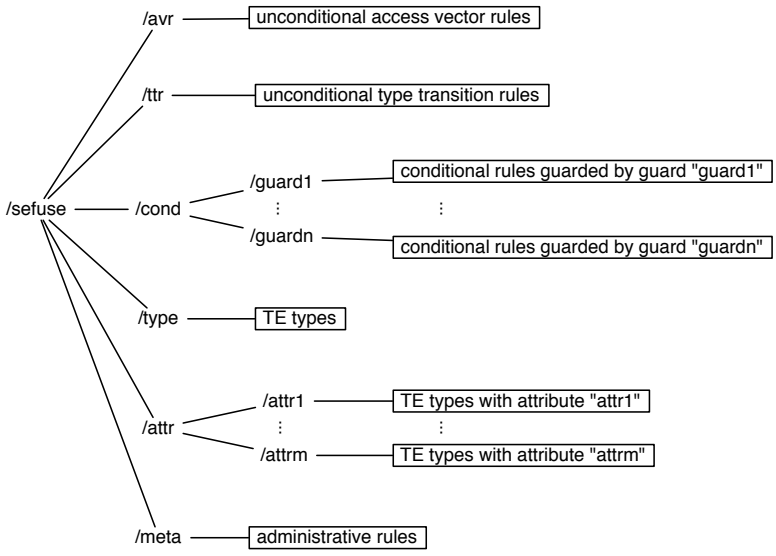


Figure 3.12.: Layout of the virtual filesystem

We now explain the mapping of operations from the administrative model to virtual filesystem interface. Paraphrasing the discussion of the Plan 9 filesystem [126], *object-oriented readers may approach the rest of this [explanation] as a study in how to make objects look like files.* The mapping from policy modifications to filesystem operations, together with the layout of the filesystem, are summarized in Table 3.1; the arborescence of the filesystem is represented in Figure 3.12.

We have chosen to expose each policy statement as either a file or a directory. The rationale for that choice is easier to explain by considering the alternative: any grouping of policy statements within a virtual file would force a stateful implementation of the interface.

Indeed, if several statements are grouped *within* the same virtual file, then adding or removing a statement from the group, or modifying any of these statements, requires to `open()` the file first and then perform the modification. From an implementation perspective, this means that the interface is stateful: an open file descriptor has to be maintained for the whole duration of the edit. There are many reasons to avoid such statefulness. Our main motivation was to keep the implementation simple so that it can be inspected easily. Another reason was the desire to support concurrent edits of the policy. In that case, any un-necessary increase of the granularity of edits must be avoided, hence our decision to expose each policy statement as a virtual file.

As mentioned earlier, the policy and the administrative policy can both be accessed and edited using the same interface. We feel it is important that, in the same way that the administrative model is recursive, and therefore supports an arbitrary stacking of administrative policies, that the interface to the administrative policy also supports the edition of administrative rules of any depth.

3.4.2 System Integration

The virtual filesystem is supported by a userspace server, which we named `sefuse`, as it uses FUSE [127] to communicate with the kernel when handling the virtual

filesystem operations. FUSE is a kernel extension included in the mainstream Linux kernel since the kernel version 2.6.14. This extension allows filesystems to be implemented in userspace by relaying filesystem calls out of the kernel, to userspace daemons.

Our prototype is implemented with support for the SELinux binary policy format version 24. The binary policy is loaded by `sefuse` when it is started. Policy modifications are propagated to the kernel by serializing the in-memory policy and loading it into the kernel. `sefuse` supports the modification of access vector rules, as well as the edition of recursive administrative permissions. The integration of our prototype is illustrated in Figure 3.13. As one can see from the figure, `sefuse` loads the same policy that is loaded in the kernel at boot time. At boot time, the policy is loaded in the kernel by writing it to the `load` pseudo file in the `selinuxfs` virtual filesystem, which is mounted under the `/selinux` path. `sefuse` uses the same mechanism to propagate policy changes into the kernel. The modified policy is also serialized to disk when the daemon shuts down, so that it will be used for the next boot. The administrative policy is stored separately from the policy; it is also loaded when `sefuse` starts and serialized back when `sefuse` shuts down. As `sefuse`, the administrative policy is protected from the rest of the system by being only accessible form within the `sefuse` domain.

The initial decision to implement the prototype with FUSE was made because it allowed for a much easier development (easier debugging essentially) than if we had developed the filesystem directly as a kernel component. This is also acceptable from a security standpoint, as we explain below. In this explanation, we list the hypotheses that we rely on to claim that our implementation is dependable. For each hypothesis we make, we explain why we think it is reasonable.

Hypothesis 3.4.2.1 *The implementation and configuration of Type Enforcement in SELinux constitutes a reference monitor.*

SELinux was not designed from the start as a secure system: it was not designed first as a whole and then implemented according to its design. Instead, SELinux is

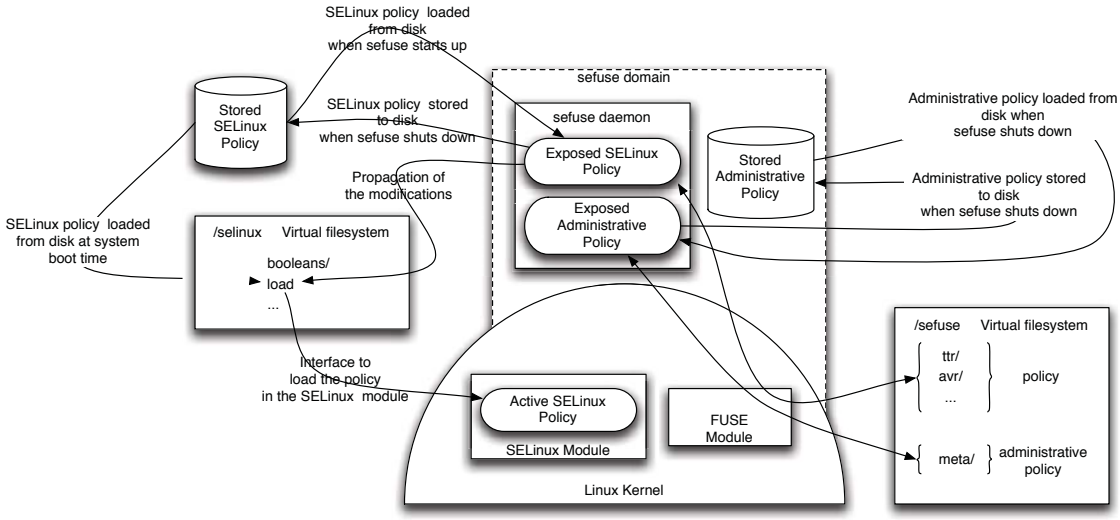


Figure 3.13.: Integration of `sefuse` within SELinux, to safely expose the SELinux policy as a virtual filesystem

a security extension that was retrofitted into the Linux kernel. As a result, there is no causal link from a model of a reference monitor to its implementation –this implementation being SELinux– that can be used to prove that the implementation of Type Enforcement in SELinux constitutes a reference monitor. Since it is not practically possible to prove that SELinux constitutes a reference monitor we explain why we think it is reasonable that our implementation assumes so, for each aspect of the definition of a reference monitor. For *full mediation*, the work of Zhang et al. [128, 129] provides a strong indication that the hooks of the Linux Security Module framework [109] are invoked on every access path where they should be. This indicates that the current implementation of SELinux as a security module is invoked to mediate accesses on all the access paths. For the *tamper-proof* aspect, SELinux is a security mechanism implemented inside a monolithic kernel, the Linux kernel. As a consequence, the integrity of the mechanisms of SELinux relies on the integrity of the Linux kernel. Assuring the integrity of a kernel of this size (a little less than 9 million lines of code for version 2.6.24 [130]) is not a tractable problem. It is therefore not possible to guarantee this integrity. The access control enforced by SELinux, however, helps reduce the attack surface of the kernel. Considering the *assurability*, the SELinux module shipped with version 2.6.24 of the Linux kernel is comprised of 16,3K significant lines of code[8]. There is no publicly documented effort on assuring the code of the SELinux security module. The above hypothesis has to be relied on but cannot be proven.

Property 3.4.2.1 *On SELinux, implementing the administrative model for Type Enforcement in userspace is not weaker, from a security standpoint, than implementing it in the kernel.*

`sefuse` can be protected from the rest of the system using TE confinement. This method of securely extending the operating system with userspace extensions, by

[8]This count of lines of codes was obtained using `sloccount`, available at `http://www.dwheeler.com/sloccount/`

relying on TE for protection, has been successfully demonstrated in LOCK [131]. A similar approach was successfully taken by Stern in the development of the Extended Access Control Subsystem on top of Trusted Xenix [132]. Formally, what we did is that we implemented a TCB subset [133] to regulate accesses to the security policy of SELinux.

Hypothesis 3.4.2.2 *Once the system is booted, only* `sefuse`*, running in its own isolated domain, is allowed to load the security policy in the kernel.*

If this hypothesis on the base security policy of the system is satisfied, then it is guaranteed that the only modification to the policy allowed at runtime are mediated by our administrative model. This hypothesis can be evaluated by searching for the `load_policy` permission in the security policy.

Hypothesis 3.4.2.3 *Our implementation of the administrative model is correct.*

Although no formal audit of our code has been performed so far, we have preserved the auditability of our code by keeping it small and clear.

Property 3.4.2.2 *Provided the administrative policy allows only changes that preserve the security goals of the system, our integration of an administrative model in SELinux does not weaken the system security.*

This property holds, provided hypotheses 3.4.2.2 and 3.4.2.3 hold. What we are interested in stating with this property is that the addition to SELinux of administrative *mechanisms* that implement the administrative *model* described in Section 3.3 does not by itself weaken the security guarantees that the system can offer. Indeed, the mechanisms that we have implemented can be configured to implement many administrative *policies*, from one administrative policy that grants absolutely no modification rights to the base policy, to an administrative policy that grants all administrative rights to any subject of the system.

3.4.3 Summary

We have presented the design and integration of our prototype implementation of the administrative model from Section 3.3. By using a virtual filesystem abstraction, this prototype allows common filesystem tools to be used to interact with both the base TE policy and the administrative policy. When analyzing aspects of the policy, we have found this interface convenient for read-only consultations of the policy. Our development platform was SELinux (on RedHat Fedora Core 10), where this prototype is implemented as a trusted subsystem. We have shown under which assumptions the implementation can be considered to be a reference monitor. The main assumption is that the Linux kernel's integrity has to be relied on.

3.5 Conclusion

In this chapter, we have formally modeled TE, using stuck semantics. In this model, we have isolated a core set of TE features (TE-core) that the rest of our work relies on. Our modeling has been performed in a modular fashion, where we have shown how the different features of TE can be composed together. We have then shown, by first modeling Core RBAC and then comparing Core RBAC and TE-core, that Core RBAC can not be used to address the confinement problems that we set to address. This justifies our choice of TE over RBAC as the base access control model, that we then extend with an administrative model. The administrative model for TE that we presented is able to precisely express which rules a given subject is allowed to modify. Moreover, this administrative model is recursive by nature, which allows the definition of an administrative policy on the administrative policy itself, and so on. This is an improvement other most existing RBAC administrative models, where the administrative policy requires on fully trusted subject for its configuration. Finally, we have presented the design of a prototype implementation of this model that was performed on RedHat Fedora Core 10.

4. OVERLAY LABELING: REFINING THE POLICY COUPLING

The original motivation for the work presented in this thesis was to let users refine the security policy of the operating system on which they work, and on which they are not administrative users. In other words, this idea was to let users themselves decide which of their ambient permissions they wish to extend to applications that run on their behalf. In the related work, we have argued why the security mechanisms provided on stock UNIX systems (and by analogy on other systems that rely on identity based access control) do not provide a satisfactory answer to this problem (see Section 2.5.1). In the previous chapter, we have introduced an administrative model that supports fine-grained delegation of administrative permissions on the TE policy.

This administrative model is a *necessary* step towards enabling users to refine the TE policy of the system. However, this administrative model is not *sufficient* by itself to support all the refinements to the policy that are necessary to support the fine-granularity of access controls that we set to achieve. Here is a concise example to motivate this chapter; we elaborate on it when we expose the grading program problem (see Section 4.1), which was the original motivation for this thesis.

While the administrative model we defined for TE supports the precise specification of allowed manipulations on TE constructs, it has no notion of which type is associated to which system objects. This is a reasonable design decision for the administrative model as it is also the case that the TE policy has no notion of this coupling, either. The implication, however, is that user policies are limited to using the existing labeling of objects. Since users do not have control on the system labeling decisions, their policies are limited to being expressed in terms of system-defined labels. For instance, most binaries installed on the system are labeled with a unique

type: `bin_t`. As a result, `/usr/bin/gcc` (the compiler) and `/bin/ls` (the utility to list files) can not be treated differently by the TE policy, which reasons on objects only through their types. The only workaround is for the user to make a copy of one of the programs and relabel that copy. This workaround isn't satisfactory as it results in wasted disk space, and updates to the original program won't propagate to the copy. Missing updates are particularly a problem when the updates address security or functionality issues. The labeling of network packets suffers from a similar limitation, without even a workaround.

In other words, without a way for users to extend the labeling of objects that they are not allowed to relabel, whatever TE policy they may write is constrained to the granularity with which these objects are currently labeled on the system. Allowing the users to relabel system objects would not be the correct solution, as it would let users break the existing semantics of the system policy. We have designed a solution that supports *overlay labeling* of filesystem objects and network packets.

This chapter starts by a presentation of the grading program problem (see Section 4.1), which concludes by a demonstration of how this problem is only partially solved with the administrative model from Chapter 3. The two following sections present the labeling of filesystem objects (see Section 4.2) and the labeling of network packets (see Section 4.3). In each of these two sections, we present how the labeling can be extended to support overlay labeling. Then, we present how the infrastructure designed for overlay labels can be re-used to support the grouping of object by predicates expressed on type attributes (see Section 4.4). Finally, we address issues that concern the correctness of these designs: whether they can be used to subvert the system policy (see Section 4.5), and whether their usage is reversible (i.e. is the deployment of an overlay label a destructive operation ?) (see Section 4.6).

4.1 Motivation

4.1.1 The Grading Program Problem

The *grading program problem*[1] is a historical motivating example for protection systems [69], where an automatic grading program is used to evaluate the functionality of a student's assignment submitted for grading. The core of the problem is trust (or lack thereof) between the grading program and the graded program, also know as the *mutually suspicious subsystems problem* [135]. While the submission being graded is a software conceived by the student, prepared for submission on his/her account, the grading is done by the teaching assistant (TA), who runs the grading program (and hence the student submission as well) in his TA account.

The execution of the grading program needs to take place in a confined environment in order to:

- **Enforce the assignment restrictions**

  System programming assignments consist in having the students program features that are otherwise provided by the same system on which the students develop their programs. It is therefore necessary to confine the execution of the submitted programs, to make sure that they do indeed implement the features they should, as opposed to delegate the processing to the features offered by the host system. A common example of such cheating is when a student, instead of implementing a filesystem per the assignment specifications, programs stubs that call their corresponding routines from the filesystem of the host OS.

- **Protect the TA account from the submitted program**

  The submitted program may (and regularly is) misbehaved, in which case the TA account needs to be protected from *accidental damage*. This is an instance of the *debugging problem* [7,69], and it is partially addressed by memory protection

[1] This presentation of the grading problem is slightly different from the original presentation in [134], but the core of the problem remains the same

and quota mechanisms. The submitted program can even be *malicious*, in which case it constitutes a *trojan horse* [4].

The first point, enforcing the assignment restrictions, is already addressed by existing techniques of library interposition that were developed for goals including intrusion detection. Moreover, this part has more to do with detection than prevention: it is not a security issue that the submitted program uses system libraries it shouldn't. What really matters is to detect such uses, so that the students do not get credit for relaying calls to the proper system library when the goal of the assignment was to have them implement the library itself. In short, the first part of the grading problem does neither justify nor require new access control research, thus we do not address it; we focus on the second part instead, protecting the TA account from the submitted program.

4.1.2 Example Programming Assignment

We now present a programming assignment that will provide some substance to the administrative operations we introduce later in this section: refining the labeling on the filesystem, and creating domains. The networking aspects, although introduced here, will be covered in another section (see Section 4.3).

The assignment consists in programming a simple HTTP server, as required in undergraduate network programming classes (e.g. CS422 at Purdue's department of Computer Science). The features that must be implemented by this server are the following. It must be capable of serving static files and support directory browsing. It must also support CGI scripts, and loadable modules (shared libraries instead of scripts).

The corresponding restrictions that have to be enforced on the assignment are:

1. The access of the HTTP server to the filesystem should be restricted so that it can not serve the whole content of the TA's home directory. More precisely, the server should serve only the static content located within the directory

designated for static content. Since the electronic grade sheets are stored in the TA's account, the server could otherwise potentially deliver the grade sheets of the whole class. This would be a legal issue in the USA (FERPA act [136]).

2. The filesystem access restrictions should be discriminating enough to let CGI scripts access common command-line utilities (e.g. the programs from GNU coreutils package: `expr`, `echo`, `printf`, `test`, ...), while still enforcing the confinement explained above. For instance, one of the example scripts provided to the students uses the `cal` command-line tool to generate textual calendars.

3. The network accesses of the server should also be restricted. Incoming access should be restricted so that only connections coming from machines used in the grading tests are passed to the server. The benefit of this restriction is to further limit the exposure of the server to external attacks. The server should be prevented from initiating remote connections, so that a trojan web server will not be able to spontaneously submit files from the grading system to another system. This is useful when the testcases are not publicly released.

We now present how the administrative policy, based on the administrative model from the previous chapter, can be configured to let the TA set up a TE configuration that enforces these restrictions. More precisely, we will show *how far* these restrictions can be enforced. The parts that can not be enforced are the motivation for the following two sections (see Sections 4.2 and 4.3).

4.1.3 Creating Types and Domains and Configuring Accesses

We will now show how the TA can set up a confinement domain in which the grading can safely be run.

The first step in setting up the confinement domain is to create the domain itself. In order to do that, the TA must create the domain type (say, `grading_t`), and attach the domain attribute to it. Let us focus on the type creation first. The problem with

allowing the creation of new types is that it is hard to write rules on how they can be manipulated, until they exist. And at that point, they do not need to be created anymore.

Our solution to this problem is to let users create types "through" types attributes. In other words, an attribute is given as a part of the request by the user to create a type. If the user is allowed to create types that bear this attribute, then the request is satisfied: the type is created and the specified attribute is attached to the type. The next step is for the TA to turn the newly created type into a domain, by attaching the `domain` attribute to the `grading_t` type. This operation is also allowed by a rules that specifies that the TA can attach the `domain` attribute to types that bear a given attribute.

We are starting to see that type attributes are an important part of the administrative policy [2]. As illustrated in Figure 4.2, type attributes can be used to group types. These groups can then be used instead of types in the specification of access vector rules. This is how administrative permissions are granted to the TA to let him create the type for the domain and then attach the `domain` attribute to it. The policy statements and administrative rules needed to let the TA create a grading domain are reproduced in Figure 4.1.

Once the TA has created a grading domain, the next step is to declare the entry point(s) of the grading domain, as well as setting up automatic domain transitions on these entry points, and allowing them. That is, the TA has to designate which type(s) of executable files can be used to enter the grading domain. One way to do this is to create a type to label the grading script, say `grading_exec_t`. Then, with the grading script relabeled to the `grading_exec_t` type, set a domain transition

[2]Type attributes were also used when constructing the emulation of RBAC on top of TE (see Section 3.2.2).

[3]In the reference policy, this permission is granted by attaching the `file_type` attribute to a type. The reference policy contains a rule specifying that types bearing this attribute can be used to label filesystem objects. A variation of the rule we presented could let the TA attach the `file_type` attribute to the types he has created.

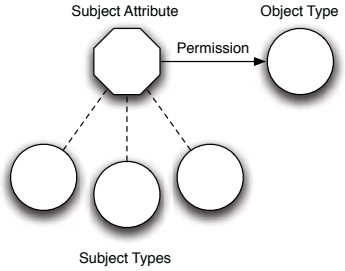
[4]Besides the permission to activate a swap file or a quota file.

Existing policy elements	
`type ta_t;`	The type associated with the TA processes
`type ta_file_t;`	The type associated with the TA files
`attribute ta_type;`	Attribute for tagging the types created by the TA
Existing policy rules	
`allow ta_t ta_type:file { relabelto relabelfrom }`	The TA can relabel to and from types that bear the `ta_type` attribute
`allow ta_t ta_file_t:file { relabelto relabelfrom }`	The TA can relabel to and from the type normally associated with his home directory (`ta_file_t`)
Existing administrative rules	
`allow ta_t ta_type:type create;`	The TA can create types "through" the `ta_type` attribute
`allow ta_t ta_type:attribute(domain) attach;`	The TA can turn types with the `ta_type` attribute into domains
`allow ta_type self:filesystem associate;`	The types created by the TA (with the `ta_type` attribute) can be used to label filesystem objects[3]

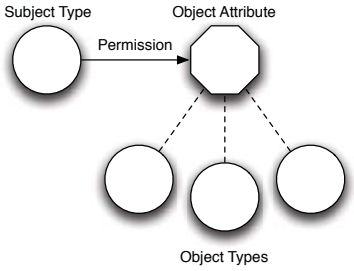
`allow ta_t *:av_rule({ta_t ta_type}, ta_type, \`
`file , ~ { swapon quotaon } ) \`
`{ insert remove }`
	The TA can grant himself and his domains almost any permission[4]on files of the type he has created

`allow ta_t *:tr_rule({ta_t ta_type}, {ta_t ta_type},\`
`* , {ta_t ta_type} ) \`
`{ insert remove }`
	The TA can set up type and domain transitions between any of the types he has created

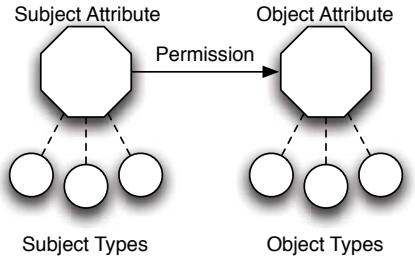
Figure 4.1.: Example configuration that lets the TA create domains



(a) Type attributes can be used to group subjects and their permissions. In this case, a subject can be seen as having permissions "through" one of its type attributes.



(b) Type attributes can also be used to group objects, and therefore the access rules that regulate their accessibility. In this case, an object can be seen as being accessible "through" its attribute.



(c) In the general case, both the subject and object of a TE rule can be expressed using attributes.

Figure 4.2.: Type Attributes. Attributes can be used in two main ways. By tagging/untagging a subject type with an attribute, permissions can be added/removed to that type, if the attribute is used in rules like (a) (and more generally like (c)). The same tagging mechanism can be used to make an object type accessible, if the attribute used for tagging is already used in a rule like (b) (and more generally like (c)).

from the ta_t domain to the grading_t domain upon execution of executables of the grading_exec_t type. Contrary to the original design in Flask [9], SELinux does not require an extra permission to connect the type being from and the type being relabeled to. If needed, one could use the validate_trans statement to constrain the set of allowed relabelings. We have not addressed this issue; neither does the SELinux reference policy.

As a result of creating the grading domain, creating the type for the grading domain executable entry point, and setting up the necessary type transition and access vector rules, the following statements are added to the policy:

```
type grading_exec_t
typeattribute grading_exec_t ta_type
allow grading_exec_t self:filesystem { associate } ;
allow ta_t grading_exec_t:file { read getattr execute } ;
allow grading_t grading_exec_t:file { entrypoint } ;
type_transition ta_t grading_exec_t:process grading_t
allow ta_t grading_t:process transition
```

Additionally, the TA can set up filesystem type transitions so that the log files created during the run of the grading are automatically typed with the grading_log_t type, and accessible in append-only mode from within the grading domain. This would result in the following additional statements in the policy:

```
type grading_log_t
typeattribute grading_exec_t ta_type
allow grading_log_t self:filesystem { associate } ;
allow grading_t grading_log_t:file { create link append } ;
allow grading_t grading_log_t:dir { search write add_name } ;
type_transition grading_t grading_log_t:dir grading_log_t
```

4.1.4 Limitations in the Mapping of Types to Objects

We have just showed that the administrative model we designed in Chapter 3 can support delegated administration of the TE policy, even when it involves the creation of new types and domains. However, the granularity at which regular users can write policy statements is still limited, from a system perspective, by the granularity of the mapping from types to objects. This mapping is defined outside of the TE policy.

