TEpla: A Certified Type Enforcement Access-Control Policy Language

Amir Eaman

Thesis submitted to the University of Ottawa
in partial fulfillment of the requirements for the Ph.D. degree in
Computer Science

Ottawa-Carleton Institute for Computer Science
School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

In today's information era, the security of computer systems as resources of invaluable information is of crucial importance not just to security administrators but also to users of these systems. Access control is an information security process which guards protected resources against unauthorized access as specified by restrictions in security policies. One significant obstacle to regulate access in secure systems is the lack of formal semantics and specifications for the policy languages which are used in writing security policies.

Expressing security policies that are implemented pursuant to required security goals and that accommodate security policy rules correctly is of high importance to the system's integrity, confidentiality, and availability. The semantics of the most widely used policy languages such as SELinux is expressed in a declarative manner using a colloquial natural language (e.g., English), which leads to ambiguity in the interpretation of the policy statements. For this reason, both the development and the analysis of security policies are generally imprecise and based on cognitive concepts; that is to say, they are not conducted in a mathematically-precise and verifiable way.

Type Enforcement (TE) is a MAC (Mandatory Access Control) access control mechanism that is used in the SELinux security module. Type Enforcement (TE) is implemented based on the type/domain field of security contexts. TE allows the creation of different domains in the system by assigning subjects to domains and subsequently associating them with objects. TE mandates a central policy-driven approach to access control.

We propose a small and certifiably correct TE policy language, TEpla, as an appropriate candidate for the primary access control feature of SELinux, Type Enforcement. TEpla can provide ease of use, analysis, and verification of its properties. TEpla is a certified policy language with formal semantics, exposing ease of reasoning and allowing verification. We use the Coq proof assistant to mechanize semantics and to machine-check the proofs of TEpla, ensuring correctness guarantees are provided. Having a certified semantics simplifies and fosters the development of certified tools for policy-related tasks such as automating various kind of policy analyses.

Acknowledgments

I would like to express my special thanks of gratitude to my thesis advisor, Dr. Amy Felty, for her support and encouragement throughout my Ph.D. studies and all the people who made this possible.

Dedication

This thesis is dedicated to my father, Asadolah, who as a high-school teacher taught me the value of hard work and perseverance and to my mother, Parvin, who dedicated her life to her children, and to people who seek knowledge.

Table of Contents

List of Tables

List of Figures and Listings

Acronyms

ABAC Attribute-Based Access Control. 7, 8

AC Access Control. 15, 16

AV Access Vector. 11

AVC Access Vector Cache. 9

CXACML Core XACML. 47, 51

DAC Discretionary Access Control. 7–9, 19

IT Information Technology. 4, 5, 7, 22

LSM Linux Security Module. 2, 9

MAC Mandatory Access Control. 2, 7, 9, 13, 14

MLS Multi-Level Security. 2, 10–12, 19

NSA National Security Agency. 9

RBAC Role-Based Access Control. 2, 7, 8, 10, 12

SEAndroid Security Enhancements for Android. 13, 14, 18

SELinux Security-Enhanced Linux. 2–7, 9–14, 16–23, 31, 34, 47, 48, 103, 116, 117

SoD Separation of Duty. 13, 14, 37, 43, 44, 101, 102

TCB Trusted Computing Base. 15–18

TE Type Enforcement. 2, 4, 5, 10–12, 19, 34, 57, 114–117

UML Unified Modeling Language. 3

Chapter 1

Introduction

In the domain of information security, access control techniques are important for protecting resources from unauthorized access. Access control as a security mechanism is concerned with the management of access requests to resources. To determine if a request is allowed, it is checked against a set of authorization rules which are written in a particular policy language dependent on the type of access control available in the underlying computer system. Accordingly, access control policy languages have a crucial role in expressing the intended access authorization to regulate requests to resources.

1.1 Overview

Security administrators define security specifications to express the security access requirements of a system. Security policy languages used to develop security policies significantly affect this process. This is because the policy developers' understanding of the semantics of the languages has a direct effect on the way they write policies. In addition, security administrators utilize policy-related tools to help with policy development as well as to validate that