As a consequence, even if the users can have a fine control on the TE policy, they may be forced to give up the principle of least privilege due to the granularity at which system-owned objects are labeled. For instance, the CGI scripts that the example assignment has to support only need access to a few of the system binaries (e.g. `date`). Furthermore, they should *not* be allowed to run the `mail` command under the TA's identity. Indeed, although the grading program runs inside the `grading_t` domain, it still runs under the TA's UNIX identity. This means that emails sent from within the grading domain would be sent under the TA's identity. Unfortunately, both `date` and `mail` are labeled with the same `bin_t` type, which prevents from distinguishing them within the policy. Similarly, although network packets can be labeled, the specification of the labeling is also done outside of the TE policy.

The only partial workaround is for the TA to make a copy of the subset of system-owned files that he wishes to let the confined grading environment have access to. This workaround introduces duplicates of system files, which is a known issue with `bsd` jails [86]. Having these duplicates introduce the need for reconciling them (conflict-resolution for data files; propagation of the system updates for executables), and wastes disk space. There is no workaround for network packets that does not involve re-writing firewalling code, which is notoriously hard to get right [137].

In the next two sections we explain, in turn, how we addressed this problem for filesystem objects labels and for network packets labels. Our solution preserves the semantics of the existing system policy.

4.2 Refining Filesystem Objects Labels

As we explained in Chapter 3, the TE policy considers an object only through the type the type attached to the object and the type attributes attached to this type. In the SELinux terminology, attaching a type to an object is called *labeling* the object.

The labeling of persistent filesystem objects is an important part of the deployment of a TE policy on SELinux. This labeling couples the policy, expressed in terms of types, to the concrete files that are used during the boot process, to boot the system in a trusted state. More generally, the type attached to each file and directory determines which parts of the filesystem a process can read to, write from, and execute code from. These are essential aspects when one wants to practically assure a system.

Two antagonistic goals are at play when defining the labeling of the filsesystem. On one hand, one wants to define as few labeling rules as possible, to keep the set of rules tractable. On the other hand, one wants to define very precise labeling rules to enforce a fine grained-policy. However, every new rule potentially conflicts with an existing rule (and raises the need for a means of resolving rule conflicts). Also, a finer-grained policy yields a larger set of labeling rules, which is harder to audit.

In the rest of this section, we will present how the filesystem labeling works in SELinux and why the standard implementation does not let users extend the labeling. Then we present our solution that lets users extend the labeling so that they can write policy statements at a granularity level of their choosing. In other words, our solution lets users define finer, coarser, or simply different groupings of objects.

4.2.1 Filesystem Labeling Specifications

SELinux relies on a filesystem labeling specification that indicates which label should be attached to which filesystem object. On a Fedora 10 distribution that uses the targeted policy, the current specifications can be found in the file `/etc/selinux/targeted/contexts/files/file_contexts`. Each line of this file is composed of three tab-separated fields. In order, these fields are:

1. filesystem path pattern: an extended POSIX 1003.2 regular expression, which can specify one or several paths on the filesystem (regular expression branches '|' are supported in extended regular expressions).

2. an optional object class specification, that indicates the class of filesystem object the specification is supposed to cover. The class uses the same one-letter encoding used by the `ls` utility when using its long output format. If the class option is used, it should be prefixed with a dash sign ('-'). For example, one would pass "`--`" to limit the specification to regular files, "`-d`" for directories, and "`-b`" for block special files, "`-c`" for character special files, "`-l`" for symbolic links, "`-s`" for socket links, and "`-p`" for named pipes.

3. security context: either "<<`none`>>" or a full security context in the same form as displayed by the "`-Z`" option of `ls`. "<<`none`>>" indicates that the matching filesystem objects should not be labeled. This form of security context is used to avoid labeling objects that are automatically labeled by the system (e.g. the `/selinux` virtual filesystem). A full security context looks as follows, taking the context of `/bin/ls` as an example on a Fedora 10 system:

   ```
   system_u:object_r:bin_t:s0
   ```

   This context consists of four colon-separated fields: a TE user (`system_u` for all filesystem objects), a TE role (`object_r` for all filesystem objects), a TE type (`bin_t` is attached to most system binaries), and an MLS context (`s0` means sensitivity level 0, which is the default MLS context applied all over the system when the MLS policy is not activated, which is the case here). In the following, when we refer to labeling or relabeling a filesystem object, we are actually refering to the setting of the type field in the security context of that filesystem object. All other fields keep their default values that we just presented.

This file is used primarily by the `setfiles` utility (and the `fixfiles` utility wrapper script), which is used to label the filesystem in two main cases. When SELinux

is deployed on a system that did not have SELinux active, the filesystem needs to be labeled so that filesystem objects are properly labeled and the policy can be enforced. The second case is to restore the file contexts to their proper values, on a system where SELinux was previously deployed but temporarily disabled. Indeed, when SELinux is disabled, so are the type transition rules that would normally guarantee a proper labeling of newly created filesystem objects (this default behavior is the result of type transition rules, which were described in Figure 3.3, Chapter 3).

4.2.2 Filesystem Labeling: Semantic Limitations

To motivate the next part of this section, we now present the stock semantics of filesystem labeling on SELinux and how they prevent the realization of the grading example.

When `setfiles` decides how to label a file according to the labeling specifications, it reads rules from the specification file and labels each file according to the last rule that matches it. This means that the last matching rule determines entirely how the file will be considered from the perspective of the TE policy. Moreover, since a file can be labeled with only one security context, and a security context can contain only one type, then a file can only have one type attached to it, even if one were to use other means of applying the labels than running `setfiles`.

Let us consider the grading program example again. Typically the whole subtree of the filesystem starting in the TA home directory will be labeled with the type `user_home_dir_t`[5]. What we would like to be feasible is the following. The TA would label a subset of his home directory with a type that indicates that this part of the filesystem is used for grading student submissions, say `grading_files_t`. The TA would then define a grading domain, say `grading_t` in which the grading program would run. This domain's filesystem write accesses would be limited to this subset of the filesystem to protect the TA account from accidental damage. This scenario can be

[5]There are few exceptions, for instance to protect the `ssh` configuration files and keys.

handled with stock SELinux and our administrative model from Chapter 3, provided the TA is granted the proper administrative priviliges on the `grading_files_t` and `grading_t` types.

However, an important part of the filesystem access restrictions cannot be handled by a stock SELinux system, even with the administrative extensions from Chapter 3. The limitation of having a single type attached to an object is problematic when a user, here the TA, would like to refine the accesses granted on these objects. As explained in Section 4.1.4, existing solutions are not satisfactory as they involve either duplicating or relabeling files, both of which are problematic. We present our solution to this problem in the next section.

4.2.3 Overlay Labeling of Filesystem Objects

Now that we have presented the limitations of filesystem labeling in a stock SELinux system, we present our solution to this problem. A satisfactory solution should allow users to overlay labels of their choice on top of system types and create policy statements that refer to these labels, without breaking the system policy statements that refer to the existing (system) labels. Our solution uses type attributes to support multiple labels per filesystem objects.

Our solution rests on the observation that type attributes can be used instead of types in most policy statements[6]. Based on this observation, our solution consists in "promoting" policy types. In type promotion, a *synthetic type* is generated to replace the original type from the policy and the original type name becomes an attribute of the synthetic type. This process is illustrated in Figure 4.3. After type promotion the access control decisions of the policy are bound to type attributes, and not types anymore. At this point, it becomes possible to relabel objects without breaking the semantics of the system policy, as long as the promoted type are properly handled.

Type promotion can be performed at different times: it can be performed on demand, or it can be performed ahead of time. We chose to perform ahead of time type

[6]Our solution also handles the special case of the field that specifies the new type in a type transition.

promotion. This yields a simpler algorithm and offers better performance guarantees: only the objects belonging to the new overlay need to be relabeled. The pathological case with on demand type promotion is when the creation of an overlay necessitates the relabeling of a significant fraction of the filesystem objects, as this will potentially take a *very* long time.

The next two parts of this section present in more details how type promotion is performed and, assuming all filesystem types have been promoted, how to handle the request by a user to overlay his own labeling on filesystem types.

Type Promotion

The process of promoting a type involves two aspects: promoting the type in the TE policy, which we describe first, and adjusting the filesystem labeling to effect the policy changes, which we describe last.

Promoting a type in the policy consists of the following steps, which are illustrated in Figure 4.3:

- generate a synthetic type: generate a type name that does not yet exist in the policy, and add it to the policy
- attach attributes of the promoted type to the synthetic type
- replace the promoted type by an attribute of the same name
- attach this attribute to the synthetic type

Since most fields of TE rules can accept types and types attributes interchangeably (see Section 3.1.2) the rest of the policy is unaffected by the type promotion, except for type transition rules. Indeed, the field that specifies the new type of an object (after the transition) only accept types. Indeed, a type attribute refers to the set of types that bear it whereas a type transition rule specifies the *single* new type that will be attached to an object. As a result, the following extra step needs to be taken when promoting a type:

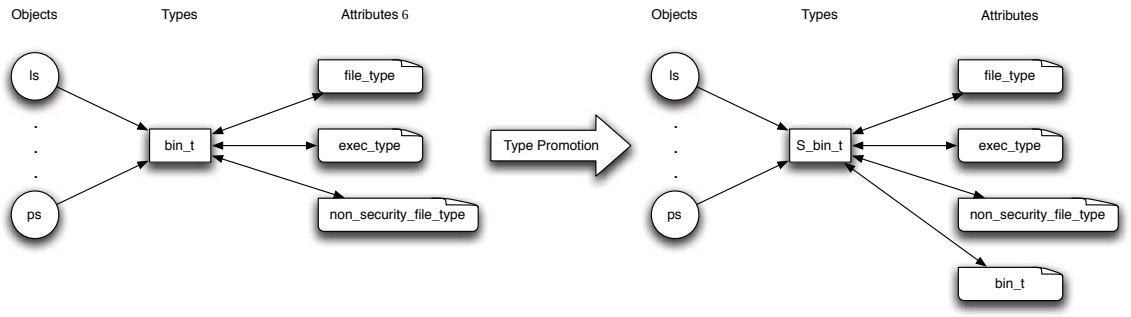


Figure 4.3.: Type promotion consists in generating a synthetic type name (here, `S_bin_t`) to replace an existing type (here, `bin_t`), which becomes an attribute of the new synthetic type. Type promotion is necessary to support overlay labeling. The type promotion algorithm is provided in Algorithm 1.

- if there are type transition rules which define transitions to the promoted type, they need to be replaced by type transitions to the synthetic type.

The adjustments to the filesystem labels are performed using the same mechanism that is used to deploy the initial labeling or fix a corrupted one. A copy of the original filesystem labeling specification (`\tt/etc/selinux/targeted/contexts/files/file_contexts`) is made. In this file, for each promoted type, and for each context that refers to it, the type component of the context is replaced by the synthetic type that was generated as part of the type promotion. This file is then a specification of how to correct the filesystem labeling, now that types have been promoted. This specification can be used by the same tool that is used for labeling filesystem objects: `fixfiles`.

Creation of a Label Overlay

We now present how the request by a user to overlay his own labeling on filesystem types is handled, provided all filesystem types have been previously promoted.

A user requests the creation of a label overlay by providing a regular expression that determines the filesystem objects to relabel, and a string that will be used as the name of the type attribute that will materialize the overlay.

Algorithm 1 Promote type xxx

$synth\_type \leftarrow new\_synth\_type()$
$policy.types \leftarrow policy.types \cup \{synth\_type\}$
for all $attr_i \in xxx.attributes$ **do**
  $synth\_type.attributes \leftarrow synth\_type.attributes \cup \{attr_i\}$
end for
$policy.types \leftarrow policy.types \setminus \{xxx\}$
$policy.attributes \leftarrow policy.attributes \cup \{xxx\}$
$synth\_type.attributes \leftarrow synth\_type.attributes \cup \{xxx\}$
for all $ttr_i \in policy.type\_transition\_rules$ **do**
  if $ttr.target\_type = xxx$ **then**
    $ttr.target\_type \leftarrow synth\_type$
  end if
end for

The overlay labeling involves the following steps, which are formally presented in Algorithm 2 and illustrated by an example in Figure 4.4:

- create a type attribute for the overlay, and add it to the policy
- create the synthetic types required to represent the overlay: the regular expression will cover a non-empty set of filesystem objects (otherwise, there is nothing to do). Since this set of objects is non-empty, the set of existing types used to label these objects will itself be non-empty. These existing types are actually synthetic types themselves, since the type promotion was performed ahead of time. For each of these existing types, a new synthetic type needs to be created to represent the intersection of the existing type and the overlay label. The new synthetic type will bear the overlay attribute and all the attributes of the existing type, hence materializing the intersection of the overlay and the existing type.
- relabel the filesystem objects designated by the regular expression: each designated object is relabeled with the synthetic type that represents the intersection of its current type and the overlay.

Please note that, contrary to type promotion, the overlay labeling does not involve any adjustment of the type transition rules. This is not an oversight. By leaving type transition rules unchanged, overlay labeling preserves the system policy that governs how new objects are supposed to be labeled upon creation. Indeed, overlay labeling is supposed to let users attach additional labels on objects for which they do not have administrative privileges. It should not let users change the default labeling rules for types on which they do not have administrative privileges.

4.3 Network Packets Labels

Network packets are one of the classes of objects to which access is regulated by SELinux. Network packets can therefore have a type attached to them, and this type

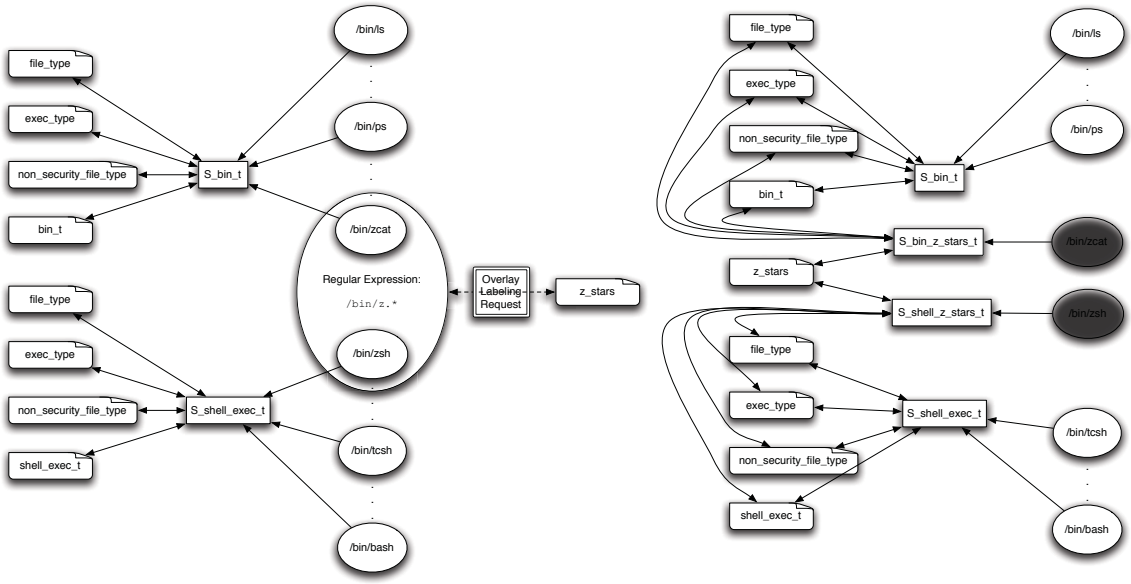


Figure 4.4.: Overlay Labeling of Filesystem Objects. In this example, the user requests to label all files that match regular expression `/bin/z.*` with the label `z_stars`, which happens to match `/bin/zcat` and `/bin/zsh`.

Algorithm 2 Overlay labeling of files that match the regular expression *regex* with attribute *attr*

Require: $attr \notin policy.attributes$
  $policy.attributes \leftarrow policy.attributes \cup \{attr\}$
  $synth\_types \leftarrow \varnothing$ {Track the synthetic types that get generated}
  for all $file_i \in regex.matches$ **do**
    $type_i \leftarrow file_i.type$
    if $synth\_types[type_i] = null$ **then** {Construct a synthetic type if needed}
      $stn \leftarrow new\_synth\_name()$
      $synth\_types[type_i] \leftarrow stn$
      $policy.types \leftarrow policy.types \cup \{stn\}$
      for all $attr_i \in type_i.attributes$ **do**
        $stn.attributes \leftarrow stn.attributes \cup \{attr_i\}$
      end for
      $stn.attributes \leftarrow stn.attributes \cup \{attr\}$
    else
      $stn \leftarrow synth\_types[type_i]$
    end if
    {Relabel the object with the synthetic type}
    $file_i.type \leftarrow stn$
  end for

is used when evaluating access control according to the TE policy. As for filesystem objects, we find it desirable to let users overlay their own labels on network packets, provided that this feature can be supported while preserving the semantics of the system policy. In the context of the grading program problem, this feature is useful to let the TA confine submitted web servers to listening to a given port number, and to let them access only packets that come from the machine used to send them test requests.

In this section, we first present the mechanisms that are available on SELinux for the labeling of packets. Then, we present how the packet filtering features of SELinux are used to support the packet labeling. After introducing formally the problem of overlay labeling of network packets, we show that an encoding on top of the normal labeling infrastructure would have poor performance. We then present our solution, based on interval trees.

4.3.1 Overview of Packet Labeling on SELinux

SELinux provides several mechanisms to specify and apply labeling decisions to network packets. We provide an overview of these mechanisms and the policies they support.

There are two main cases in the operation of a networked SELinux system, when considering a given network connection. The system at the other end of the connection either supports labeled networking, or it doesn't. If it does, then the labeling can be done according to a mapping of the remote *peer* labels (and therefore its labeling policy) to the local labels (defined by the local labeling policy). If the remote system does not support labeled networking, then the labeling decision has to be made according to the local policy exclusively.

Peer Labeling

Peer labeling is used for implementing a distributed trusted application, as it supports verifying the distributed deployment of the application along the lines of the trusted network interpretation [138] of the "Orange Book" or the shared reference monitor [139]. There are several means of configuring peer labeling. Currently, two mechanisms are available for peer labeling, Netlabel and labeled IPSec extensions.

Netlabel [140] supports the transfer of labeling information by using the Commercial IP Security Option (CIPSO) [141] extension of the IPv4 protocol. There the domains are only defined in MLS terms: a sensitivity level and a set of categories. Very few environments offer the physical network security required for a direct deployment of CIPSO. Since CIPSO consists in a set of options in IP packet headers, it is vulnerable to the same spoofing issues as the IP protocol. This issue can be worked around by protecting CIPSO-labeled traffic inside a VPN tunnel, or by using labeled IPSec extensions.

Labeled IPSec extensions [98] have been developed to support the transmission of peer labels inside IPSec Security Associations (SAs). The labeled IPSec extensions support the exchange of full SELinux security contexts, but are currently conflicting with another IPSec extension: Explicit Congestion Notification (ECN). The Internet Key Exchange (IKE) negociation establishes the context of each child SA, which establishes the context of the remote peer (a process in a given domain). The Labeled IPSec extension was submitted as a draft to the IETF on July 10th 2009 [7].

As our work does not cover the enforcement of distributed security policies, we will not be covering peer labeling further.

Local Labeling

Local labeling is useful in all the cases not covered by peer labeling. Actually, most installations of SELinux fall in this category. First, most installations lack the

[7] `http://tools.ietf.org/id/draft-jml-ipsec-ikev2-security-context-01.txt`

physical network security required for CIPSO deployment. Second, the deployment of either VPN tunnels to protect CIPSO traffic or IPSec peering (with the labeled extensions) is difficult on systems that are not under the same administrative domains. Indeed a pre-requisite to peer labeling is to agree on either the same labeling scheme or a mapping from one scheme to the other.

SECMARK is the current mechanism for local labeling. It is built on top the netfilter framework [142], as an extension of the packet mangling features. Practically, this means means that the labeling rules for packets are written as part of the packet filtering rules. This mechanism replaces the original packet labeling that was dropped when SElinux transitioned from being an external patch to being a security module shipped with the standard Linux kernel, where it is integrated on top of the of the Linux Security Module framework [109]. The older mechanisms, which relied on the specification of network security contexts within the security policy, were supported with the `compat_net` backward-compatible code. `compat_net` has been dropped out of the Linux kernel as of version 2.6.30 [99].

Our Choice

Our works supports overlay labeling of network packets on locally-defined labels. More specifically, since the `compat_net` support is officialy phased out [99], our solution builds on top of the SECMARK functionnality. While we do not specifically address overlay labeling for peer-defined or peer-negociated labels, we believe that the techniques we develop for overlay labeling in this section and the previous one are applicable to these scenarios.

4.3.2 Overview of the Netfilter Framework and `iptables` Implementation

Netfilter is the packet filtering framework embedded in the last three major versions of the Linux kernel (3.0.x, 2.6.x, and 2.4.x series). `iptables` is the standard tool for used for configuring netfilter. SELinux uses an extension of `iptables`, SEC-

MARK (described in the next section), to apply local labeling rules to packets. In this section, we present the functionalities offered by `iptables`, and the semantics of the rules matching. We do not discuss the new `nftables` filtering framework as is still in an early stage of development and not yet integrated in the standard Linux kernel.

Since netfilter is a framework, it does not provide functionnality by itself. Instead, it exposes an API through which packet processing extensions can register functions that will be invoked to process packets. `iptables` is a packet processing extension built on top of netfilter. This extension exposes several *tables*, which correspond to different aspects of the processing of packets, e.g. `filter` for packet filtering, `nat` for network address translation, and `mangle` to alter packets. For each table, several default chains are available, corresponding to the different steps in the processing of a packets that are relevant for that table. These steps are illustrated in Figure 4.5; our focus is the `mangle` table because it is the table that is used for labeling packets[8].

`iptables` is supplied with a tool, called `iptables`, which is used to edit the list of rules contained in chains. Rules can be added, removed, and inserted in each chain. Each consists of a set of matching specifications (e.g. a range of source IP addresses and a destination TCP port) and a target, wich specifies what should be done to a packet that matches the rule. A packet can leave a chain in one of the following ways: a matching rules specifies a final decision for the packet, the end of the chain is reached, or the packet is sent to another (user-defined) chain. When a packet is sent to another chain, it may or may not return to the current chain, depending on whether a final decision is reached in the chain the packet is sent to.

Inside a given chain, the semantics of the rules matching is that the first matching rule is applied. If the target of the rule indicates a final target, the decision is applied and processing stops at this rule. There are several kinds of final targets. ACCEPT and DROP are built in targets, to respectively accept or silently drop a packet. An

[8]A full discussion of netfilter is beyond the scope of this work. A detailed diagram of the flow of packets through the netfilter framework can be found in Figure A.1, for reference. That diagram shows which default chain is available for each table, as well as the order in which these chains are invoked as a packet flows through the system.

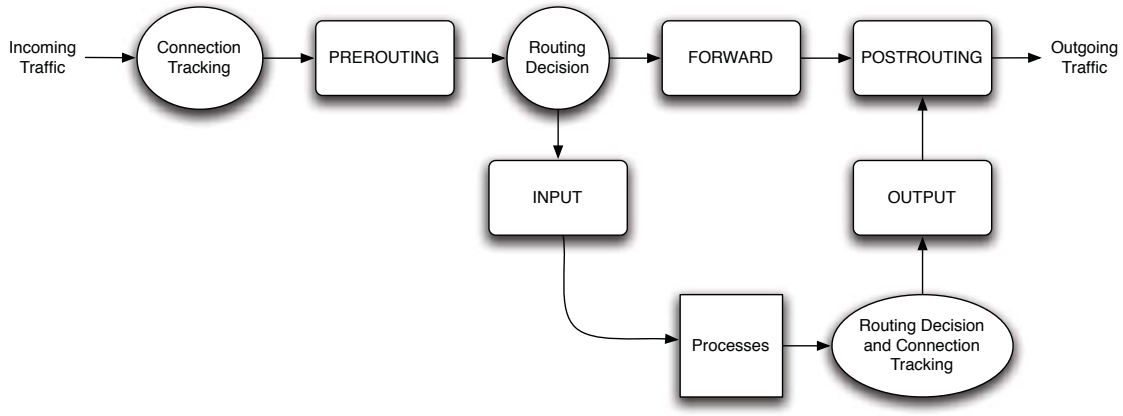


Figure 4.5.: Default chains in netfilter: input, forward, output, pre-routing and post-routing. These five chains are available to the netfilter `mangle` table, which is used for packet labeling. The `filter` table, which is applied after the `mangle` table, does not have the pre-routing and post-routing chains. With filtering, which embodies traditional network firewall techniques [100], the access-control decision is made exclusively according to filtering rules. With labeling, the decision can be delegated to another component. For instance, packets labeled with security contexts are decided upon by the SELinux enforcement mechanisms, based on the policy of the system. The chains of interest to confine processes are the input and output chains, where packets can either be filtered or labeled.

extension of `iptables`, REJECT, can be used to precisely define how the rejection of the packet should be manifested (e.g. sending an ICMP "port unreachable" packet or a TCP reset packet). The QUEUE target can be used to send packets directly to user-space. In some cases, RETURN can be constitute a final target, as discussed below.

The end of a chain can be reached in two ways: either processing reached the last rule, which doesn't match, or the current rule matches, and its target is RETURN. Two things can happen when the processing of a packet reaches the end of a chain. If the current chain is a built-in chain, then the default chain policy (either ACCEPT or DROP) is applied. If the current chain is a user-defined chain (explained below), it can not have a default policy. In this case, processing returns to the calling chain, and processing resumes in the calling chain, after the rule whose target was to call the current chain.

Here is a simple example of how the filtering of ICMP packets can be configured with `iptables`[9]:

```
iptables -A OUTPUT -o eth0 -p icmp -m state \
             --state NEW,ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -i eth0 -p icmp -m state \
             --state ESTABLISHED,RELATED -j ACCEPT
```

The first rule authorizes (`-j ACCEPT`) any ICMP packet (`-p icmp`) to be sent by the machine (`-A OUTPUT`) on interface `eth0` (`-o eth0`). The second rule authorizes only icmp packets that either are in response to a previous ICMP packet (`--state ESTABLISHED`) or are related to another connection (`--state RELATED`) being tracked by the connection-tracking subsystem. The connection-tracking supported by the CONNTRACK netfilter module is critical to supporting this precise filtering of ICMP packets. Prior to CONNTRACK, the solution consisted in allowing

[9]This example is based on the simple firewalling script by James Stephens, available at `http://www.sns.ias.edu/~jns/`. ICMP packets are used for network diagnostics, for instance by the `ping` command.

only ICMP "echo reply" packets to enter the system. However, some attacks took advantage of this common rule for Denial of Service (DoS) purposes, for instance the smurf attack. The rule presented in the above example avoids this problem.

We mentioned user-defined chains briefly above. User-defined chains can be used to support function call semantics in the structure of the firewalling rules. That is, the filtering of a packet can be delegated by one chain to another. This is done by using the called chain as a target of a filtering rule in the calling chain. By putting final targets in the rules of the called chain, this feature allows structuring the rules as a call tree. The advantage of using this tree structure to group rules is that it limits the number of rules that have to be be processed every time a packet flows through the system.

For example, the following set of `iptables` commands create a new chain (`ssh-chain`), to which the processing of all `ssh`-related packets (TCP traffic going to or from port 22) is delegated:

```
iptables -N ssh-chain
iptables -t filter -A INPUT -p tcp --sport 22 -j ssh-chain
iptables -t filter -A OUTPUT -p tcp --dport 22 -j ssh-chain
```

This structuring of rules can be used to support the encoding of decision trees, as we will see later. To support custom traffic shaping, `iptables` also supports adding arbitrary 32bit marks on packets. This feature is provided by the MARK extension. The idea behind this feature is to decouple packet classification (the grouping of packets in service classes) from the enforcement of a differentiated quality of service policy on these packets. The MARK extension can be coupled with connection tracking for faster classification of packets that are part of an ongoing connection (this extension is called CONNMARK and is coupled with CONNTRACK). In this case, the first packet that initiates the connection is used to determine the mark attached to the connection. Following packets, which are identified as part of the same connection, are marked based on the mark attached to the connection. The advantage of this

approach is that the potentially complex process of deciding which mark to attach to a packet is only performed explicitly once, when a connection is initiated. Following packets of the same connection are classified much quicker by recalling the mark attached to the connection.

Here is an example set of rules[10] that splits connections in two categories for a simple load balancing scheme. This set of rules uses three user-defined chains: one for restoring the mark on established connections (`RESTOREMARK`); the two others (`CONNMARK1` and `CONNMARK2`) are used to set marks on new connections.

```
iptables -t mangle -N CONNMARK1
iptables -t mangle -A CONNMARK1 -j MARK --set-mark 1
iptables -t mangle -A CONNMARK1 -j CONNMARK --save-mark
iptables -t mangle -A CONNMARK1 -j LOG \
        --log-prefix 'iptables-mark1: ' --log-level info

iptables -t mangle -N CONNMARK2
iptables -t mangle -A CONNMARK2 -j MARK --set-mark 2
iptables -t mangle -A CONNMARK2 -j CONNMARK --save-mark
iptables -t mangle -A CONNMARK2 -j LOG
        --log-prefix 'iptables-mark2: ' --log-level info

iptables -t mangle -N RESTOREMARK
iptables -t mangle -A RESTOREMARK -j CONNMARK --restore-mark
iptables -t mangle -A RESTOREMARK -j LOG \
        --log-prefix 'restore-mark: ' --log-level info

# restore the fwmark on packets that belong to an existing connection
iptables -t mangle -A PREROUTING -i eth0 -p tcp \
```

[10]This example is an excerpt from `http://www.sysresccd.org/wiki/index.php?title=Sysresccd-networking_en_Iptables-and-netfilter-load-balancing-using-connmark&printable=yes`

```
        -m state --state ESTABLISHED,RELATED -j RESTOREMARK

# if the mark is zero if means the packet does not belongs to an
# existing connection
iptables -t mangle -A PREROUTING -p tcp -m state --state NEW \
        -m statistic --mode nth --every 2 --packet 0 -j CONNMARK1
iptables -t mangle -A PREROUTING -p tcp -m state --state NEW \
        -m statistic --mode nth --every 2 --packet 1 -j CONNMARK2
```

In the next section we present how the packet marking infrastructure is used in SELinux for attaching security contexts to network packets.

4.3.3 TE Packet Labeling with the SECMARK SELinux Extension

The SECMARK and CONNSECMARK targets of `iptables` are the Linux Security Module counterpart of the MARK and CONNMARK extensions. The SECMARK and CONNSECMARK targets are used to decouple packet labeling from access control enforcement on these packets, in the same way that MARK and CONNMARK are used to decouple packet classification from service differentiation.

The SECMARK and CONNSECMARK targets can only be used in the mangle table. Figure 4.5 shows the default chains available in the mangle table, and where these chains are placed in the flow of network packets.

Supporting the overlay labeling of network packets involves the evaluation of potentially large combinations of criteria, in order to determine which synthetic type to attach to a packet. The fact that targets can trigger a jump to or a return from user-defined chains could potentially be used to encode the overlay labeling rules for network packets in terms of a decision tree materialized by a set of `iptables` chains, where each chain would correspond to a decision. Such an encoding is discussed in Section 4.3.4. In the remainder of this section, we present an example of how the labeling of packets, together with an access vector rule in the TE policy, can be used to enforce access control on network traffic.

The following example demonstrates the use of the SECMARK and CONNSECMARK features available for the labeling of network connections, applied to the labeling of both control and data connections of FTP[11]. We explain the meaning of these rules contained below, and provide a graphical illustration of their meaning in Figure 4.6.

```
1  # Ensure the FTP helper is loaded
2  modprobe ip_conntrack_ftp
3
4  # Create a chain for connection setup marking
5  iptables -t mangle -N SEL_FTPD
6
7  # Accept incoming connections, label SYN packets, and copy
8  # labels to connections.
9  iptables -t mangle -A INPUT -p tcp --dport 21 -m state --state NEW \
10               -j SEL_FTPD
11 iptables -t mangle -A SEL_FTPD -j SECMARK \
12               --selctx system_u:object_r:ftpd_packet_t:s0
13 iptables -t mangle -A SEL_FTPD -j CONNSECMARK --save
14 iptables -t mangle -A SEL_FTPD -j ACCEPT
15
16 # Common rules which copy connection labels to established
17 # and related packets.
18 iptables -t mangle -A INPUT -m state --state ESTABLISHED,RELATED \
19               -j CONNSECMARK --restore
20 iptables -t mangle -A OUTPUT -m state --state ESTABLISHED,RELATED \
21               -j CONNSECMARK --restore
```

The content of this script is as follows. Line 2 ensures that the kernel module for ftp connection tracking is loaded. Line 5 creates a new iptable chain in which the

[11]This example is reproduced from an article on James Morris's blog, available at `http://james-morris.livejournal.com/11010.html`

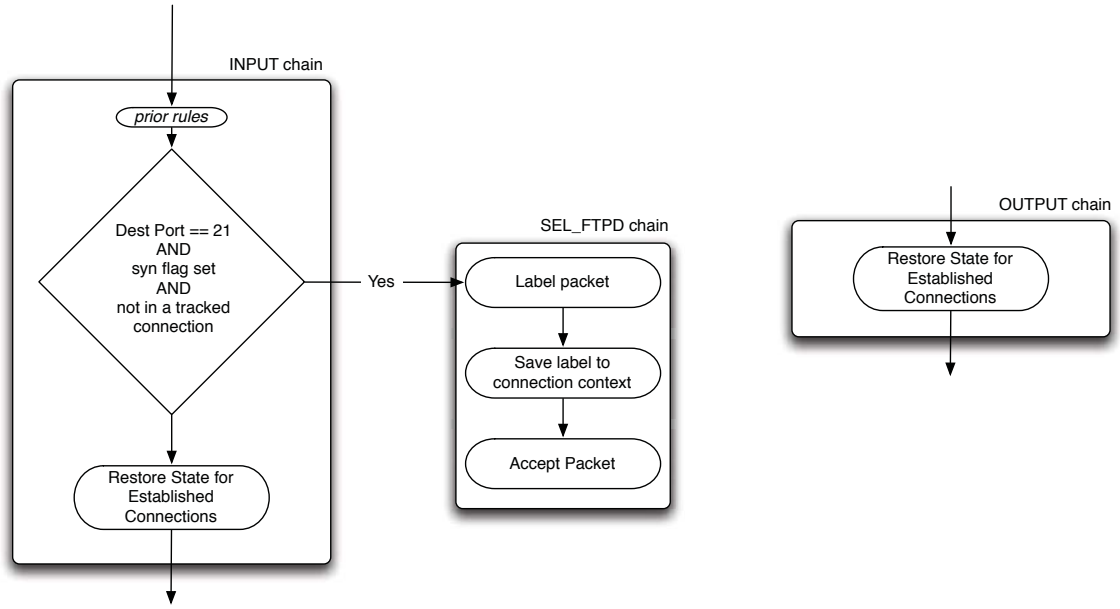


Figure 4.6.: Graphical representation of the structure of the rules from the SECMARK/ CONNSECMARK example.

rules for the labeling of FTP packets will be grouped (`SEL_FTPD`). Line 9 adds a rule in the mangle chain of the input table that redirects the processing of FTP packets (selected by the value of their destination port) for which only the `syn` flag is set to that new chain, and instructs the connection tracking module to consider this packet as creating a new connection (this is indeed the beginning of a new connection if only the `syn` flag is set).

Line 11, a label is applied to the new packet. Line 13, that labeled is saved in the connection context. Line 14, the packet is accepted. Line 18 and 20, which are only reached if the rule on line 9 does not match, are used to label FTP packets of ongoing FTP connections, based on the label that was applied to the initial packet of each connection and save in the connection state (Line 9 to detect the initial packet, Line 11 to label the initial packet, and Line 13 to store the labeling to the connection state).

The structure of this set of rules (illustrated in Figure 4.6) assumes that the default policy of the INPUT and OUTPUT chains is to accept packets. Otherwise, it would

not make much sense to accepts only the first packet of a connection (Line 14), and then reject all the following packets of that connection. Instead of singling out the first packet of a connection as the only one accepted, the purpose of the accept rule in Line 14 is actually to provide an early exit of the firewall chain, to optimise the filtering time.

This example assumes an accept-by-default policy, whereas secure configuration examples of firewall rules usually put a strong emphasis on using deny-by-default policy structures [100]. Accept-by-default is actually a reasonable choice in the case of SELinux, since processes are denied by default access to network packets.

For the FTP daemon to have access to the ftp packets, the following access vector rule needs to be added to the TE policy:

```
allow ftpd_t ftpd_packet_t:packet { recv send };
```

The `recv` permission is checked when the process tries to get the data from the socket, which triggers a read of the socket buffer, by hooking into the `socket_sock_recv_skb()` hook. The `send` permission is checked by hooking into the output access control hook of netfilter: `NF_INET_POST_ROUTING`.

Now that we have introduced the features available in SELinux to perform network packet labeling, and enforce access control based on these labels, we return to the problem of supporting overlay labels for network packets.

4.3.4 Network Packets Overlay Labeling

In this section we study the problem of overlay network packet labeling. That is, how to attach several labels on network packets, without altering the semantics of the existing firewalling rules of the system. We start by defining the problem, show its ties to known NP-hard problems, and provide an approximate solution.

Problem Definition

The overlay labeling of packets that we want to support is a variant of the packet classification problem. The common part of the two problems involves reaching a decision about a network packet based on the content of (some of) its header fields. More formally, the common part of these problems can be stated as follows.

Each packet has d header fields: $f_1, \ldots, f_d$, with their respective values $v_1, \ldots, v_d$. The binary representation of each of these fields has a length l_i (with $1 \leq i \leq d$), hence the following relation for these fields: $\forall i \in 1 \ldots d, 0 \leq v_i \leq 2^{l_i} - 1$

There is a set L of labels that can be attached to packets.

A packet classification rule is a $(d+1)$-tuple of the form $(r_1, \ldots, r_d, l)$, where, $r_1, \ldots, r_d$ are respectively ranges on the $f_1, \ldots, f_d$ header fields ($\forall i \in 1 \ldots d, r_i = (x_i, y_i) s.t. 0 \leq x_i \leq y_i \leq 2^{l_i} - 1$) and $l \in L$ is the label that should be attached to a packet that fits simultaneously in all these ranges. The matching is formally defined as: $\forall i \in 1 \ldots d, x_i \leq v_i \leq y_i$.

A more visual description is the following. The d header fields define a d-dimension space, in which points are packets. Classification rules are d-dimensional axis-parallel boxes; the label can be represented as a color on the box. The classification problem is therefore to determine which box(es) a point is contained in, in order to decide on the label(s) that should be attached to a packet.

Now that we have presented the core of these problems, we can highlight their respective differences. Our problem, the overlay labeling problem, is to find *all* the labeling rules that match a given packet. The packet classification problem is to decide, given a set of *prioritized* packet classification rules, which rule *best* matches the incoming packet. This difference has consequences. For instance, the complexity analysis has to take into account the size of the set of rules that are found to match the incoming packet. Indeed, for n classification rules there may be $O(n)$ rules that match a given packet, in degenerate cases with a thick overlap of rules.

For a practical deployment, we actually have to support a mixture of packet classification and overlay labeling rules. Indeed, existing system labeling rules are

defined in terms of packet classification, with the first matching rule having a higher priority.

Simple Encoding atop `iptables`, and its Limitation

As we just above, there is a difference between packet classification and overlay labeling. What `iptables` does is packet classification. The rules are evaluated in order, and the first rule that matches is applied. Thiscan be worked around by separating labeling rules from filtering rules and placing labeling rules first. This way, the packet filtering decisions are evaluated after all the labeling decisions have been applied. Another limitation to address is that SECMARK, only support attaching one label per packet, so only the label corresponding to the last matching labeling rule is applied to a packet. This limitation of one label per packet can be overcome by using indirection techniques similar to what we developed to support the overlay labeling of filesystem objects (see Section 4.2.3).

An encoding of network packets overlay labeling atop existing `iptables` rules can be constructed, as illustrated in Figure 4.7. The construction is to some extent similar to overlay labeling for filesystem types. Type promotions is necessary for network packets overlay labeling and is performed ahead of time as well. What's more, the packet labeling rules themselves need to be modified, such that they can be extended to support the later deployment of overlay labels. This modification consists in replacing each individual labeling rule with a chain whose last (default) rule performs the same labeling (step 0 in Figure 4.7).

With this modification, an overlay label can later be deployed by inserting a rule for the the overlay labeling in each chain, and by adding another rule whose criteria is the overlay criteria, and whose target jumps to a chain whose default behavior is to label packets with the overlay label. Step 1 in Figure 4.7 represents the deployment of an overlay label that specifies that packets matching condition `c3` should be labeled with label `t3`.

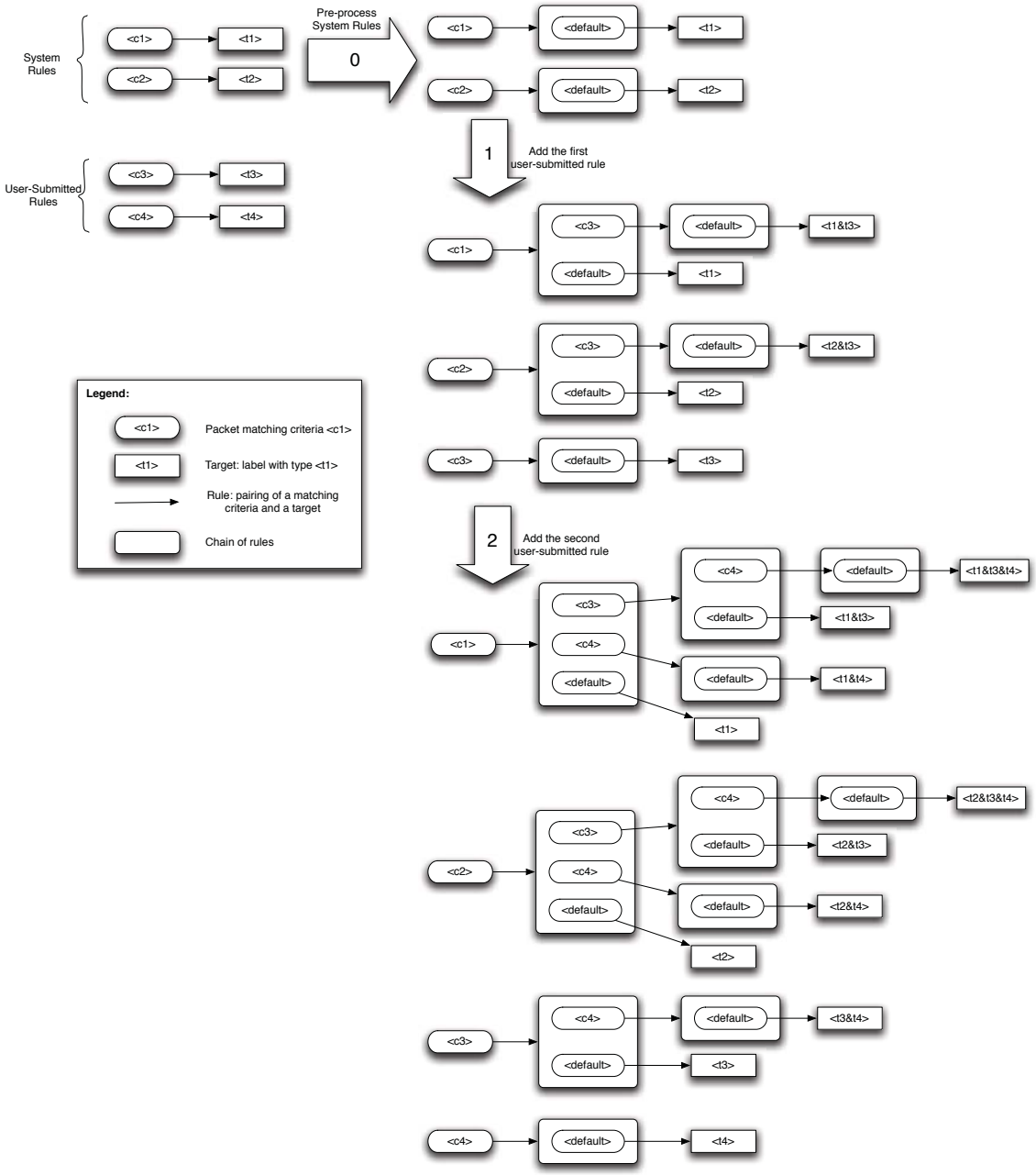


Figure 4.7.: Encoding the overlay labeling of network packets in terms of packet classification rules. This approach has a linear complexity for the runtime processing of packets, as one can verify by looking at the linear increase of the depth of the tree of filtering rules. However, the cost of adding a new rule to the tree is exponential in terms of the number of rules, which prevents the solution from being applicable in practice.

This encoding, however, scales poorly. Supposing there are m system-defined labeling rules and n overlay labeling rules. We see that the addition of the first overlay rule requires the addition of $m+1$ chains and $m+1$ synthetic types, yielding $2m+1$ chains and their associated types. By induction one can show that, with the deployment of n overlay labels, this encoding results in the creation of $(m.2^n + \sum_{i=1}^{n} 2^n)$ labeling chains and synthetic types. Since we expect our system to support at least a few hundred overlay labeling rules, this encoding is impractical.

What this scheme does, essentially, is encode the overlay labeling directly as a decision tree. This encoding can likely be optimized, but the optimization of decision trees is an NP-complete problem [143]. There is another approach to our problem which is tractable, as we show next. This approach, however, will require new packet classification mechanisms.

Solution

The problem of classifying a packet with multiple labels according to n d-dimensional criteria can be solved using techniques from computational geometry. There are several similar problems in computational geometry, e.g. windowing queries, range queries, and stabbing queries. Mapping our problem in terms of a stabbing query in d dimensions yields an efficient and scalable solution, which we present below. Besides the seminal articles that introduced the data structures mentioned and referenced below, we want to acknowledge the following materials as instrumental in helping us piece together this solution. We have used lecture materials from Antoine Vigneron [144], "Foundations of Multimensional and Metric Data Structures" by Hanan Samet [145], and"Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry" by Kurt Melhorn [146]. We found the description of interval trees in [147], [148] and [149] to be misleading, as pointed by Hanan Samet [145].

Stabbing queries The problem that we are solving has a direct mapping to the *stabbing problem*, which is defined as follows. In a d-dimensional space, given a set of n axis-parallel boxes[12] and a query point in that d-dimensional space, find all the k boxes that contain the query point.

Many data-structures have been proposed to support stabbing queries in one dimension: interval tree [148, 150], segment tree [151], priority search tree [152], and interval skip lists [153]. Each of these data structures support stabbing queries in time $O(\log n + k)$, where k is the number of returned intervals. However, they offer a different trade-off between space and time complexity for the storage and update of the rules. We discuss this point later.

These data structures are commonly generalized to $d > 1$ dimensions by recursively nesting them. The idea is the following: the top-level data structure is used to index on one dimension. Each nodes of the top-level data structure points to another data structure that indexes on the $d-1$ remaining dimensions. The nesting ends on the data-structure that indexes the last dimension. The query complexity for this generalization is therefore $O(\log^d x + k)$, as can be proved by induction on the number of dimensions.

Proof of complexity for the multi-dimensional generalization of the stabbing problem

- Base case: with one dimension, the complexity is $O(\log n + k)$, as demonstrated in [148, 150–153]

- Induction: with $d > 1$ dimensions: the longest path that can be traversed in the data structure that represents the d-th dimension is of length $\log n$, since there are n intervals (corresponding to n isothetic boxes) indexed in this dimension. Along that path, for each node that contains stabbed intervals the $d-1$ dimensional data structure attached to the node is queried. So there can be

[12]Formally, axis-parallel boxes are called isothetic boxes

at most $logn$ such queries. Hence a total number of visited nodes of at most $O((\log n).(\log^{d-1} n)) = O(\log^d n)$. Since there can up to n isothetic boxes found along this path, the total complexity is therefore $O(\log^d n + k)$, with $k \leq n$

We now present the data structures available for efficiently performing stabbing queries and justify our choice.

Choice of a Supporting Data Structure Our criteria for selecting a base data structure to support the stabbing queries were the following, by order of importance

1. Query time: for psychological acceptability, it is important that introducing our mechanisms does not degrade the performance of the system too much.

2. Storage space: the data structures used for labeling packets should fit in memory.

3. Update time: it is desirable to use a dynamic data structure in order to minimize the cost of updates. In other words, we would like to avoid rebuilding the data structure from scratch after each update. There are general techniques to turn static data structures into dynamic ones [154]; not all of them are practically implementable. The rationale for this criteria is that, although changes to the security policy are not that frequent, it is still better if they can be effected quickly to avoid disrupting network traffic too much.

Similarly to the packet classification vs. packet overlay labeling issues, data structures that support efficient queries on intervals (or ranges, or segments) overlaps do not all interpret the problem the same way. Some structures are more efficient than others depending on whether one wants to determine if there is an overlap, how many overlaps there are, any first found overlap, the first found overlap with priorities, or all the overlaps. We are interested in the later problem: finding all the overlay labeling rules that match a given packet.

In Table 4.1, we have summarized the performance of computational geometry datastructures that we have surveyed for solving this problem. We have chosen the interval trees created by Edelsbrunner [150].

Table 4.1: Comparison of the time and space complexity of datastructures that support stabbing queries on intervals, where a stabbing query is defined as returning all the intervals that contain the stabbing point.

Datastructure	Query Time	Storage Space	Update Time
Segment Tree [151]	$O(\log n + k)$	$O(n^2)$	$O(\log n)$
Interval Tree [148]	$O(min(n, k \log n))$	$O(n)$	$O(\log n)$
Interval Tree [150]	$O(\log n + k)$	$O(n)$	$O(\log n)$
Priority Tree [152]	$O(\log n + k)$	$O(n)$	N/A
Interval Skip List [153]	$O(\log n + k)$	$O(n)$	$O(\log n)$

4.3.5 Interval Trees to Support Network Packets Overlay Labeling

In this section, we describe interval trees (as formulated by Edeslbrunner [150]), the datastructure that we have chosen to support the overlay labeling of network packets. First, we give a high level decription of the construction of interval trees, which we use as a basis to present how the stabbing query works atop an interval tree, in details. Then, we revisit the construction of the interval tree in details. Finally, we present a generalization of this datastructure to multiple dimensions.

The construction of an interval tree consists in recursively chosing a median point to divide the set of intervals in three sets: 1/ those that overlap the median point, 2/ those to the left of it, and 3/ those to the right of it. The intervals from the first set are attached to the current node. The intervals to the left (resp. right) of the median point are split in a similar fashion in the left (resp. right) sub-tree. As a result of this divide-and-conquer approach on n intervals, the height of the tree is at most $\log n$. As each interval is stored exacty once in exactly one node of the tree, the storage space requirement is $O(n)$.

Assuming the interval tree is already constructed, we now explain how a stabbing query is evaluated and the complexity of this evaluation.

Search

The search proceeds as follows. The stabbing point p_s is compared with the median point p_m of the current node. Three cases are possible:

- $p_s = p_m$: this is the simplest case: all the intervals attached to the current node are stabbed by p_s. All the intervals attached to the current node are returned; the search is over.

- $p_s < p_m$: stabbed intervals have to be searched for in two sets: the intervals attached to the current node, and the intervals stored in the left subtree. To search for stabbed intervals in the current node, we can exploit the fact that all the intervals attached to the current node end after the stabbing point. This is true because these intervals are stabbed by the median point, p_m (and $p_s < p_m$). If these intervals start before the stabbing point p_s, then they are stabbed by it. By keeping a list of the attached intervals sorted, in increasing order, by their starting point, the test works as follows. The current point p_i in this list is compared to p_s. If $p_i \leq p_s$, then the corresponding interval is stabbed by p_s. This continues until a point p_i is found that satisfies $p_i > p_s$. The cost of this test is $O(1)$ if no interval is stabbed; otherwise, it is $O(k)$, where k is the number of stabbed intervals.

- $p_s > p_m$: this case is similar to the $p_s < p_m$ case, except that a list of intervals, sorted by their enpoints in decreasing order, is used to find the stabbed intervals in the current node.

From this description of the search, one can infer that a given search traverses at most $\log n$ nodes. On each node, the test for stabbed intervals either takes $O(1)$ time when it fails, or $O(k)$ when k intervals are stabbed. Consequently the complexity of

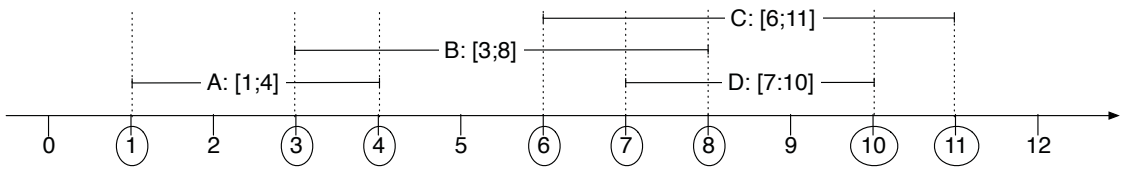


Figure 4.8.: Example of one-dimensional intervals that are used to illustrate the construction of an interval tree. The are three intervals: A: [1;4], B: [3;8], C: [6;11], and D: [7;10]. On the x axis, the circled coordinates indicate the values that correspond to a least one interval endpoint.

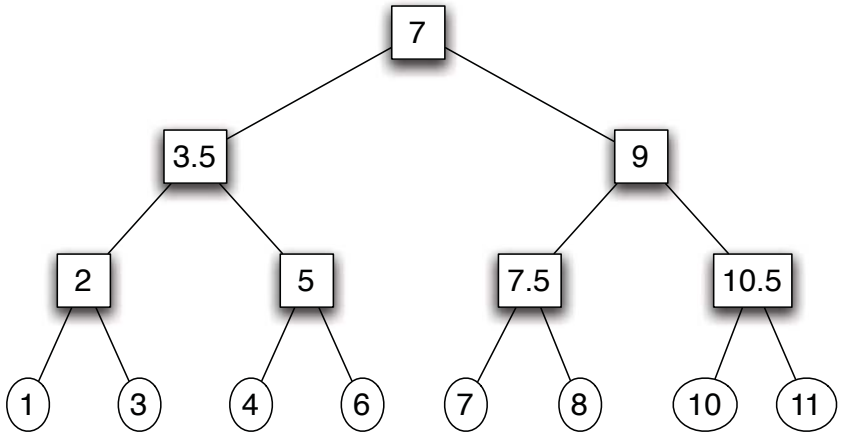


Figure 4.9.: First step in the construction of an interval tree. A binary search tree is built based on the endpoints of the intervals from Figure 4.8.

a stabbing query in the interval tree is $O(\log n + k)$, with k being the total number of intervals returned by the stabbing query.

It is possible to optimize the search in an interval tree by keeping track of the active nodes of the tree. Active nodes are the ones where intervals are stored. By adding an extra set of pointers to the nodes of an interval tree, it is possible to reduce the time taken by the search by limiting the search to explore only active nodes. In the following subsection, we presented how an interval tree can be constructed. After this presentation, we show how the marking of active nodes and the setting of their pointers to active nodes can be done, while inserting intervals into the interval tree, and without degrading the complexity of interval insertion.

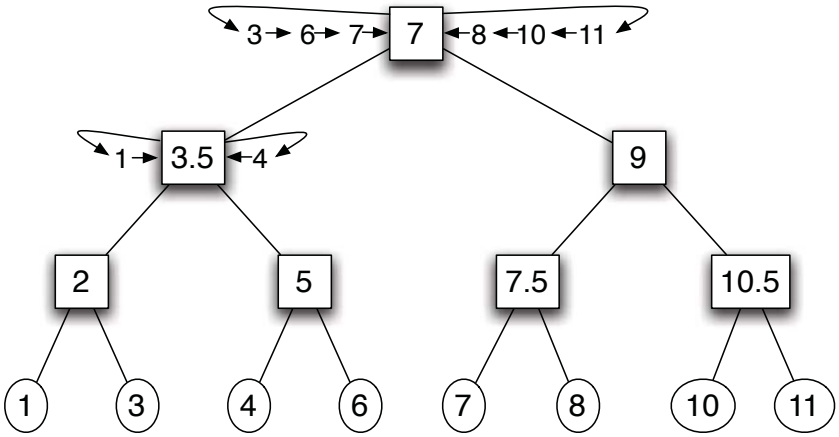


Figure 4.10.: Interval tree built based on the endpoints of the intervals from Figure 4.8.

Construction

We now explain how an interval tree is constructed, and we illustrate this construction based on the example of Figure 4.8. The first step in the construction of an interval tree involves sorting the endpoints of the intervals (the circled nodes in Figure 4.8). Based on this list, a binary search tree that contains the interval endpoints as leave nodes can be constructed; the key value of inner nodes can picked anywhere between the two values of their child nodes. It is customary to use the median value. We have done mostly so in the example (see Figure 4.9). The next step consists in inserting the intervals one by one in the search tree. Intervals get attached to the first encountered node whose key is comprised between the endpoints of the interval. For instance, consider interval A. interval A's endpoints (1 and 4) are both lower values than the tree's root key, so the insertion continues in the left subtree of the root, whose key value is 3.5. This value is contained in A. As a result, A is attached to that node. This requires adding A's beginning point (1) to the list of beginning points sorted in increasing order, and adding A's ending point (4) to the list of ending points sorted in decreasing order. The result of inserting intervals from Figure 4.8 into the tree from Figure 4.9 are illustrated in Figure 4.10.

The cost of the construction of a one-dimensional interval tree is therefore composed of the following elements:

- the cost of constructing the binary sort tree, which is the cost of sorting the endpoints of the intervals: $O(n \log n)$
- the cost of locating the node where each interval must be stored. For each interval, this is the cost of a search in the binary search tree, $O(\log n)$. Cumulatively, the cost is $O(n \log n)$.
- the cost of inserting the endpoints of an interval at the node where the interval is stored. If a list is used the cumulative cost is $O(k^2)$, where k is the upper bound on the number of intervals that can end up being stored on a given node.

This is not acceptable as the overlapping of segments can be arbitrarily thick and therefore k can be comparable to n. The solution is to use a sort tree to index the elements of the lists containing the endpoints, as illustrated in figure 4.11. With this trick, the cumulative cost of storing the endpoints of an interval in a node is reduced to $O(k \log k)$, while still preserving the time complexity of the stabbing queries (the linked list can be used to return the attached intervals in $O(k)$).

Consequently, the cost of constructing the interval tree is $O(n \log n)$.

An additional feature of interval trees, in their most optimized form, is to mark nodes that are active and maintain a doubly-linked list of active nodes. Nodes are considered active if they have at least one interval attached to them. This optimization allows faster queries in practice on a sparsely populated interval tree ; the theoretical bound remains unchanged. As noted by Melhorn in [146], if the set of base nodes on which the interval tree is built contains only endpoints of the intervals stored in the tree, "then the mark bits and the doubly linked list is [sic] not needed." Since we plan on building the interval tree based on the endpoints of the packet selection criteria, this remark applies and an implementation of overlay labeling for network packets does not need to implement this optimization. Overlay labeling, however, use multi-dimensional criteria. We present the generalization from one to several dimensions in the next section.

Generalization to Higher Dimensions

Interval trees can be generalized, from storing 1-dimensional intervals, to storing d-dimensional intervals (isothetic boxes). This is possible because the d-dimensional stabbing query is *decomposable* as a search problem on each of the dimensions, the final answer being the join of the answers on each dimension.

Consequently, it is tempting to implement a straightforward decomposition of the storage and the search for intervals in d separate 1-dimensional interval trees. The

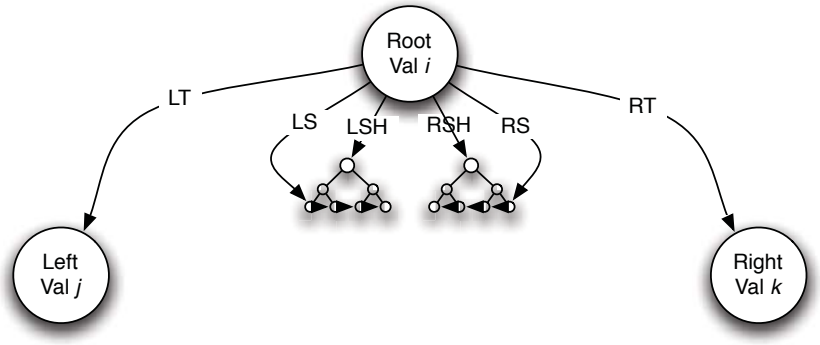


Figure 4.11.: Representation of one non-terminal node of a one-dimensional interval tree, which has two non-terminal children nodes. The node contains 3 sets of pointers. LT and RT point to the children nodes in the primary structure, and hence the heads of the left and right subtrees in the primary structure. LSH and RSH point to the heads of the left and right search trees in the secondary datastructure, where the endpoints of intervals that contain value i are stored. These pointers are used when inserting or deleting intervals from the interval tree. LS and RS respectively point to the leftmost interval starting point in LSH and to the rightmost interval ending point in RSH. These additional pointers are used to accelerate the lookup of interssected intervals.

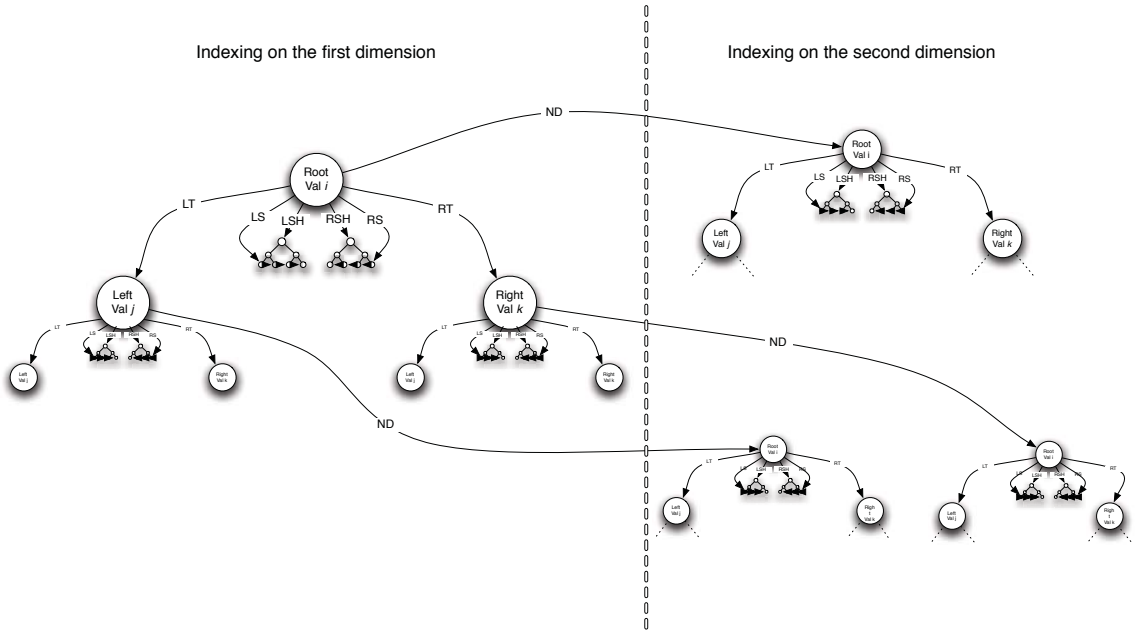


Figure 4.12.: Generalization of interval trees, from one to two dimensions. In addition to the three sets of pointers described in Figure 4.11 , each node of the primary structure also contains a pointer ND ("next dimension") to an interval tree that indexes, according to the other dimension, the intervals attached to that node.

interval trees get queried separately, and the final result is assembled by keeping only the intervals that are returned in every query. This would result in simpler code and simpler datastructures than nesting the datastructures (which we illustrate in Figure 4.12). Also, it would seem to offer a great time complexity: $d.(O(k) + \log n)$. However, degenerate cases can force this solution to run in $O(n)$, when any of the 1-dimensional queries returns a number of elements comparable to n.

For instance, if all the overlay labeling criteria specify traffic coming from the same network interface, then each packet will be matched on that dimension, forcing the labeling to run in $O(n)$, even when the other criteria do not overlap at all. Because of the possibility of such degenerate cases, a simple join strategy is not adequate. The nesting of interval trees, on the other hand, guarantees that only the relevant intervals will be looked at.

We present a 2-dimensional nesting (see Figure 4.12); it generalizes easily to more dimensions. With a 2-dimensional interval tree, the main tree is built based on the endpoints of intervals in the first dimension. The nodes of the main tree, instead of pointing directly to intervals, point to a 1-dimensional interval tree. In this tree, the interval endpoints are sorted and stored according to the second dimension. Also, when intervals are stabbed in this last dimension, it means that they were stabbed in the previous dimension, so they can be returned in the result of the stabbing query.

The direct advantage of this nesting is that intervals are returned if and only if they are stabbed in each and every dimension. This property makes the performance more resilient to degenerate configurations, where $O(n)$ intervals overlap on a given dimension. It is actually possible to construct an adversarial set of rules such that there are *always* $O(n)$ intervals stabbed in each dimension of the query. The construction relies on the fact that each dimension is finite. The idea is to partition each dimension in d intervals, and then use the cartesian product of these intervals to partition the space. This results in 4 squares when $d = 2$, 27 cubes when $d = 3$, and more generally d^d partitions for a given d. Then, the n rules are created as an equal distribution of these hypercubes. The result of this construction is that, on each

dimension, a stabbing query will return n/d (and thus $O(n)$) elements. Therefore, the complexity of the search can be forced to be $O(log(n/d)) + k$ on each dimension. Based on the generalization of the search to multiple dimensions that we presented earlier (see Section 4.3), this means that the multi-dimensional search can be forced to run in $O(log^d(n/d) + k$. In the general case, there is therefore no better solution than the one we proposed (within a constant factor)[13].

4.3.6 Summary

In this section, we have presented the problem of the overlay labeling of network packets. After explaining how the labeling of network packets works on SELinux we have showed that, contrary to filesystem overlay labeling, the existing mechanisms can not be used to support network overlay labels in practice. The problem of overlay labeling of network packets can, however, be mapped to the problem of stabbing queries, from the field of computational geometry. Finally, we have presented a detailed explanation of how a multi-dimensional generalization of interval trees offers a solution that has an optimal complexity for the general case.

4.4 Predicates on Type Attributes

In this chapter, we have presented methods to let the user refine the coupling between the abstract TE policy and the concrete system objects on which the policy is being enforced. In order to support this refinement of the labeling, we have introduced the notion of type promotion, where a type is promoted to being an attribute of a synthetic type. Type promotion changes the type enforcement access vector rules, from being expressed on types, to being expressed on type attributes. The goal of this section is to show how this reasoning on type attributes can be re-used to support expressing the policy at a higher level of specification. Those higher level

[13]The construction used in this proof is due to Prof. Mikhail Atallah.

specifications can in turn be used to either express new higher level properties on the policy, or to refactor existing ones.

To motivate this section, we start by presenting an example of refactoring that can be applied to the SELinux reference policy. There, all types that are to be applied to files are labeled with the attribute `file_type`. Two additional attributes, `security_file_type` and `non_security_file_type`, are then used to distinguish file types that are relevant to the security of the system (e.g. `/etc/shadow`) from the ones that aren't. All accesses to files that are considered security-relevants are systematically audited. Accesses to other files are not necessarily audited.

We find it error-prone to rely on every file type to bear either the `security_file_type` or the `non_security_file_type` attribute. We think that using either a blacklist or a whitelist approach to identify security-relevant files would yield a more assurable policy. Otherwise, the policy has to be analyzed to guarantee that all types that bear the `file_type` attribute either bear the `security_file_type` or the `non_security_file_type` attribute.

A blacklist consists in explicitly listing the files that should not be included in the allowed accesses. A whitelist contains the complement set of the blacklist, files for which accesses are explicitly allowed. Using either a blacklist or a whitelist approach guarantees that every file type, whether it is considered security relevant or not, will belong to exactly one of the two sets. This contrasts with the current mechanism, where checking must be performed on the policy to guarantee the same property. Indeed, a file type that does not bear either the `security_file_type` or the `non_security_file_type` security attribute will not be considered to belong to either category. Using a blacklist or a whitelist approach will guarantee that the file belongs to at least and at most one of the categories.

To support blacklists and whitelists in a direct manner, we need a means of representing and reasoning on the fact that a type does not possess a given attribute. To continue our example, this consists in using either the `security_file_type` attribute and the expression of its absence

(`!security_file_type`), or the `non_security_file_type` attribute and the expression of its absence (`!non_security_file_type`).

Conjunction and disjunction operators would also provide a useful language construct for refactoring the policy. For instance, the `security_file_type` could be replaced by a more generic `security_relevant` attribute. Then, the files that need systematic auditing would be specified by the predicate (`security_relevant AND file_type`). The rest of this section presents the syntax, semantics, and design of system support for type attribute predicates.

4.4.1 Predicates

Now that we have illustrated why we think that type attribute predicates are a useful extension, we present their syntax and how they can be integrated. The description of the system integration in this section rests on simplifying assumptions: the policy is static and predicates are not allowed to reference one another. We discuss how to handle dynamic changes of the policy in the next section. Avoiding infinite evaluation loops that can arise when predicates can reference one another is discussed in the subsequent section.

Language and Interpretation The language of type attribute predicates is structured as follows. Type attribute predicates are propositions, from propositional calculus. The atomic formulas are the type attributes. More complex formulas are formed by connecting these atomic formulas with the conjunction or disjunction logical operators, or by prefixing the formulas with the negation operator.

For a given type, the interpretation of a predicate is simple: each of the atomic formulas evaluates to true if the corresponding attribute is attached to the type; it evaluates to false otherwise. The interpretation of the logical connectors is the standard one.

Deployment We now describe how type attribute predicates can be deployed atop a security policy, that we assume static for now. As we have showed for filesystem and network overlay labeling, it is possible to use the support of type attributes to encode additional labeling schemes, provided the original types have been promoted. The same technique is used to support type attribute predicates.

The idea is to first create a type attribute that will be used to represent the fact that the predicate is satisfied. Then, for each existing type, the system evaluates if the attributes that are attached to it are such that the predicate is satisfied. If so, the attribute that represents the type attribute predicate, the *materializing attribute*, is attached to that type. These modifications are confined to the policy; no object needs to be relabeled.

In the next two section, we describe how this simple scheme is extended to support policy dynamics and to efficiently evaluate the predicates without entering infinite evaluation loops.

4.4.2 Handling Policy Dynamics

We have presented how, given a *static* policy, predicates can be used to define overlay labels. A question that naturally arises at this point is whether and how this kind of overlay label can support a dynamic policy. For instance, what should be done when a type is added to the policy ? More precisely, which predicate overlay labels should be associated with this type upon its creation ? Also, which actions should be taken when the attributes of a type are modified, by either adding or removing attributes to the type ?

Essentially, our system needs to support triggers on the policy database, so that any modifications to the policy types and attributes is immediately accompanied by the appropriate updates to predicates. For each modification, the companion update is as follows.

Type Creation: A type can be created in two ways: either directly, or through a type attribute. As a consequence, the type has either one or no type attribute attached to it upon creation. The system looks up the existing predicates to figure out which ones apply to the new type. The search is narrowed by using the fact only predicates based on negative clauses or based on the type attribute attached to the type can be satisfied. For each predicate that matches the new type, the corresponding type attribute must be attached to the type. Because our system supports the negation operator, it could potentially enter an infinite evaluation loop. We show in section 4.4.3 how we address this problem.

Attribute Creation: Creating an attribute has no consequence from the perspective of predicates. Predicates can only refer to existing type attributes. Consequently, there can not be a predicate refering to a type attribute before this attribute is created.

Attribute Attachment: When an attribute is attached to a type, the system needs to evaluate only the predicates that reference that attribute. For each of these predicates, one of three cases can happen:

- the additional attribute enables the satisfaction of the predicate, which was not previously statisfied. In that case, the corresponding type attribute must be attached to the type.
- the additional attribute invalidates the satisfaction of the predicate, which was previously statisfied. This case is only possible with predicates that use the negation operator. In that case, the corresponding type attribute must be detached from the type.
- the additional attribute has no impact on the satisfaction of the predicate. This can happen for instance when the disjunction operator is used; in this case no additional action needs to be taken by the system.

Attribute Detachment: When an attribute is detached from a type, the required operations are the reverse of the ones carried out for attribute attachment. For each predicate that references the attribute being detached, one of three cases can happen:

- the removal of the attribute invalidates the satisfaction of the predicate, which was previously statisfied. In that case, the corresponding type attribute must be attached to the type.
- the removal of the attribute enables the satisfaction of the predicate, which was previously unstatisfied. This case is only possible with predicates that use the negation operator. In that case, the corresponding type attribute must be attached to the type.
- the additional attribute has no impact on the satisfaction of the predicate. This can happen for instance when the disjunction operator is used; in this case no additional action needs to be taken by the system.

Attribute Deletion: When an attribute is deleted, it must first be detached from each type it was attached to. Detaching an attribute from a type is described above.

4.4.3 Runtime Support

In order to efficiently support predicates on type attributes, the system needs to be extended. The system needs to maintain, for each type attribute, a list of the predicates in which it is referenced. This is required to support efficient attribute attachment and detachment, as explained in the previous section.

Predicates on type attributes can lead to a serious evaluation problem if predicates are allowed to reference the type attributes that materialize predicates: the evaluation may not terminate. Consider the following two predicates:

```
A := !B
B := !A
```

By referencing one another, these predicates create an infinite evaluation loop. To prevent this problem, predicates are prohibited from refering to type attributes that are used to represent predicates. Therefore, the system needs to maintain an additional flag for each type attribute, indicating whether it is used as a marker for a predicate. When a user attempts to create a predicate, the system can use this flag to ensure that the predicate does not reference other predicates.

This problem (introducing loops through the usage of negation) is well know in logic programming. There are two main ways of solving it. The first solution is to give up the negation operator, which we are not willing to do. The second solution is to use stratification [155]. The principle of stratified logic is to assign a rank to each predicate, with the following constraints:

- A predicate must have a rank superior or equal to the rank of each predicate that it references positively.
- A predicate must have a rank strictly superior to the rank of each of the predicates that it references negatively.

Effectively, stratification prevents the use of negation to build circular dependencies among predicates. The design we have presented does not implement a stratified predicate evaluation strategy. It does, however, prevent the formation of these circular dependencies and offer the negation operator.

4.4.4 Motivating Example Revisited

We now revisit our motivating example and show how type attribute predicates can be used to write simple yet expressive access control rules.

In Section 4.2.3, we presented filesystem overlay labeling. It enables users that do not have the administrative right to relabel some file to single out a set of files in order to treat them separately for access control. In the introduction of this chapter, we considered a property like the following: “This domain has access to system binaries, except compilers.”

This property can be expressed simply with predicates. Provided that the compiler is labeled, say, `compiler_t`, the group of files that are "system binaries, except compilers" can be concisely designated by the predicate (`bin_t AND !compiler_t`).

On a standard SELinux, even with filesystem overlay labels, this property would be cumbersome to express: in TE, there are no negative permissions[14]. As a result, the user would have to explicitly designate all the system binaries, except the compilers, with an overlay label. While this is effectively how the above predicate will be deployed on the system, predicates shift the burden from the user to the system.

In essence, the predicates support the specification of groups of objects in terms of set operations: intersection, union, and complement. This flexible means of designating object can ease both the refactoring of the existing policy and the expression of new properties.

4.5 Policy Dynamics: Avoiding Subversion

This chapter has discussed several use cases of overlay labeling, and how to efficiently support these use cases. For filesystem and network overlay labeling, the motivation was to support refinements of the coupling between the abstract TE policy and the concrete set of objects managed by the system on which the policy is deployed. For the designation of objects based on type attribute predicates, the goal was to enable a higher-level specification of the policy.

Overlay labeling, however useful, can be dangerous if not properly regulated. Indeed, a simple implementation of overlay labeling could allow a user to grant permissions, to himself or other users, that he would never be entitled to grant without overlay labeling. The gist of the problem is the following. A (malicious) user defines an overlay label, and then proceeds to give away access rights to object bearing this label. Suppose that the overlay label happens to cover types for which the user does not have the administrative authority to grant access. The user could try to work

[14]SELinux has a notion of explicit deny, with the `neverallow` access vector rules. The rules are enforced at compile time only, to prevent errors in the written policy.

around this restriction by adding rules to the policy that grant access, expressed in terms of the overlay label instead of the base type.

This problem has to be dealt with under two angles. By restricting the span of types that user-defined overlays can cover, so that overlays defined by a given user can only cover types that he has *some* privileges on, one can prevent a user from defining overlays that could only be used in attempts to subvert the policy. By restricting the content of rules that can be formulated based on these overlays, one can prevent a user from adding rules (that refer to overlay labels) which subvert the policy.

The remaining of this section is organized as follows. We first define which invariants we want to preserve on the policy. Then we present the infrastructure required to keep track of policy dynamics, in order to preserve policy invariants and hence avoid subversion. Finally, we present the enforcement of these invariants.

4.5.1 Policy Properties

The properties that we want to preserve correspond to the notion that the intent of the base system policy must be preserved. More precisely, the system-defined type transition and access vector rules must have precedence on the user-defined ones. Administrative users can modify the system policy according to the administrative privileges that are granted to them. Regular users, however, should be prevented from modifying either the TE policy or its coupling with the underlying system in a way that violates the system policy.

This means that regular users can not grant permissions they do not possess themselves. Users that are extended administrative privileges can additionally perform the operations that require these privileges (including granting permissions, if applicable), but no more than that. Below, we introduce definitions that are then used to formally define the invariants that should be preserved.

Definition 4.5.1 (Access contour of a domain) *A domain's access contour is the set of all non-administrative permissions granted to a domain. This contour can be*

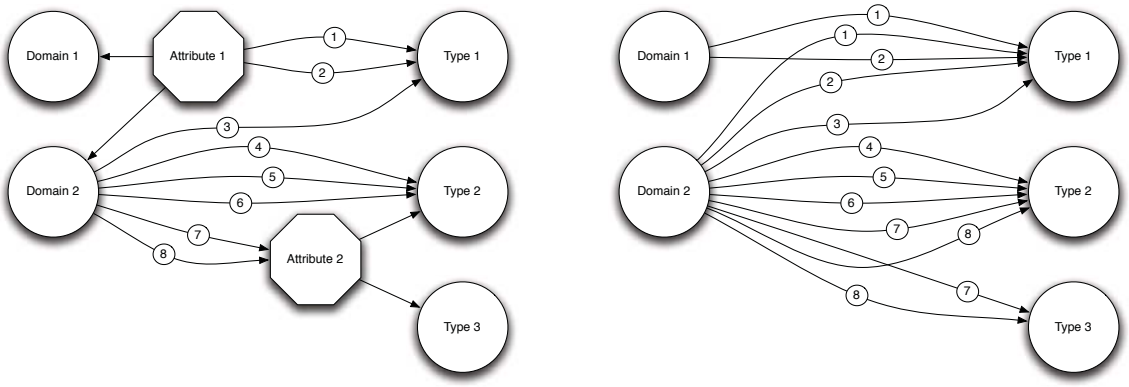


Figure 4.13.: Example permission graph showing how attributes are used to factor the policy. The left part of the figure is a representation of how attributes can be used to factor the expression of permissions given to domains. The right part is a representation of the permissions effectively granted. Attribute 1 is used to factor the assignment of Permission 1 and Permission 2 on Type 1 to Domain 1 and Domain 2, by grouping the domains under the same attribute. Attribute 2 is used to factor the assignment of Permission 7 and Permission 8 on Type 2 and Type 3 by grouping Type 2 and Type 3 under the same attribute. These two usages of attributes can be combined.

defined in simple graph terms. For each type that a domain has access to, and for each permission that the domain has on that type, there is an edge from the domain to the type.

As illustrated in the first graph of Figure 4.13, attributes are commonly used to factor the policy. These attributes are expanded according to Algorithm 3 to yield the access contour graph, the second graph in Figure 4.13.

Definition 4.5.2 (Access contour inclusion) *The access contour of one domain d_1 is included in the access contour of domain d_2 if, for each permission that d_1 possesses, d_2 possesses the same permission. In graph terms, this means that if we replace d_1 by d_2 in d_1's access graph, the resulting graph is a sub-graph of d_2's access graph.*

For example, in Figure 4.13, the access contour of Domain 1 is included in the access contour of Domain 2.

Algorithm 3 Expansion of type attributes, to obtain the access contour of a domain $\mathtt{d}$

$contour \leftarrow \varnothing$
{Expansion of domain attributes}
$source\_labels \leftarrow \{\mathtt{d}\}$
for all $attr_i \in \mathtt{d}.attributes$ **do**
  $source\_labels \leftarrow source\_labels \cup \{attr_i\}$
end for
{Expansion of accessible type attributes}
for all $label_i \in source\_labels$ **do**
  for all $a(label_i, \mathtt{t}, \mathtt{c}, \mathtt{o}) \in \Psi$ **do**
    if $\mathtt{t} \in policy.attributes$ **then** {Expand attributes if needed}
      for all $type_i \in \mathtt{t}.types$ **do**
        $contour \leftarrow contour \cup a(\mathtt{d}, type_i, \mathtt{c}, \mathtt{o})$
      end for
    else
      $contour \leftarrow contour \cup a(\mathtt{d}, \mathtt{t}, \mathtt{c}, \mathtt{o})$
    end if
  end for
end for

Based on these definitions, we can reformulate rigorously the definition of the policy invariants that have to be preserved. Regular users can grant permissions, within the access contour of their domain, to domains they have created. These are implicit administrative permissions granted to users in order to let them segregate applications that run on their behalf. Any attempt at granting permissions that does not satisfy this contraint must be authorized by an explicit administrative permission. If not, it should be rejected.

4.5.2 Taming Indirect Constructs: Preserving Policy Invariants

When a user attempts to add a permission, the system needs to determine whether the user is employing his implicit right to partition his user account, or if the user is actually attempting to perform an administrative operation that must be explicitly authorized.

Since our extension with overlay labels adds another level of indirection, additional controls have to be performed when evaluating the request by a user to add a permission to the policy. Indeed, overlay labeling requires the promotion of base types to attributes and the creation of synthetic types. These operations, if not tracked, tend to obufscate the permissions specified in the original security policy.

Type promotion (see Section 4.2.3) replaces a type by a synthetic type, to which all attributes of the original type get attached, and a new attribute that represents the promoted type. This attribute is also attached to the synthetic type. When type promotion is performed, the permissions are adjusted to preserve the semantics of the original policy, while reflecting the promotion. This adjustment was described in Algorithm 1.

What we are concerned with here is to guarantee that, although the system extends *implicit* administrative permissions to its users, the original policy intent can not be modified without exerting *explicit* administrative permissions.

Overlay labels, since they are encoded as type attributes, can be analyzed using the same notion of access contour as defined above. A user is implicitly allowed to

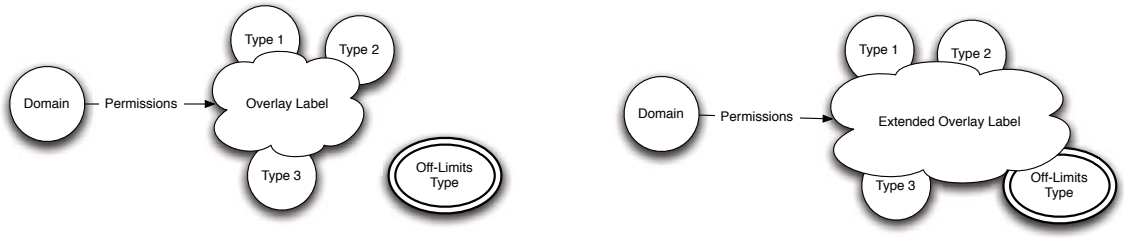


Figure 4.14.: Illegal extension of an overlay label. The overlay is extended to cover a type on which the user that extends the overlay has neither direct access nor administrative rights. This implicit extension of rights must be prevented by the system.

create a permission to a given overlay label, if this permission fits in the user's access contour. The addition of any permission that does not fit in the user's access contour must be explicitly authorized by an administrative permission.

It would be desirable to let users redefine overlay labels, as opposed to just creation and deletion of the overlays. This way, the rules that refer to the overlay label would not have to be first dropped, and then re-created every time the overlay label is redefined. However, care should be taken to drop rules that, after expansion of the overlay label, result in permissions that overlap the access contour. This case is illustrated in Figure 4.14.

This feature, however, is better left out of the core system. It is simpler to support only creation and deletion of overlay labels, with a deletion of all the rules that refer to an overlay when the overlay is deleted. This way, permissions that would become illegal as a result of widening the overlay are simply rejected when they are added back to the overlay label. While the overlay is being expanded, permissions that were referring to the overlay can be cached by a helper program. It would help such a program if the administrative interface was transactional. With a transactional interface, the transaction corresponding to a rejected redefinition of an overlay label would simply be rolled back.

4.6 Reversibility

Finally, we address reversibility, which is an important factor of psychological acceptability for administrative models [77]. Type promotions and overlay labeling are not naturally reversible operations. To support reversibility of these operations, the system must actually keep a record of them, so that they can be undone. For instance, once a type has been promoted, it simply becomes an attribute of a synthetic type. At this point, there is no way to tell this attribute apart from the other type attributes of the original type, which are also attached to the synthetic type as a result of type promotion.

In this section, we present how type promotions and the deployment of overlay labels can be undone. For each of these operations, we start by presenting the metadata that the system needs to maintain in order to support these undo operations. Then we show how these undo operations can be carried out.

4.6.1 Undoing Type Promotions

Property 4.6.1.1 *To support reversible type promotions, it is necessary and sufficient to keep track of the names of the promoted types.*

Reversing a type promotion requires two operations. First, the type attribute that represents the promoted type needs to be removed from the policy. Second, every policy statement that references the synthetic type that was generated for the type promotion needs to be replaced by a statement that references the name of the original type. These statements include, the type declaration, the attachment of attributes to the type, type transition rules, and access vector rules. Since types and type attributes share the same namespace, the removal of the type attribute must happen before the synthetic type name is replaced by the original type name.

The above explanation shows why it is sufficient to keep track of the names of the promoted types. The necessity of storing this information stems from concerns for correctness. If the system does not verify that the name for which a type demotion

is requested does indeed correspond to a promoted type, then the system could be fouled in performing the above operation with an attribute that does not represent a promoted type. Also, undoing type promotion should only be performed after all the overlay labels have been undone.

4.6.2 Undoing Overlay Labeling

Property 4.6.2.1 *To support reversible overlay labeling, it is necessary and sufficient to keep track of the following information for each overlay label: its name, its deployment criteria (a regular expression for a filesystem overlay label; packet filtering criteria for a network overlay label), and, for each synthetic type that had to be generated, its name and the name of the type it was derived from.*

In our demonstration of this property, we first consider the case of reversing non-overlapping overlay labels. Then, we consider the case of overlapping overlay labels.

Assuming non-overlapping overlay labels, the reversal of an overlay label proceeds as follows. Given the name of the overlay label, we can find all the types that the overlay label is attached to; these types are synthetic. For each of these synthetic types, the attribute that represents the overlay label has to be removed and the objects that bear the synthetic type have to be relabeled to bear their original synthetic type (the one they bore before the overlay label was deployed). This requires two pieces of information: the parenthood relation between synthetic types, in order to know which type to relabel to, and the regular expression that was used to deploy the overlay label, in order to locate the objects that need to be relabeled.

The preceding explanation showed why the recording of synthetic types derivations, as well as the recording of their deployment criteria are necessary conditions. Recording the names of overlay labels is also necessary, for the same reason that it is necessary for safe reversal of type promotions. If the system does not verify that the name for which the removal of an overlay is requested does indeed correspond to a overlay label, then the system could be fouled in performing the above operation

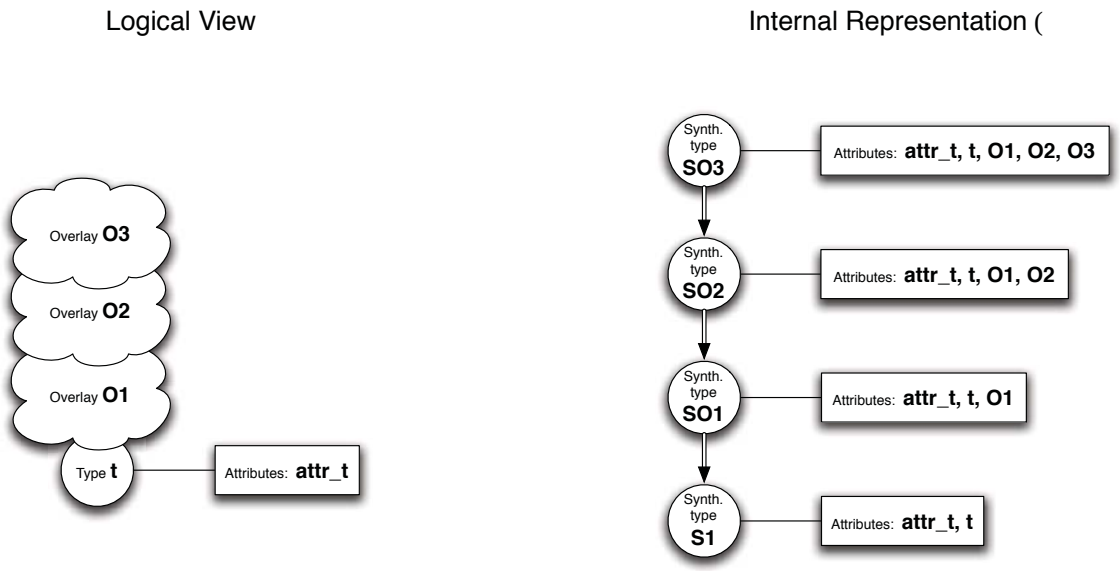


Figure 4.15.: Simple example of overlapping overlay labels: the successive overlays are successive strict refinements of the original type.

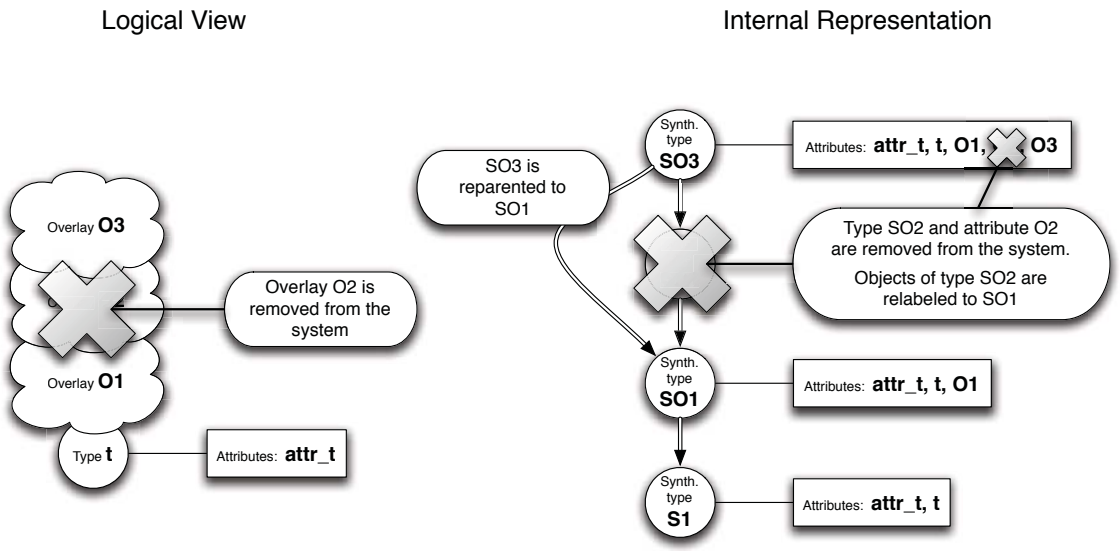


Figure 4.16.: Removing the overlay O3, based on the configuration from Figure 4.15.

with an attribute that does not represent a overlay label.

We now show that this property holds even in the case of overlapping labels. We illustrate this with a simplified case of overlay labeling (see Figure 4.15), where each

successive overlay only refines a single (overlay) type. We show how our solution applies to this simplified case, and how it generalizes to overlays that refine several (overlay) types and to types that are refined by several overlays. In other words, our solution generalizes to the branching cases but we start by considering only the straight line case first.

Given the example configuration from Figure 4.15, suppose that we want to remove the overlay O2 from the system, the questions to answer are the following. For objects that are labeled with the type SO2, which type should they be relabeled to ? How do we find these objects ? The type SO3 was derived from SO2; which type should it be now considered a derivation of ? Which adjustments need to be performed on type attributes ?

The adjustments to the policy, illustrated in figure 4.16 are as follows. Since overlays are internally represented by attaching attributes to types, the attribute that corresponds to the overlay label must be removed from the system. This involves detaching that attributes from all the types that bear it. Also, since the creation of a synthetic type, SO2 in our example, is required to separately label the objects that are covered by the overlay, that type needs to be removed from the system. This operation could be done by iterating over all the types of the policy. However, it can be carried more efficiently if the system keeps a bi-directional track of parenthood relations among synthetic types. In that case, all the types that bear the overlay attributes can be efficiently located, as they are children types of the synthetic type being removed (e.g. SO3 is a child type of SO2). When that type is removed from the system, the objects that bore that type need to be relabeled. They are relabeled to the type they would have had if the overlay had never been created, SO1 in our example.

To summarize, removing the O2 overlay from Figure 4.16 requires using the following information: the name of the attribute that materializes the overlay (O2), the synthetic type that was created to materialize the overlay (SO2), the parent type of that synthetic type (SO1), the child synthetic type (SO3) that what created to label

objects from O2 that are covered by overlay O3, and the criteria that was used to deploy the O2 overlay. In the above, we have shown why how each of these items is necessary to perform one of the actions needed to reverse the deployment of an overlay label. As the reversal of the deployment does not need additional infortmation, these items are also sufficient.

The operation of removing an overlay label from the policy generalizes to overlapping overlays that cover several types and to overlays that are covered by several overlays. This generalization is presented in Algorithm 4.

Algorithm 4 Remove overlay O

```
{Reparent types as needed and remove the overlay attribute.}
for all type_i ∈ O.synth_types do
  for all type_child ∈ type_i.children_types do
    type_child.parent_type ← type_i.parent_type
    remove_attribute_recursively(type_child, O.attribute)
  end for
end for
{Relabel object to the parent type. (only for filesystem overlays)}
for all object_i ∈ locate_objects(O.deployment_criteria) do
  if object_i.typeinO.synth_types then
    object_i.type ← object_i.type.parent_type
  end if
end for
{Remove the synthetic types from the policy.}
for all type_i ∈ O.synth_types do
  policy.types ← policy.types\{type_i}
end for
```

4.7 Conclusion

In this chapter we have motivated the need to let users refine the labeling of objects on a TE system, and presented solutions to address this problem. The need for labeling refinements stems from the need to enforce the principle of least privilege. If regular users are able to create and configure TE domains (using the administrative model from Chapter 3) but can not specify exactly which objects a domain has

access to, due to a coarse labeling, then our work would not address the fine-grained requirement that is part of our thesis statement. The solutions we presented are the following.

Filesystem label overlays are a technique that we designed, based on using type attributes, to let users overlay arbitrary labels on system objects. This technique is itself based on another technique that we designed, called type promotion, which is used to support the later deployment of overlay labels.

Network label overlays are similar to filesystem label overlays, only for network packets. Developing this technique was more involved that for filesystem overlay labels, as a straightforward solution has an exponential space usage, and optimizing that space usage is NP-complete. Our solution relies on applying a datastructure from computational geometry, interval trees. We have proved that the complexity of this solution is optimal within a constant factor in the general case.

Predicate overlay labels are a technique to specify the TE policy at a higher level, based on predicate logic expressed over the attributes attached to types. It can be supported the overlay labeling mechanisms developped for filesystem overlay labels.

We have concluded this chapter by exposing how these overlay labeling techniques can be reversed, and how it is possible to contain the use of overlay labels to prevent them from being misused in order to subvert the policy of a system.

5. EVALUATION

In the two previous chapters, we have presented and motivated our extensions to TE and to the labeling mechanisms offered by SELinux. In this chapter, we evaluate these extensions. First, we demonstrate the expressive power of our mechanisms by analyzing concrete use cases. Then we show why current mechanisms do not fulfill the users needs, whereas our extensions do. We do this by comparing our mechanisms to related models and implementations. Finally, we provide benchmarking results gathered on our proof-of-concept implementation.

5.1 Expressive Power: Case Studies

In this section, we present three case studies that are all tied to real security needs that we have either directly experienced or that were reported to us by colleagues. There was no fully satisfactory solution for any of these scenario. We show how each scenario can be addressed by our extensions and we explain why the set of existing solutions was not satisfactory.

5.1.1 Review of TE and our Extensions

Before proceeding to the case studies, we provide a review of TE and our extensions. A domain is the unit of confinement: processes run within a domain. In SELinux's version of TE, processes are in a domain by virtue of their type being the type of the domain. (As a reminder, domains are types that bear the `domain` attribute.) Consequently, permissions are granted to types. These permissions are expressed in terms of operations that can be performed on objects of a given class and type. The class of an object is directly determined by the class of system resource

that the object belongs to (e.g. file, directory, socket). The object type, however, is determined according to the local deployment of the TE policy. This is where overlay labeling comes in handy by supporting refinements of the grouping of objects, while preserving the semantics of the original deployment of the policy. The final step in the configuration of a domain is to make it reachable from other domains. This involves configuring programs as entrypoints of domain and setting up automatic domain transitions. The entrypoints of a domain are the programs through which the domain can be entered. Based on the current type of a process (i.e. its domain) and the type attached to the program it attempts to execute an automatic domain transition determines the type that will be attached to the process after it starts executing the new program. The domain transition, however, will only happen if three conditions are met. The transition must be allowed by the policy, the process has to be allowed to execute the program, and the program must be declared as an entrypoint of the transition's target domain.

5.1.2 Subdividing a User Account: The Grading Program Problem

In this section, we revisit the grading problem that we introduced in Section 4.1 to motivate the introduction of overlay labels. We provide a more complete treatment of this example, in which we also illustrate the rationale for the introduction of administrative templates in Section 3.3.3. The remainder of this section covers the grading program case study, following the presentation order that was used for the previous review on TE and our extensions.

Properly addressing the grading program problem requires several features. The user deploying a grading program should be able to create and configure a new confinement domain, which includes configuring the entrypoints of the domain. Then, the user should be able to decide which resources are accessible from within that domain. These operations (and their authorization) can be specified as follows.

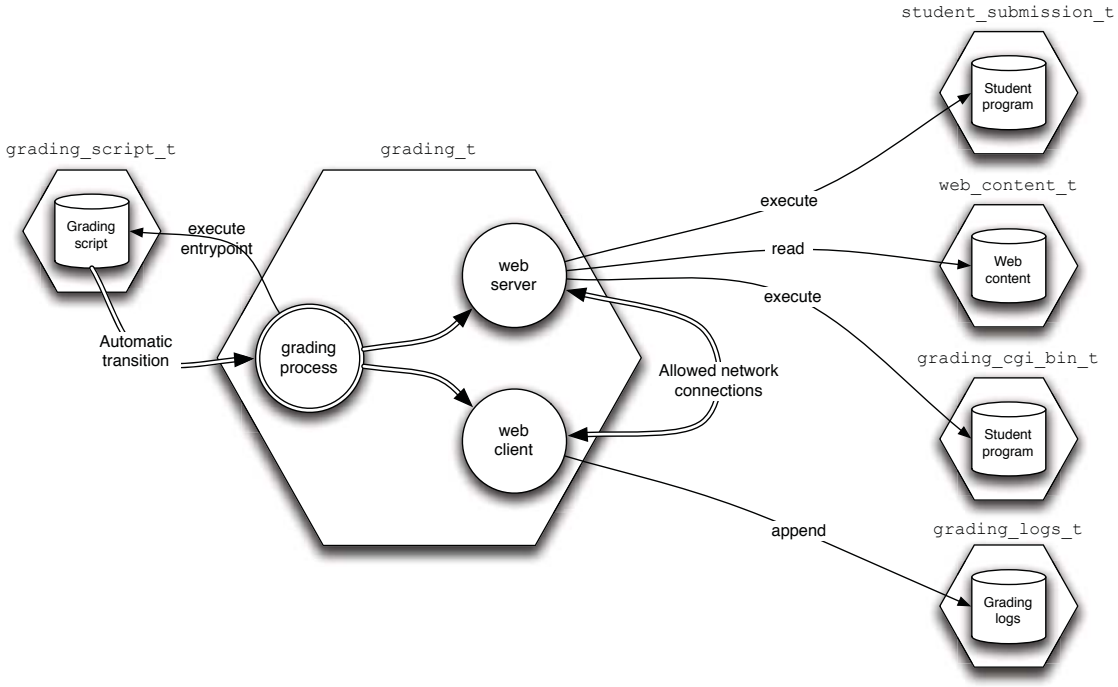


Figure 5.1.: Summary view of the permissions granted to the domain (`grading_t`) used to confine the grading program.

Creating a domain

Creating a domain involves two operations, and therefore two permissions: creating a new type, and then attaching the `domain` attribute to that type. Our preferred solution is to allow the creation of types through attributes. This way, types are automatically labeled with an attribute that can be used to refer to them in the policy. This comes handy when one wants to specify that a given user (here, the TA) is allowed to create domains. As we explained in the TE recap, creating a domain involves two operation. The creation of types through attributes allows to connect the two rules needed to allow the creation of a domain. Without such a means of connecting the two rules, the policy would have to be somewhat hard-coded: the names of the domain that a user is allowed to created would have to be explicitly mentioned in the policy. By using attributes, this restriction is avoided. The two rules to allow the TA to create a grading domain are the following.

```
1 allow ta_t ta_type:type create;
2 allow ta_t ta_type:attribute(domain) attach;
```

First, the TA is allowed to create types through an attribute (`ta_types` in this example) that is attached to his account[1]. Second, the TA is allowed to attach the `domain` attribute to types that were created through the type attached to his account. To create a domain named `grading_t` in accordance with these permissions, the TA would perform the following operations on the virtual filesystem:

```
1 create /sefuse/attrs/ta_type/grading_t
2 create /sefuse/attrs/domain/grading_t
```

The first operation will create the type `grading_t` through the `ta_type` attribute; the second will attach the `domain` attribute to this new type, making it a domain. Once

[1]Our system does not directly handle the assignment to accounts of attributes through which the users can create types. We consider this step to be part of account provisioning, which falls outside the scope of our work.

a domain is created, the next step is to assign permissions to that domain, which requires some preliminary relabeling of objects.

Labeling Objects

Since all TE permissions are expressed in terms of types, the TA has to assign separate labels to objects. These labels need to be different when the granted permissions have to differ. For instance, the web server should have have only read access to the static content it serves, and the web client should only have append access to its logs. Consequently, the static web content has to be labeled with a type that is different form the type that labels the logs.

As illustrated in Figure 5.1, the static web content is labeled with the type `web_content_t` and the grading logs are labeled with the type `grading_logs_t`. The TA needs to be allowed to perform this labeling of objects. Authorizing the TA to change the label of files requires two permissions, `relabelfrom` and `relabelto`, for each relabeling operation.

For instance the following permissions allow the TA to relabel regular files in his home directory, labeled `ta_home_t`, with the type associated the static web content, `web_content_t`.

```
1 allow ta_t ta_home_t: file relabelfrom
2 allow ta_t web_content_t: file relabelto
```

Type attributes can be used to factor the relabeling rules between the standard label of the TA's home directory and any type that the TA has created.

```
1 allow ta_t { ta_home_t ta_type }: file { relabelfrom relabelto }
2 allow ta_t { ta_type ta_type }: file { relabelfrom relabelto }
```

The first rule[2] allows the TA to relabel files from (resp. to) the type of his home directory files to (resp. from) any type that he has created through the `ta_type`

[2]SELinux supports a compact notation for access vector rules, where each field can contain a set of elements of the expected kind, separated by spaces and surrounded by curly braces. Internally, these rules are expanded to the simple format of rules that we modeled in Chapter 3. We use this format here and in following examples to compactly represent access vector rules.

attribute. The second rule allows the TA to relabel any object, currently labeled with a type he created, with another type he created. The relabeling is performed using the standard SELinux `chcon` "change security context") utility.

Assigning permissions to a domain

There are two ways to let a user assign permissions to a domain. The straightforward approach is to grant the administrative permissions one by one. For instance, the following rules state that the TA can grant the permission to read files from his account (labeled `ta_home_t` to types he has created through the `ta_type` attribute:

```
1 allow ta_t _:av_rule(ta_type, ta_home_t, file, getattr) insert
2 allow ta_t _:av_rule(ta_type, ta_home_t, file, open) insert
3 allow ta_t _:av_rule(ta_type, ta_home_t, file, read) insert
```

Clearly, granting the administrative permissions one-by-one can be tedious. This is one of the main reasons why we have introduced the notion of administrative templates in our model (see Section 3.3.3). Instead of having an administrator essentially create one administrative rule for each administrative permission that he wishes to grant the user, an administrator can grant the user the permission to add permissions to a domain, provided that another (template) domain possesses the same permissions. For instance, the following rule allows a user to grant any permissions from his default domain to a domain that he has created.

```
1 admin_domain_template ta_t ta_t ta_type
```

As they allow the factorization of the administrative policy, administrative templates make it easier for the admin to both define and reason on the administrative policy.

For the grading program, the TA wants to grant several permissions to the domain where the grading program will be confined, `grading_t`.

```
1 allow grading_t grading_script_t : file { read gettatr execute \
2                                           entrypoint open }
```

```
3 allow grading_t web_content_t : file { read getattr open }
4 allow grading_t grading_logs_t : file { append getattr open }
```

The network access control permissions are covered in the next case study, where we analyze the deployment of web applications.

Fine grained grouping of objects

The deployment of the policy, and hence the granularity at which the system groups objects under types, sometimes forces a coarse granularity on the policy that can be expressed on objects. This is due to the fact that the TE policy can only be expressed in terms of the types and attributes attached to objects. We have introduced overlay labels in Chapter 4 to circumvent this limitation. Overlay labels allow users to refine the grouping of system objects in an arbitrary fashion.

For instance, the web server that students implement has to support some Common Gateway Interface (CGI [156]) features. Some of the CGI scripts that the server runs rely on system binaries (e.g. `cal` to provide a textual calendar). This means that the web server has to be allowed to execute some of these system binaries. As most system binaries are labeled with the same type (`bin_t`), giving access from the grading domain (`grading_t`) to the default type of system binaries (`bin_t`) would result in un-necessarily broad permissions.

Instead, the TA can use a filesystem overlay label to group together binaries that are referenced from the CGI scripts used for grading.

```
1 fs_overlay /usr/bin/cal grading_cgi_bin_t
2 fs_overlay /bin/date grading_cgi_bin_t
3 fs_overlay /bin/echo grading_cgi_bin_t
```

The grading domain can then be granted execute access to this subset of the system binaries, instead of access to all system binaries.

```
1 allow grading_t grading_cgi_bin_t : file { getattr open read \
2                                            execute_no_trans }
```

Defining the domain entrypoints

The entrypoints of a domain are sensitive by nature. These programs are, in essence, the gatekeepers of a domain: they are trusted to restrict how the permissions granted to the domain are used, by offering a limited set of operations that can be performed. For instance, the `passwd` program limits how the read/write permission on the `/etc/shadow` file (where password hashes are stored) granted to the `passwd_t` domain can be used.

Contrary to this example, there is no need for an administrator to worry about the entry points that a user sets for domains that are strictly sub-domains of his user account. Indeed, allowing the creation of sub-domains enables security-conscious users to better protect the permissions they are entrusted with, by confining programs to which they do not want to extend all of their ambient permissions. In other words, we consider that restricting the entry points that a user can set to one of his account sub-domains would be counter-productive, as it could discourage users from using the account sub-domain feature[3].

Therefore, we think that a user should be able to set any executable file as an entrypoint to a sub-domain of his account, as long as it is executable from their user account. In the grading program problem, a practical approximation is to allow the TA to set any file labeled `bin_t` or `ta_types` as an entrypoint to a domain that bears the attribute `ta_types`. The corresponding administrative permissions are the following.

```
1 allow ta_t _:av_rule(ta_types bin_t:file entrypoint) insert
2 allow ta_t _:av_rule(ta_types ta_types:file entrypoint) insert
```

[3]We think, however, that administrative control on the entrypoints of domain remains a good idea in other cases, for instance in the case of web applications, which we discuss next.

Setting up automatic domain transitions

To finalize the setup of the grading domain, the TA needs to set up an automatic domain transition so that the grading script will be automatically placed in the grading domain upon execution. The type transition rule to enter the grading domain upon execution of the grading script looks as follows.

```
1 type_transition ta_t grading_script_t : process grading_t
```

The following administrative permissions lets the TA perform this operation. The first one allows exactly this operation, whereas the second is a generalized version that uses attributes.

```
1 allow ta_t _:tr_rule(ta_t, grading_script_t, process, grading_t) insert
2 allow ta_t _:tr_rule(ta_t, ta_types, process, ta_types) insert
```

Summary

We have shown how our extensions allow a regular user (here, the TA) to define subdomains within their user account to confine applications that they decide not to trust (here, student submissions). These subdomains have permissions that are a strict subset of the permissions of the user account. We restricted our presentation to covering overlay labels for filesystem objects. In the next section, we use the deployment of web applications as an example to present the usage of network packets overlay labels and how users can be allowed to grant permissions that they do not possess.

5.1.3 Hosting User-owned Web Applications

When we revisited the grading program problem, we showed how a user can create domains and grant them a subset of the user account permissions. Here, we look at an extension of this scenario: confining web applications on a web server. There are two

main reasons to confine web applications: protecting the host systems, and shielding applications from one another. The confinement is desirable to address the threat of an application behaving maliciously. This malicious behavior usually results from the application having a security flaw that gets exploited. We postpone presenting the confinement of hostile applications to section 5.1.4.

The attentive administrator of a web hosting site can follow recommended security guidelines (e.g. [157]) and achieve a setup where the host system is protected from vulnerabilities in the hosted web applications. In other words, the host system can not get corrupted through a hijacked web application, but that is not what we are interested in solving.

We are interested in solving the other motivation for confining web applications: to isolate them from one another. More specifically, we will show in the following how our extensions enable the confinement of web applications, even when they run in the same UNIX user account. The cheapest solution for hosting a web site is virtual hosting, where not only the physical machine, but also the operating system instance and the web server daemon are shared by several users. For instance, this is how personal web pages are currently supported at Purdue University and several other academic institutions (e.g. Stanford University and the University of North Carolina at Chapel Hill [4]). To ensure that web applications installed by one user can not access the data of other users, these sites resort to using a feature of the Apache web server that runs the applications under the user identity. This feature is called `suExec` [157], as the web server process executes the user application after calling the `setuid` system call, to set its identity to that of the user that installed the application. Consequently, the application can only access data that is normally available to the user on behalf of whom it is running.

The `suExec` solution has also been extended to support the same feature for PHP applications [158]. These solutions, however, suffer from the same limitation

[4]See `http://www.stanford.edu/services/web/cgi/security.html` and `http://help.unc.edu/3136`.

that we highlighted in the previous section: they do not offer means for a regular system user to further refine the confinement of her application. All the applications installed in the same user account run with the same user privileges, and therefore can compromise one-another.

Our extensions can be used to remedy this situation. The situation in this case study is more complicated than the previous case study, as the domains in which the web applications are to be confined requires privileges that a normal user application would not have. Indeed, regular user applications are not allowed to talk with the web server. On the other hand, web applications *must* be allowed to do so if they are to serve web requests.

We expose the use of our extensions in two steps. First, we cover how the system administrator (or the web administrator) can extend some of his administrative privileges, in a controlled fashion, to a user that wants to install a web application. Then, we show how this user can further refine the permissions granted to this application in order to enforce a fine-grained confinement. Refining the permissions is important to reduce the attack surface of web applications, and to reduce the exposure of the user's account to an application that would be hijacked.

Subdomains with Additional Privileges

A user-deployed web application needs to receive a subset of the user's permissions and a subset of the webserver's permissions in order to be functional. The user's permissions are needed for the application to perform operations within the user's account: access data, perform computing tasks, and store back some application state. The webserver's permissions are needed for the application to handle a connection received by the webserver, on a port normally reserved to the webserver, and accepted through a file descriptor that belongs to the webserver.

The permissions that the user-deployed web application need to answer requests received by the web server are the following.

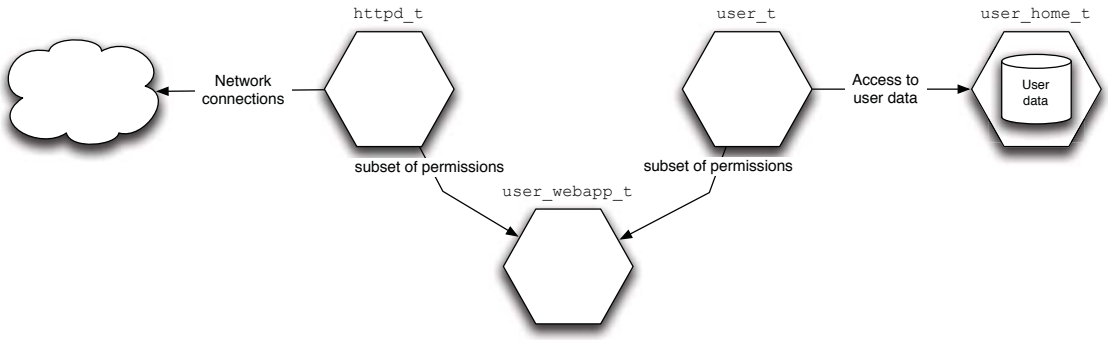


Figure 5.2.: The domain of a user web application can receive permissions both from a web application permission template and from the user domain, treated as a permission template.

```
1 allow user_webapp_t httpd_t : fd use
2 allow user_webapp_t netif_type : netif { ingress egress }
3 allow user_webapp_t port_type : tcp_socket { recv_mesg send_mesg }
4 allow user_webapp_t http_server_packet_t : packet { send recv }
```

The first rule allows read access to open file descriptors passed by the web server to the web application (this is the mechanism by which a web server hands over the request handling to child processes). The second rule allows ingoing and outgoing traffic of the web application domain to go through network interfaces. The third rule allows a bidirectional flow of data over a TCP socket. The `name_bind` and `name_connect` have deliberately been omitted from this rule, as in this case we do not want to let the web application, either bind sockets to ports, or initiate remote connections. The fourth rule allows the datagrams to be sent and received on the socket by the web application.

To set each permission, the user needs an administrative permission that allows the setting of this permission. Each of these administrative permissions have to be granted by an administrator. It is desirable for the administrator to have a means of factoring the administrative policy, as noted in the previous case study. We have designed our administrative templates so that they can be composed. This composability allows the granting of administrative permissions from several templates, to the same user, and on the same domain. For instance, the above permissions can be used as a template that lets the user assign the same permissions to one of his domains, with the following administrative template.

```
1 admin_domain_template user_t user_webapp_t user_types
```

This template can be coupled with another template that lets the user grant permissions from his main domain to a domain he has created through the `user_types` attribute.

```
1 admin_domain_template user_t user_t user_types
```

This composition of administrative templates in a non-hierarchical way is powerful. The Policy Management Server proposed by Tresys [73] can not handle this scenario.

Refining the Permissions

The networking permissions granted to the `user_webapp_t` domain in the example above are broad: while they do not allow the domain to create new sockets, the domain is allowed to communicate with any host that the webserver accepted a connection from, since the connections are passed by means of an open file descriptor which corresponds to an unknown (as far as the application is concerned) endpoint. This file descriptor corresponds to the socket on which the web server accepted the client connection. The file descriptor is left open when the web server forks a new process which starts executing the user's web application.

As we mentioned in the introduction of this case study, it is desirable for the user to grant only a refined subset of the permissions of her user account to a domain in which a web application will run. In the case of an online journal (a blog), several refinements are interesting.

If the blog is used to convey proprietary information, it is desirable to restrict the network permissions of the domain so that it can only communicate with hosts on the internal network. This can be done using the following network overlay label.

```
1 net_overlay --source 10.0.0.0/24 intranet
2 net_overlay --dest 10.0.0.0/24 intranet
3 allow intranet_blog intranet : packet { send recv }
```

The first two rules define the `intranet` network packets overlay overlay label that labels packets sent or received on the internal network. The last rule allows the `intranet_blog` domain to send and receive packets that bear the `intranet` label.

Another way to refine the permissions granted to a web application is to use predicate overlays. For instance, if the user wants to grant execute access to all

executables except the compiler, the following two overlay labels can be used (by default, the compiler is labeled like other regular executables with the `bin_t` type).

```
1 fs_overlay /usr/bin/gcc compiler
2 predicate bin_not_compiler ( bin_t AND NOT compiler )
```

The first overlay label attaches an additional attribute `compiler` to the GCC compiler binary, located at `/usr/bin/gcc`. This is a filesystem overlay label (see Section 4.2.3). The second overlay label expresses the fact that if a an object is labeled with the `bin_t` attribute[5] and not with the `compiler` attribute, then it should be labeled with the `bin_not_compiler` attribute. This second overlay is a predicate overlay (see Section 4.4).

5.1.4 Analysis of Malware

The analysis of malware, and more generaly performing experiments with potentially hostile software is a use case that was reported to us by Pascal Meunier, in the context of the ReAssure project[6] (see Figure 5.3). ReAssure is a network testbed that was designed from the ground up to offer a strong confinement in order to safely support any kind of experiments, including the manipulation of viruses, worms, and botnet software.

In the previous case studies, we have illustrated how a user can create new domains and grant them a subset of the permissions he possesses on his account. We have then showed how a user can be allowed to grant permissions that he does not have in his own account, e.g. handle connections received by the web server. Finally, we have showed how a user can refine the labeling of filesystem and network packet objects on the system in order to define permissions as precisely as he wishes.

[5]When using overlays, `bin_t` will be an attribute, and not a type as it would be in the standard policy, since overlay labels rely on a previous pass of type promotion. Please refer to Section 4.2.3 for an explanation as to why, when using overlay labels, `bin_t` is an attribute and not a type anymore.

[6]`http://reassure.cerias.purdue.edu`

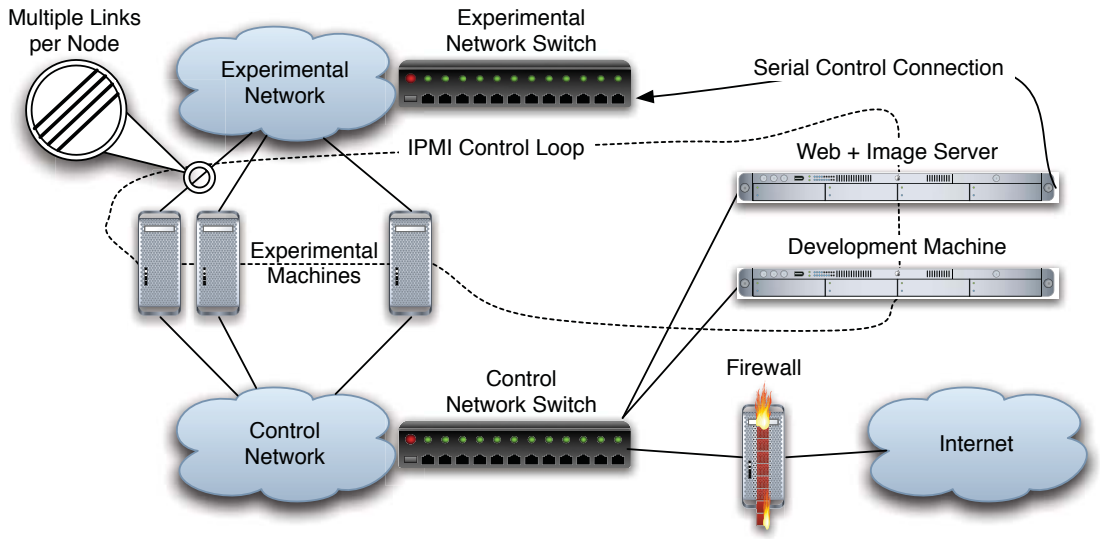


Figure 5.3.: Architecture of the ReAssure testbed. The testbed consist of a set of machines on which the experiments are run (left). There are two physical networks. The control network (bottom) is used to deploy images from the image server (right) to the experimental machines, and to remotely connect to them. The experimental network (top) can be configured to emulate any arbitrary topology; the experiments run on this network.

In this case study, we show how SELinux and our extensions can help in experiments with malware. Then we show how a user could share administrative permissions on a domain. In the context of this case study, this would allow the sharing of an experiment.

Experimenting with Malware

Since SELinux was designed with assurance in mind, it includes logging mechanisms. All denied accesses are logged by default. Additionaly, granted accesses can also be logged. When performing behavioral analysis of malware, this audit log is useful in determining the operations that the malware is attempting. It is up to the experimenter to decide which operations to allow.

SELinux, however, relies on the integrity of the Linux kernel as a whole. Consequently, SELinux's protection is not sufficient when experimenting with malware that loads code in the kernel. Such experiments can be performed by either running the experiment in a virtual machine, or by protecting the kernel with access control performed by an underlying hypervisor [159, 160]. The hypervisor approach falls outside the scope of this work. The problem with virtual machines is that they are themselves vulnerable to attacks, as illustrated by the CVE-2005-4459 and CVE-2009-1244 vulnerabilities [161]. These attacks allow arbitrary code execution on the host platform of a virtual machine. Since ReAssure relies on the administrative network to be free of attacks, it is important to guarantee that an attacked virtual machine will not be able to send network traffic on the administrative network. The ability to finely confine applications, including their network traffic, that we demonstrated with the web application case study can be applied in this case as well to increase the assurance that the application is contained.

Sharing Experiments

By default, experiments deployed on the testbed are private. A feature being investigated with the ReAssure project is to have several users of the testbed collaborate on and share an experiment. In the following, the experimenter that created the experiment will be called Alice and the experimenter with whom Alice decides to share her experiment will be called Bob, for brevity.

An experiment can be shared in different ways. Alice can let Bob access or modify resources of the experiment, by granting Bob access to files used by the experiment. This can be expressed using an administrative template.

```
admin_domain_template alice_t alice_t bob_t
```

A deeper level of sharing is for Alice to let Bob run the experiment, which involves the ability for Bob to execute at least one of the entrypoints of the experiment, an automatic type transition from Bob's domain to the experiment domain, and the permission that allows this transition. The above administrative template already allows Alice to grant Bob the permission to execute the entrypoint of the experiment. Two additional administrative permissions are required for Alice to be able to share her experiment with Bob in this way.

```
allow alice_t _:tr_rule(bob_t, experiment_exec_t, \
                        process, experiment_t) insert
allow alice_t _:av_rule(bob_t, experiment_t, \
                        process, transition) insert
```

Finally, if Alice decides to share all her administrative rights on the experiment with Bob, she can do that by inserting the following administrative template in the policy.

```
admin_resource_template bob_t alice_t experiment_t bob_t experiment_t
```

If Alice wanted to share only a subset of her administrative permissions on the experiment, she could create a template domain, `bob_permissions`, that has only these

permissions on the experiment. Then she would create an administrative template that grants these permissions to bob.

```
admin_resource_template bob_t bob_permissions experiment_t bob_t \
                            experiment_t
```

Alice could be allowed to create these administrative templates by the following permissions.

```
allow alice_t _:admin_resource_template(bob_t, alice_t, experiment_t, \
                            bob_t experiment_t)  insert
allow alice_t _:admin_resource_template(bob_t, bob_permissions, \
                            experiment_t, bob_t experiment_t)  insert
```

5.1.5 Summary

The common trait of the case studies we have presented is that users can explicitly manipulate TE permissions, domains, and domain transitions. The specification of these manipulations can be arbitrarily coarse or fine-grained. Moreover, a wide range of manipulations can be specified. At one extreme, no manipulations are allowed except to the system administrator; this extreme corresponds to the situation on SELinux without our extensions. At the other extreme, any user of the system is permitted arbitrary manipulations of permissions; this extreme is not useful. Between these extremes, many scenarios can be supported, from letting users segregate the applications they use into domains to which they grant only a chosen subset of their user permissions, to delegating permissions on the policy that covers system services.

5.2 Comparison to Previous Work

In this section, we compare our work to closely related work.

5.2.1 Administrative Models

The domain and resource templates are comparable to the notion of administrative roles in RBAC administrative models: they are abstraction that support the grouping of administrative permissions, which can then be granted to subjects of the system, enabling to grant the corresponding permissions.

ARBAC

Our model differs from the ARBAC family of administrative models [56, 75] in the following aspects.

First, we do not introduce additional policy constructs without a means of administering them. The administrative permissions that we introduced in Section 3.3 are recursive. The rationale and implication of this design is that, for any permission, a permission can be defined to regulate its creation or removal. Consequently, when we introduced administrative templates in Section 3.3.3, we introduced the accompanying administrative permissions that regulate their creation and deletion. This approach ensures that the administrative policy supported by our mechanisms is not hardcoded in them, but actually a policy as well, whose modifications can also be regulated by an administrative policy. The goal of this last point is to allow changes to the administrative policy, while constraining them. The motivation behind this goal is to allow the enforcement of the principle of least privilege in the granting of administrative rights as well as regular rights. The ARBAC model does not address this aspect.

Second, administrative permissions can be specified at different granularities. In the PRA97 part of the ARBAC97 model, the assignement of permissions to roles is

regulated according the *can_assignp* and *can_revokep* administrative relations. While these relations take into account the role to (or from) which the permissions can be assigned (or revoked), as well as preconditions on the roles to which the operation is applied, they do not take into account *which* permission can be granted.

UARBAC

UARBAC [77] is another administrative model for RBAC. It differs from ARBAC by relying on a principled approach to its design, as opposed to ARBAC which "was developed in a piecemeal manner" [56]. Using a principled approach to design is commendable. We thus evaluate how our extensions satisfy the requirements enunciated in [77] as a consequence of the principles being followed. We consider each requirement in turn, substituting TE for RBAC when needed.

1. "Support decentralized administration and scale well to large [TE] systems."

   Our administrative model support the definition of permissions on administrative permissions themselves. Consequently, any administrative permissions can be delegated, which supports the decentralized administration. The support of permissions templates allows the grouping of permissions, which is turns supports the scaling of the administration to large sites. Moreover, type attribute predicates can be used to further factor the policy by expressing some of its properties at a higher level.

2. "Be policy neutral in defining administrative domains."

   Our administrative model does not impose any constraints on administrative permissions, besides well-formedness. The creation of types through attributes is used simply as a means of tagging new types to tie them in the administrative policy and allow post-creation administrative operations on them.

3. "Apparently equivalent sequences of operations should have the same effect."

   Granting administrative permissions through administrative templates is equivalent to granting each administrative permission referenced by the template.

4. "Support reversibility."

   All operations from our administrative model are reversible, except the destruction of policy elements, which is the case in UARBAC as well. The algorithms for reverting type promotion and overlay labeling are provided in Section 4.6.

5. "Predictability."

   We have designed all the administrative operations to be as straightforward as possible. By offering indirect means of granting administrative permissions, however, we may have introduced some slight chance of surprising users. For instance, if administrative permissions are defined according to a template, any change to the template domain can impact the users whose administrative permissions are based on the template. This is an area where our design could be improved, one possible way being to support immutable administrative templates, which in turn would require to extend TE with explicit negative permissions that override positive permissions (see our discussion on negative permissions in Section 4.4.4, where we discuss the support for the negation operator in predicate attributes).

6. "Using [TE] to administer [TE]"

   Our original design followed the path of Tresys's PMS [73], in trying to attach a type to policy constructs. This would have allowed the administration of TE to be defined exclusively in terms of TE rules. However, as argued in Section 3.3.1, this is not a practical solution. Consequently, we have expressed our model as an extension to TE that adds support for pattern matching policy objects.

5.2.2 Operating System Access Control

In this section, we compare the mechanims offered by our work, on SELinux, with other work on operating system access control.

Traditional Mechanisms

As discussed in the introduction and the related work sections, traditional discretionnary access-control mechanisms on UNIX (`setuid` [48], `chroot`, and `jail`) can be used to enforce some confinement on processes, towards applying the principle of least priviledge. However, superuser privileges are still needed to configure such confinement. Moreover, the permissions to choose from are sometimes coarser than the permissions of the API accesses they control, as noted in the case of the socket API.

While they are not configurable by regular users, these mechanisms can still be used to deploy effective countermeasure to privilege escalation, by using privilege separation [6] which was proven effective in practice. The scheme of using file descriptors as capacities described in [6] is also used with Type Enforcement deployments, where is it easier to audit due to the finer grain of the policy used to confine the different components of the application.

Systrace

Systrace [91] performs access control by allowing the specification allowed system call patterns, which are enforced by system call interposition. This allows for a refinement of the ambiant permissions that are granted to a process. Using system call interposition has many pitfalls [92], one of which is its handling of file aliasing: if a file is pointed to by two different hard links, the access decision performed based on the file path may be different, depending on the path provided to the system call. This handling of aliasing can be viewed both as a feature and a way to bypass the con-

finement; we are concerned about the second view[7]. Systrace provides an additional primitive that fall outside of this category: privilege elevation. The idea of privilege elevation is that, even for system daemons, only a few system calls require elevated privileges. Instead of exerting these privileges in a separate daemon, as in privilege separation [6], the privileges of the current process can be temporarily elevated just for the operation that requires elevated priviledges. For instance the `bind` operation to a priviledged port, say port 80 for a web server, requires root priviledges. With systrace, a web server can be run unpriviledged and have its priviledges elevated just for the bind operation on port 80. This is not a mechanisms that is supported in our work.

AppArmor

One of the main goals of AppArmor [108] was usability. Recent analyses by Chen et al. [162] tend to confirm that this goal was met. By refusing the abstraction afforded by type labeling, to preserve usability by keeping the familiar pathnames, AppArmor makes it harder to compose proofs for an audit of a platform. Also, by exposing only a restricted set of permissions (again for usability), AppArmor forces some level of granularity on the permissions that it can regulate. What we consider the worst example of this coarse granularity is the fact that network communications can only be restricted in terms of families of protocols (e.g. ethernet and bluetooth at layer 2, IPV4 at layer 3, and TCP at layer 4); there is not notion of endpoints! While the network access control can indeed be enforced by firewalling features of the OS, AppArmor does not offer a means of coupling this filtering directly with the application. For instance, there is no way to guarantee that only the web server can listen on port 80.

AppArmor transitions between profiles (a notion similar to TE domains and domain transitions) are also more restricted than the domain transitions supported by

[7]The presentation of systrace extensively discusses how the problem of aliasing through symbolic links (symlinks, typically created by running `ln -s`) is addressed. It does not address the problem of aliasing through hard links (created by running `ln`, without the `-s` flag).

TE. AppArmor profiles are entered based only on the path name of the application being executed, as opposed to TE, where the source (type) is also considered. So AppArmor will support only one profile per path name, regardless of the calling profile, whereas SELinux will be able to offer different transitions (and resulting permissions) to subjects that invoke the same applications from different domains.

TrustedBSD, SEDarwin, and RSBAC

TrustedBSD [68], SEDarwin [114], and RSBAC [64] can all be used to compose a form of mandatory access control with UNIX discretionnary access controls. TrustedBSD and SEDarwin provide an implementation of TE, and RSBAC could be extended with one. We have chosen SELinux as a base for our work instead because SELinux's integration at the level of the whole system is more mature, for instance with SELinux being enabled by default on RedHat Fedora Core distributions, starting with Fedora Core 3 (released in November 2004).

Capsicum

Capsicum [163] is an extension of the UNIX API with capabilities, implemented on FreeBSD, which aims at offering a gradual migration path for applications to be modified to use capabilities. The capabilities are implemented as wrapped UNIX file descriptors, and the system offers two execution modes for processes: vanilla UNIX or capability mode. At runtime, a UNIX application can transition to capability mode and have its access permissions refined by the capability mechanisms. As such, applications need to be modified to benefit from this confinement mechanism, but the modifications can be minimal (2 additional lines of code to confine `tcpdump`, for instance [163]). The comparison of Capsicum to TE [163] points that TE can not offer a comparable solution because TE application policies can not be adjusted dynamically due to the lack of an administrative model for the TE policy and object labeling. Our work addresses this point.

Distributed Information Flow Control

The distributed information flow control (DIFC) model [28], was originally designed as a model of security within a programming language model, and has been implemented in the Jif compiler [30]. More recently, DIFC has been used as a model of operating system security. We compare our work to the results of two operating systems that implement DIFC, Histar and Flume.

HiStar [121] is a capability-based OS built in a modular fashion that limits the size of the trusted computing base, and with DIFC built in from the ground up. This makes the access control mechanisms provided by HiStar more amenable to verification than the ones provided by SELinux, which is built on a large monolithic kernel where the TE mechanisms were retrofitted (see Section 3.4.2 where we discuss the assurance on SELinux extended with our administrative model and its implementation). HiStar, however, provides a modified system API, hence depriving users of backwards compatibility with the applications they use.

Flume [164] is an extension of Linux and OpenBSD with additional mediation that supports DIFC. This work was performed to address the practical limitations of HiStar, which include limited support for harware diversity and the need for applications to be significantly rewritten to run atop a different system API. Flume still requires that the applications be modified, but to a lesser extent. Similarly to our work, Flume is built as an extension that is mostly written in user-space for ease of development and portability. The main limitation of Flume compared to our work is the need to modify applications and the performance degradation Flume's implementation introduces (43 % slower on read workloads and 34 % on write workloads).

With the exeption of network overlay labeling, for which there is no implementation at this time, our extensions do not result in additional inline computation during access checks. This property stems from the fact that our extensions only modify the TE policy (for all administrative operations), and object labels (when deploying filesystem overlay labels); the enforcement mechanisms of SELinux remain

unmodified. As shown in [165], the additional cost of the access controls performed by SELinux is low (at most 4 % overhead on macrobenchmarks), hence offering better performance than the performance reported for Flume [164].

Pinup

PinUP [107] simplifies the DIFC model to focus on which applications can access which files. By doing so, PinUP is able to preserve the UNIX API and thus the is able to run unmodified applications. PinUp does not cover network access controls. Also, it does not offer differentiated accesses to applications depending on the context from which they were invoked, besides identifying the user invoking the application. This means that a subverted web browser, which is not granted access to high value files, could invoke a trusted text editor that has access to these files and use this editor to access the files[8]. To prevent this kind of attack, the context in which applications can be called has to be restricted. The TE model can capture the calling context, which is identified by the domain attempting to execute an application.

5.3 Performance

As explained previously, our administrative model does not interfere with the code path of system calls (see Section 5.2.2, where we discuss the relative performance of SELinux and Flume). Access control is still performed according to the SELinux standard implementation as a Linux Security Module. Consequently, our administrative model has no additional impact on the performance of the system. Moreover, single edits of the policy are faster through the virtual filesystem than when using the policy module mechanism provided by SELinux. This result correlates with the performance observations on Adage [166] authorization system, where the binary representation of the security policy was also edited in an incremental fashion to improve

[8]For instance, a terminal-based editor like `vi` can be "remote-controlled" by an `expect` script.

the responsiveness of the administrative interface. Contrastingly, the policy module system currently supported by SELinux involve an expensive recompilation of the whole policy every time a policy module is loaded or unloaded.

The responsiveness of the filesystem could be further enhanced by migrating the implementation from user space to kernel space. Such a migration may prove necessary to support acceptable performance when implementing overlay network labels (described in Section 4.3). Indeed, the labeling of a network connection can require the creation of a new synthetic type. Currently the virtual filesystem updates the policy in the kernel by serializing the policy that resides in user space, and loading it in the kernel by a write to the `/selinux/load` virtual file exposed by the SELinux kernel module. The cost of this operation is on the order of a second, with a the example policy provided with RedHat Fedora Core 10, on a Pentium 4 running at 1.4GHz[9]. This update time may not be an acceptable overhead for the on-demand generation of synthetic types that will be required to support network packets overlay labels; it seems acceptable otherwise.

5.4 Conclusion

On the theoretical side, our work is aligned with the direction of the work on administrative models. Our model follows a principled approach that avoids pitfalls from previous administrative models. For instance, our model avoids relying on administrative hierarchies, which are a limiting factor for ARBAC as well as PMS.

On the practical side, we have compared our work to existing mechanisms available on UNIX systems, as well as to the work on DIFC and recent work that attempts to make capabilities support as backwards-compatible with existing applications as possible. The existing mechanisms on UNIX are not administrable by regular users and are often coarse-grained as well. The work on DIFC modifies the system API,

[9]The example policy we used, once compiled, expands to 278925 access vector rules, out of which 187604 are allow rules and 91321 are audit rules. There are 7955 are type transition rules, 2555 types, and 219 attributes in this policy.

which necessitates a refactoring of applications. Also, the current work on DIFC is either not available on a current mainstream OS (needed for backwards compatibility with existing applications), or introduces a significant degradation of performance. Measurements of the performance of our prototype indicate that there is no such degradation of performance, and the system API is not modified.

Therefore, we find that our work compares favorably to existing work, for the goals it is striving to achieve (see our thesis statement in Section 1.3). Furthermore, the publications that present the Flume [164] and Capsicum [163] systems, which have similar goals, explictly mention that implementations of TE (SELinux or TrustedBSD) would compare favorably to their systems, provided they were extended with an administrative model. We consider this an additional justification of our approach.

6. CONCLUSION

In this chapter, we reflect back on the work that was presented in this thesis. We first look at what this work accomplished on a theoretical aspect and on a practical aspect, and how this addresses our thesis statement. Then, we conclude by outlining future directions in which this work could be extended.

6.1 Results

As we mentioned in our survey of the related work, our work relates to access control models and administrative models on the theoretical side, and relates to their implementations on the practical side. Therefore we consider our results from both perspectives.

6.1.1 Theoretical Results

In Chapter 3, we have formalized a significant subset of the TE features, which we named TE-core. This model was build in a way that the features can be selected and composed around the TE-base nucleus, which provides the basic evaluation of access vector rules. We have modeled Core RBAC using the same formalism that we used to model TE-core. With these two models, we have shown that a reduction from Core RBAC to TE-core exists, by constructing one, and that it is not possible to construct a reduction from TE-core to Core RBAC. Showing this inequal expressive power is a new result.

In our survey of existing access control models (see Section 2.2), we have collected several other results on the relative expressive power of access control models, defined in terms of model reductions. These results were previously scattered in the existing

litterature. The summary figure that compares the expressive power of the surveyed models (see Figure 2.9) is a contribution in its own right. This figure incorporates our contribution to these results, the comparison of TE-core and Core RBAC. Furthermore, our unified representation of access control models, base on extending the Extended Access Matrix [16] with subject transitions and object transitions supports the fine characterization of differences between access control models. For instance, it shows that a low-water mark model of integrity [14, 105, 106] can not be encoded in terms of a TE policy because TE does not support either object transitions or subject transitions on `read` or `write` operations.

The packet classification that returns all the applicable labels for a given network packet, which we developed to address the need for overlay labeling on network packets, is a new problem. The complexity of our solution is optimal and consists in a novel application of computational geometry to network packet classification.

Additionally, our administrative model allows the delegated administration of the system's security policy. The delegation of administrative privileges can be specified at the level of individual policy statements, hence supporting arbitrary schemes of delegation. For concision, delegation can also be specified by analogy with existing permissions of the policy, using the administrative templates described in Section 3.3.3.

6.1.2 Practical Results

In essence, we have managed to "Make least priviledge a right (not a privilege)" [117], without modifying the UNIX API on Linux, a popular version of UNIX. We have done so by designing an administrative model for the SELinux implementation of Type Enforcement (TE) that is part of the standard Linux kernel distribution. This administrative model can allow a user to turn any of her UNIX permissions, which are normally ambient and coarse-grained permissions permissions, into fine-grained explicit permissions, whose assignment to domains can be controlled.

We have developed a proof-of-concept implementation of the administrative model presented in Chapter 3, which exposes the system's TE policy through a virtual filesystem. This implementation relies on the FUSE infrastructure, which is also part of the standard linux kernel distribution. Consequently, our implementation is usable on all recent Linux systems (past version 2.6.28 of the kernel, and provided the SELinux feature is activated).

As noted in chapter 4, an administrative model that covers only the TE policy is not sufficient to allow users of the system to configure confinement units that enforce the principle of least privilege. This limitation owes to the fact that TE reasons *only* on the labels attached to system object. Consequently, we have designed a set of techniques to allow users to refine the labeling of system objects, while preserving the semantics of the system policy. These techniques, however, have not yet been implemented.

6.1.3 Addressing the Thesis Statement

As we have shown in our survey of the related work, TE can be composed with discretionnary access control, without changing the application programming interface that existing programs depend on, and therefore without breaking the backwards compatibility with existing applications. In our survey, we have also shown that the access control mechanisms offered by TE are fine-grained. The comprehensiveness of the access controls is an implementation issue more than a modeling one. The support by SELinux of a coupling between the network-level labeling and the local application-level security policies provides such a comprehensiveness. Finally, administrative templates provide a model and mechanism by which users can be granted the ability to subdivide their permissions among many domains. These permissions can be granted without modifying any other aspect of the configuration of the MAC mechanisms, and we have provided algorithms that can be used to prevent a subversion fo the system policy.

The type of operating system for which we set to demonstrate our thesis was the one that currently runs on most personal computers: a multi-user time-sharing system where the unit of confinement is the process and processes are isolated by means of a virtual memory manager. Our work is actually applicable to many other types of systems. Fundamentally, our work is the extension of a reference monitor. The base mechanism that supports the implementation of a reference monitor is memory protection. The memory protection can be provided by means of memory segmentation, virtual memory, or even by using a type-safe language that prevents raw memory accesses by compile-time or run-time verification of the memory accesses. If a system is not multi-user, our work remains useful as we have shown in addressing the grading problem. Our understanding of early work in access control, including the thesis by Schroeder on the mutually suspicious subsystems [135], is that with proper support at the hardware level, access control can even be performed within a single program. While our understanding of TE is that it was designed to enforce inter-process access controls, there is no fundamental reason that precludes its implementation for intra-process access control.

6.2 Future Work

6.2.1 Technical Aspects

The implementation of our administrative model is currently in user-space. Therefore, all policy modifications are transmitted to the kernel by reloading a whole policy. Currently, this operation takes on the order of a second to complete. (The time taken to modify the security policy is orders of magnitude smaller.) While this is not a problem for the interactive editing of the policy by users, it will be a problem when implementing the network labeling refinements described in Section 4.3. Indeed, each combination of overlay labels is supported by a distinct type. In our design, we propose to create these types on demand. That is, when a new connection is created and no synthetic type has yet been created to represent the combination of overlay

labels attached to this connection, then the system must create a new synthetic type[1]. We do not think that systematic delays of the order of a second are acceptable with network connections. Therefore, as a preliminary step to supporting network overlay labels efficiently, the implementation of the administrative model would have to be moved to kernel space, where the active policy would be directly modified.

In its current userspace implementation, the administrative server is confined in a way that limits its exposure to attacks *and* the damage it can cause, were it to fail. The Linux kernel does not offer internal access control mechanisms to isolate its components. As a result, a failure of a kernel-space implementation of the administrative server could corrupt the whole kernel.

Another improvement we are considering is to allow predicates to reference other predicates, with some constraints to avoid the problem of cyclic dependencies between predicates. This can be supported using the same stratification techniques that are used in the implementation of deductive logic systems [155].

6.2.2 Higher Level Language

Our work addressed one of the main perceived shortcomings of SELinux, its lack of an administrative model. One major shortcoming remains, the fact that the SELinux policy language is low level, sometimes at an even lower level than system calls. For instance, consider that the removal of a file from a directory is invoked with one system call, `unlink()`, but requires three permissions: `unlink` on the file, and `remove_name` and `write` on the directory containing the file. We are hopeful that our work, by "democratizing" the power of SELinux, will expose this problem to a larger base of users, hence increasing the chances that user-friendly higher level policy languages will be designed on top of SELinux.

[1]It is practically impossible to pre-generate all the possible combinations of labels, except for a trivial number of network packets overlay labeling rules.

6.2.3 Transactions

We would like to extend our administrative model and implementation to support transactions on the security policy. That is, we would like to replace the system of policy modules currently supported by SELinux with the notion of a group of policy modifications done as an atomic unit. Then, we would also like to use the isolation analysis of a transactional system to track the dependencies among policy modifications. This would allow for a seamless support of cascading revoke operations. A cascading revocation of rights is when, upon removal of an administrative permission, all the permissions that were granted based on it are removed from the policy. Cascading revocations could be supported by re-using the dependency analysis performed by the transactional system. The transaction validation, before commit, would also be the natural extension point to enforce custom policy invariants, similar to the work by Fraser and Badger on preserving continuous operation during policy reconfiguration, by preserving high-level policy invariants [167].

LIST OF REFERENCES

LIST OF REFERENCES

[1] Trent Jaeger. *Operating System Security*. Morgan and Claypool, 2008.

[2] Matt Bishop. *Computer Security: Art and Science*. Addison Wesley, 2003.

[3] Clifford J. Berg. *High-Assurance Design: Architecting Secure and Reliable Enterprise Applications*. Addison Wesley, October 2005.

[4] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, volumes I and II, Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA 01731, October 1972.

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

[6] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*, 2003.

[7] Butler W. Lampson. Protection. In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, March 1971. Reprinted in Operating Systems Review, 8, 1, January 1974, pages 18-24.

[8] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, August 1976.

[9] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau. The Flask security architecture: System support for diverse security policies. In *Proceedings of the 8th USENIX Security Symposium*, August 1999.

[10] Willis H. Ware. Security controls for computer systems (U): Report of defense science board task force on computer security. Technical Report Rand Report R609-1, Rand Corporation, for the Office of the Director of Defense Research and Engineering, February 1970. Available at `http://csrc.nist.gov/publications/history/`.

[11] D. Elliott Bell and Leonard J. La Padula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, also referenced as MTR-2997, Mitre Corporation (work commissioned by Electronic Systems Divisions, AFSC, Hanscom Air Force Base), The MITRE Corporation, Box 208, Bedford, MA 01730; Hanscom Air Force Base, Bedford, MA 01731, March 1976.

[12] President of the United States. Executive order 10501 of November 5, 1953: Safeguarding official information in the interests of the defense of the United States, 1953. Available at `http://en.wikisource.org/wiki/Executive_Order_10501`.

[13] President of the United States. Executive order 13526 of December 29, 2009: Classified national security information, 2009. Available at `http://en.wikisource.org/wiki/Executive_Order_13526`.

[14] K.J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-732, also referenced as MTR-3153, Mitre Corporation (work commissioned by Electronic Systems Divisions, AFSC, Hanscom Air Force Base), The MITRE Corporation, Box 208, Bedford, MA 01730; Hanscom Air Force Base, Bedford, MA 01731, April 1977.

[15] W.E. Boebert and R.Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the Eighth National Computer Security Conference*, 1985.

[16] W.E. Boebert, R.Y. Kain, and W.D. Young. The extended access matrix model of computer security. *ACM SIGSOFT Software Engineering Notes*, 10(4):119–125, 1985.

[17] Dan Goodin. Android banking trojan intercepts security texts. *The Register*, September 2011. `http://www.theregister.co.uk/2011/09/14/spyeye_targets_android_phones/`.

[18] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, September 1975.

[19] Jerome H. Saltzer. 2010 National Computer System Security Award, presented at the IEEE 2010 Symposium on Security and Privacy, acceptance speech, May 2010.

[20] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, pages 132 – 140, 1975.

[21] Per Brinch Hansen. *Classic Operating Systems*. Springer, 2001.

[22] Leonard J. LaPadula. *Information Security: An Integrated Collection of Essays*, chapter Essay 9: Rule-Set Modeling of a Trusted Computer System. IEEE Computer Society Press, 1995.

[23] Tim Moses. eXtensible Access Control Markup Language 2 (XACML 2). `http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf`, February 2005.

[24] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. Practical domain and type enforcement for UNIX. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, 1995.

[25] Umesh Shankar, Trent Jaeger, and Reiner Sailer. Toward automated information-flow integrity verification for security-critical applications. In *Proceedings of the 13th Annual Network and Distributed Systems Security Symposium*, February 2006.

[26] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[27] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[28] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1998.

[29] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, MIT, 1999. Technical Report MIT/LCS/TR-783.

[30] Jif: Java + information flow website. `http://www.cs.cornell.edu/jif/`.

[31] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL 99)*, 1999.

[32] Vincent Simonet. FlowCaml website. `http://cristal.inria.fr/~simonet/soft/flowcaml/index.en.html`.

[33] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, January 2003.

[34] Boniface Hicks, Kiyan Ahmadizadeh, and Patrick McDaniel. From languages to systems: Understanding practical application development in security-typed languages. In *22nd Annual Computer Security Applications Conference*, December 2006.

[35] Boniface Hicks, Sandra Rueda, Trent Jaeger, and Patrick McDaniel. From trusted to secure: Building and executing applications that enforce system security. In *Proceedings of the USENIX Annual Technical Conference*, June 2007.

[36] D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, 1987.

[37] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, 1996.

[38] George C. Necula. Proof-carrying code. In *Proceedinggs of the 24th Annual Symposium on Principles of Programming Languages (POPL '97)*, 1997.

[39] T.F. Lunt, D.E. Denning, R.R. Schell, M. Heckman, and W.R. Shockley. The SeaView security model. *IEEE Transactions on Software Engineering*, 16(6), June 1990.

[40] Latanya Sweeney. k-anonymity: a model for protecting privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5):557–570, 2002.

[41] Trent Jaeger and Jonathon Tidswell. Practical safety in flexible access control models. *ACM Transaction on Information and System Security*, 4(2):158–190, 2001.

[42] Jeff Atwood. Coding horror, programming and human factors, regular expressions. `http://www.codinghorror.com/blog/2006/01/regex-performance.html`, January 2006.

[43] David F.C. Brewer and Michael J. Nash. The chinese wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, 1989.

[44] Morrie Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold Company, New York, 1988. Available at `http://www.acsac.org/secshelf/book002.html`.

[45] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.

[46] US Department of Defense. *Trusted Computer System Evaluation Criteria, DoD 5200.28-STD*. National Computer Security Center, National Computer Security Center, Ft. Meade, MD 20755, December 1985. Also known as the "Orange Book", DoD 5200.28-STD, superseded by DoD Directive 8500.1.

[47] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, 1979.

[48] Dennis M. Ritchie. Protection of data file contents. US Patent 4,135,240, 1973. Patent for the UNIX `setuid` facility.

[49] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Shermann, and Karen A. Oostendorp. Confining root programs with domain and type enforcement (DTE). In *Proceedings of the Sixth USENIX UNIX Security Symposium*, 1996.

[50] Butler W. Lampson. Dynamic protection structures. In *Proceedings of the AFIPS Fall Joint Computer Conference*, 1969.

[51] David Ferraiolo, D. Richard Kuhn, and Ramaswamy Chandramouli. *Role-based Access Control*. Artech House, 2003.

[52] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. TRBAC: A temporal role-based access control model. *ACM Transactions on Information and System Security*, 4(3), 2001.

[53] James B. D. Joshi, Elisa Bertino, and Arif Ghafoor. Hybrid role hierarchy for generalized temporal role based access control model. In *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC02)*, 2002.

[54] Rafae Bhatti, Arif Ghafoor, Elisa Bertino, and James B. D. Joshi. X-GTRBAC: an XML-based policy specification framework and architecture for enterprise-wide access control. *ACM Transactions on Information and System Security*, 8(2), 2005.

[55] ANSI. Role based access control, February 2004. Standard INCITS 359-2004.

[56] Ravi Sandhu and Venkata Bhamidipati Qamar Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on on Information and System Security (TISSEC)*, 2(1), February 1999.

[57] Ninghui Li, Ji-Won Byun, and Elisa Bertino. A critique of the ANSI standard on role-based access control. *IEEE Security & Privacy*, November/December 2007.

[58] Robert W. Baldwin. Naming and grouping privileges to simplify security management in large databases. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, 1990.

[59] David Ferraiolo, Rick Kuhn, and Ravi Sandhu. Comments on a critique of the ANSI standard on role-based access control. *IEEE Security & Privacy*, November/December 2007.

[60] P. Pitelli. The Bell-LaPadula computer security model represented as a special case of the Harrison-Ruzzo-Ullmann model. In *NBS-NCSC National Computer Security Conference*, pages 118–121, 1987.

[61] D. Richard Kuhn. Role based access control on MLS without kernel changes. In *3rd ACM Workshop on Role Based Access Control*, 1998.

[62] Gansen Zhao and David W Chadwick. On the modeling of Bell-LaPadula security policies using RBAC. In *Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises – WETICE*, 2008.

[63] M. D. Abrams, L. J. LaPadula, K. E. Eggers, and I. M. Olson. A generalized framework for access control: An informal description. In *Proceedings of the 13th National Computer Security Conference*, 1990.

[64] Amon Ott. *Mandatory Rule Set Based Access Control in Linux: A Multi-policy Security Framework and Role Model Solution for Access Control in Networked Linux Systems*. Shaker Verlag GmbH, Aachen, Germany, 2007. Amon Ott's dissertation on RSBAC.

[65] Sushil Jajodia, Pierangela Samarati, V. S. Subrahmanian, and Elisa Bertino. A unified framework for enforcing multiple access control policies. *ACM SIGMOD Record*, 26(2):474 – 485, June 1997.

[66] David F. Ferraiolo, Serban Gavrila, Vincent Hu, and D. Richard Kuhn. Composing and combining policies under the policy machine. In *SACMAT '05: Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies*, pages 11–20, 2005.

[67] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. A domain and type enforcement UNIX prototype. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, 1995.

[68] Robert Watson, Wayne Morrison, Chris Vance, and Brian Feldman. The TrustedBSD MAC framework: Extensible kernel access control for FreeBSD 5.0. In *Proceedings of the FREENIX Track: 2003 USENIX Annual Technical Conference (FREENIX '03)*, 2003.

[69] G. Scott Graham and Peter J. Denning. Protection – principles and practice. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 40, pages 417–429, 1972.

[70] Dieter Gollmann. *Computer Security*. John Wiley & Sons, Ltd, 1999.

[71] Serge E. Hallyn and Phil Kearns. Domain and type enforcement for Linux. In *4th Annual Linux Showcase and Conference*, 2000.

[72] Peter A. Loscocco and Stephen D. Smalley. Meeting critical security objectives with Security-Enhanced Linux. In *Proceedings of the 2001 Ottawa Linux Symposium*, 2001.

[73] Karl MacMillan, Joshua Brindle, Frank Mayer, Dave Caplan, and Jason Tang. Design and implementation of the SELinux policy management server. In *Proceedings of the 2006 SELinux Symposium*, 2006.

[74] Jason Crampton and George Loizou. Administrative scope: A foundation for role-based administrative models. *ACM Transactions on Information and System Security*, 6(2):201–231, 2003.

[75] Ravi Sandhu and Qamar Muna. The ARBAC99 model for administration of roles. In *Proceedings of the 15th Annual Computer Security Applications Conference*, 1999.

[76] Anita K. Jones. *Foundations of Secure Computation*, chapter Protection Mechanism Models: Their Usefulness, pages 237–254. Academic Press, Inc, 1978.

[77] Ninghui Li and Ziqing Mao. Administration in role-based access control. In *ASIACCS '07: Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, pages 127–138, 2007.

[78] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, AIEE-IRE '62 (Spring), pages 335–344. ACM, 1962.

[79] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics Virtual Memory: Concepts and Design. *Communications of the ACM*, 15(5):308–318, 1972.

[80] Steve Bunch. The setuid feature in UNIX and security. In *Proceedings of the 10th National Computer Security Conference*, 1987.

[81] R. C. Daley and P. G. Neumann. A general-purpose file system for secondary storage. In *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, part I*, AFIPS '65 (Fall, part I), pages 213–229. ACM, 1965. This paper introduces the read, write, execute, and append protection bits.

[82] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7), July 1974.

[83] Andreas Grünbacher. POSIX access control lists on Linux. In *Proceedings of the USENIX 2003 Annual Technical Conference, FREENIX Track*, 2003.

[84] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[85] Simson Garfinkel, Gene Spafford, and Alan Schwartz. *Practical* UNIX *and Internet Security*. O'Reilly, 3rd edition edition, 2003. Contains the quote from Dennis Ritchie about UNIX security: "It was not designed from the start to be secure. It was designed with the necessary characteristics to make security serviceable" (page 23), from an interview with Simson Garfinkel in 1990.

[86] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *SANE 2000*, 2000. This article documents the history and motivation for the `chroot` system call. It also describes `jails` which, contrary to `chroot`, is designed for security.

[87] Alan Cox. Abusing `chroot`. `http://kerneltrap.org/Linux/Abusing_chroot`, September 2007. Summary of a thread on the Linux kernel mailing list, discussing the (in)security of `chroot`.

[88] Poul-Henning Kamp. Rethinking `/dev` and devices in the UNIX kernel. In *Proceedings of the BSDCon 2002 Conference*, 2002.

[89] Steven M. Bellovin. Virtual machines, virtual security? *Communications of the ACM*, 49(10), October 2006.

[90] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *Proceedings of the 6th USENIX Security Symposium*, 1996.

[91] Niels Provos. Improving host security with system call policies. In *12th USENIX Security Symposium*, 2003.

[92] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings Network and Distributed Systems Security Symposium*, February 2003.

[93] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *First USENIX Workshop on Offensive Technologies (WOOT'07)*, August 2007.

[94] Dixie B. Baker. Fortresses built upon sand. In *Proceedings of the 1996 Workshop on New Security Paradigms*, 1996.

[95] D.J. Thomsen and J.T. Haigh. A comparison of type enforcement and UNIX setuid implementation of well-formed transactions. In *Proceedings of the Sixth Annual Computer Security Applications Conference*, pages 304 – 312, December 1990.

[96] W. D. Young, W. E. Boebert, and R. Y. Kain. Proving a computer system secure. *Scientific Honeyweller*, 1985.

[97] Opensolaris project: Flexible mandatory access control (FMAC). `http://opensolaris.org/os/project/fmac/`.

[98] Trent Jaeger, David H. King, Kevin R. Butler, Serge Hallyn, Joy Latten, and Xiaolan Zhang. Leveraging IPsec for mandatory per-packet access control. In *SecureComm 2006*, 2006.

[99] Paul Moore. Transitioning to secmark. http://paulmoore.livejournal.com/4281.html, May 2009.

[100] William R. Cheswick, Steven M. Bellovin, and Aviel D. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley Professional, 2003.

[101] Brad Gough, Christian Karpp, Rajeev Mishra, Liviu Rosca, Jacqueline Wilson, and Chris Almond. *AIX V6 Advanced Security Features Introduction and Configuration*. IBM Redbooks, September 2007. available at http://www.redbooks.ibm.com/abstracts/sg247430.html.

[102] V.D. Gligor, C.S. Chandersekaran, R.S. Chapman, L.J. Dotterer, M.S. Hetch, Wen-Der Jiang, A. Johri, G.L. Luckenbaugh, and N. Vasudevan. Design and implementation of secure Xenix. *IEEE Transactions on Software Engineering*, February 1987.

[103] Sun Microsystems. TrustedSolaris 8 operating environment, a technical overview. http://www.sun.com/software/whitepapers/wp-ts8/ts8-wp.pdf, 2000.

[104] M. D. McIlroy and J. A. Reeds. The IX multilevel-secure UNIX system. Technical Report CSTR #163, AT&T Bell Laboratories, 1992.

[105] Timothy Fraser. LOMAC: Low water-mark integrity protection for COTS environments. In *2000 IEEE Symposium on Security and Privacy*, 2000.

[106] Ninghui Li, Ziqing Mao, and Hong Chen. Usable mandatory integrity protection for operating systems. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2007.

[107] William Enck, Patrick McDaniel, and Trent Jaeger. PinUP: pinning user files to known applications. In *2008 Annual Computer Security Applications Conference (ACSAC 2008)*, 2008.

[108] Novell Inc. AppArmor. http://www.novell.com/linux/security/apparmor/.

[109] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security module framework. In *Ottawa Linux Symposium 2002*, 2002.

[110] Security-Enhanced Linux. www.nsa.gov/research/selinux.

[111] Microsoft Corporation. Access control model. http://msdn.microsoft.com/en-us/library/aa374876%28VS.85%29.aspx, December 2009. accessed December 5th 2009.

[112] Microsoft Corporation. Mandatory integrity control. http://msdn.microsoft.com/en-us/library/bb648648%28VS.85%29.aspx, December 2009. accessed December 5th 2009.

[113] Microsoft Corporation. Role-based access control. http://msdn.microsoft.com/en-us/library/aa379318%28VS.85%29.aspx, December 2009. accessed December 5th 2009.

[114] Sparta Inc. SEDarwin. `http://sedarwin.org/`, 2007.

[115] Apple Inc. `SANDBOX_INIT(3)`. System manual page for OS X 10.5, also available at `http://developer.apple.com/mac/library/DOCUMENTATION/Darwin/Reference/ManPages/man3/sandbox_init.3.html`, July 2007.

[116] Hewlett Packard Inc. *HP OpenVMS Guide to System Security: OpenVMS Version 8.4*, June 2010. `http://h71000.www7.hp.com/doc/84final/ba554_90015/ba554_90015.pdf`.

[117] Maxwell Krohn, Petros Efstathopoulos, Cliff Frey, Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Michelle Osborne, Steve VanDeBogart, and David Ziegler. Make least privilege a right (not a privilege). In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS 2005)*, 2005.

[118] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4), October 1988.

[119] Mark Miller, Ka-Ping Yee, and Jonathan S. Shapiro. Capability myths demolished. Technical Report SRL2003-02, Johns Hopkins University, Department of Computer Sciences, Systems Research Laboratory, 2003.

[120] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th Symposium on Operating Systems Principles (SOSP 2005)*, 2005.

[121] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[122] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, April 2007.

[123] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems (TOCS)*, 12:271–307, November 1994.

[124] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, and Stephen Russell. The Mungi single-address-space operating system. *Software Practice and Experience*, 28(9):901–928, July 1998.

[125] Matunda Nyanchama and Sylvia Osborn. Modeling mandatory access control in role-based security systems. In Demurjian and Dobson, editors, *Database Security IX: Status and Prospects*. Chapman and Hall, 1995.

[126] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in Plan 9. *Operating Systems Review*, 27(2):72–76, April 1993. Reprinted from Proceedings of the 5th ACM SIGOPS European Workshop, Mont Saint-Michel, 1992, Paper number 34.

[127] FUSE: Filesystem in userspace. `http://fuse.sourceforge.net`.

[128] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the USENIX security conference*, 2002.

[129] Trent Jaeger, Antony Edwards, and Xiaolan Zhang. Consistency analysis of authorization hook placement in the Linux security modules framework. *ACM Transaction on Information and System Security*, 7(2):175–204, 2004.

[130] Greg Kroah-Hartman, Jonathan Corbet, and Amanda McPherson. Linux kernel development (April 2008). `http://www.linuxfoundation.org/publications/linuxkerneldevelopment.php`, 2008.

[131] William E. Boebert and Richard Y. Kain. A further note on the confinement problem. In *30th Annual 1996 International Carnahan Conference*, 1996.

[132] D. Sterne. A TCB subset for integrity and role-based access control. In *Proceedings of the 15th National Computer Security Conference*, pages 680–696, 1992.

[133] W.R. Shockley and R.R. Shell. TCB subsets for incremental evaluation. In *Proceedings of the Third Aerospace Computer Security Conference*, pages 131–139, 1987.

[134] George E Forsythe and Niklaus Wirth. Automatic grading programs. Technical Report CS-TR-65-17, Stanford University, Department of Computer Science, February 1965.

[135] Michael D. Schroeder. *Cooperation of Mutually Suspicious Subsystems in a Computer Utility*. PhD thesis, Massachusets Institute of Technology, Department of Electrical Engineering, 1972.

[136] Family educational rights and privacy act (FERPA). 34 C.F.R § 99.1. Available from the US Department of Education at `http://www2.ed.gov/policy/gen/reg/ferpa/index.html`, 1974.

[137] Thomas Ptacek and Timothy Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Available at `http://www.insecure.org/stf/secnet_ids/secnet_ids.pdf`, 1998. Secure Networks, Inc. White paper.

[138] US Department of Defense. *Trusted Network Interpretation. NCSC-TG-005*. National Computer Security Center, 1987. Also known as the "Red Book", provides an interpretation of the Trusted Computer System Evaluation Criteria for networks and network components.

[139] Jonathan M. McCune, Trent Jaeger, Stefan Berger, Ramón Cáceres, and Reiner Sailer. Shamon: A system for distributed mandatory access control. In *Proceedings of the 2006 Annual Computer Security Applications Conference*, December 2006.

[140] NetLabel – Explicit labeled networking for Linux. `http://netlabel.sourceforge.net/`.

[141] IETF CIPSO Working Group. Commercial IP security option (CIPSO 2.2), July 1992. Copies of this expired IETF draft are available on several websites, including NetLabel's website `http://netlabel.sourceforge.net/files/draft-ietf-cipso-ipsecurity-01.txt`.

[142] Netfilter – Firewalling, NAT, and packet mangling for Linux. `http://www.netfilter.org/`.

[143] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, May 1976.

[144] Antoine Vigneron. Computational geometry slides. Available at `http://w3.jouy.inra.fr/unites/miaj/public/vigneron/cs4235/slides.html`, 2004.

[145] Hanan Samet. *Foundations of Multimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[146] Kurt Mehlhorn. *Data Structures and Algorithms 3: Multi-Dimensional Searching and Computational Geometry*. Springer, 1984.

[147] Interval tree. `http://en.wikipedia.org/wiki/Interval_tree`, 2009.

[148] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[149] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry*. Springer, third edition, 2008.

[150] H. Edelsbrunner. A new approach to rectangle intersections, part I and II. *International Journal of Computer Mathematics*, pages 209–229, 1983.

[151] J.L. Bentley. Solutions to Klees rectangle problems. Technical report, Carnegie-Mellon University, 1977.

[152] Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, 1985.

[153] Eric N. Hanson and Theodore Johnson. The interval skip list: A data structure for finding all intervals that overlap a point. Technical Report UF-CIS-92-016, Computer and Information Sciences Department, University of Florida, June 1992.

[154] Mark Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer, 1983.

[155] John Wylie Lloyd. *Foundations of Logic Programming*. Springer, second extended edition, 1993.

[156] NCSA. The common gateway interface. `http://hoohoo.ncsa.illinois.edu/cgi/overview.html`.

[157] Ivan Ristic. *Apache Security*. O'Reilly Media, 2005.

[158] suPHP. `http://www.suphp.org/`.

[159] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[160] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *Proceedings of 11th International Symposium on Recent Advances in Intrusion Detection (RAID 2008)*, September 2008.

[161] Common vulnerabilities and exposures (CVE). `http://cve.mitre.org/`.

[162] Hong Chen, Ninghui Li, and Ziqing Mao. Analyzing and comparing the protection quality of security enhanced operating systems. In *6th Network and Distributed System Security Symposium (NDSS)*, 2009.

[163] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium*, August 2010.

[164] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of the Twenty-First ACM SIGOPS Symposium on Operating Systems principles (SOSP 2007)*, 2007.

[165] Peter A. Loscocco and Stephen D. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01)*, 2001.

[166] Mary Ellen Zurko, Rich Simon, and Tom Sanfilippo. A user-centered, modular authorization service built on an RBAC foundation. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.

[167] Timothy Fraser and Lee Badger. Ensuring continuity during dynamic security policy reconfiguration in DTE. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, 1998.

APPENDIX

APPENDIX: NETFILTER

The Linux netfilter packet manipulation framework includes more features than what we presented in Chapter 4. For instance, netfilter supports network address translation (NAT), and the manipulation of link-level packet (ebtables). For reference, we include the reference diagram of packet flow inside netfilter (see Figure A.1). This diagram illustrates the different hooks that the netfilter framework offers for customizing packet processing.

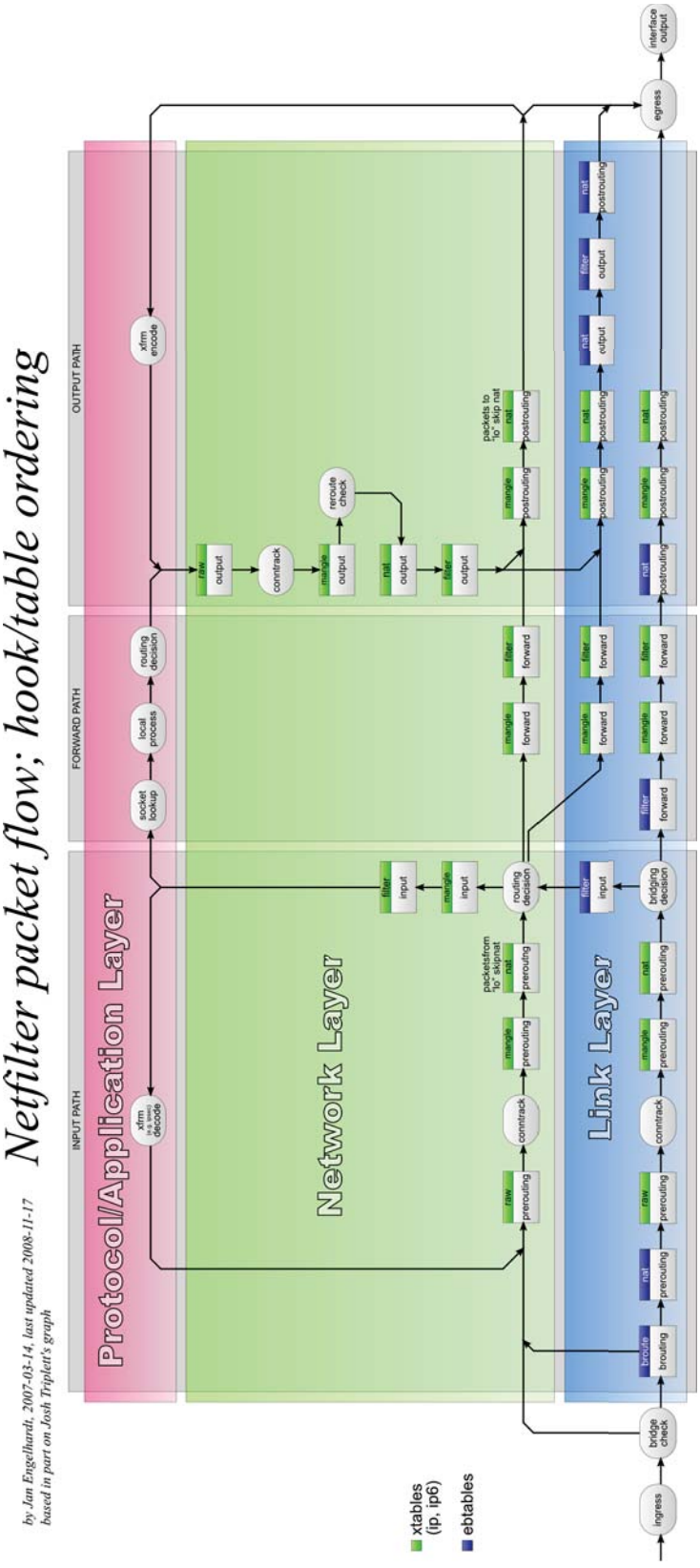


Figure A.1.: Packet flow inside the netfilter framework

VITA

VITA

Jacques Thomas studied first at Université Pierre et Marie Curie in Paris, where he obtained a Bachelor of Science degree in 1999 and a Master of Science degree in computer science in 2002. The Master of Science degree was obtained while attending the Magistère d'Informatique Appliquée d'Ile de France program (MIAIF) within the University. Jacques Thomas then entered the Ph.D. program in the Purdue Computer Science department in 2003. He as been working at Amazon since 2010.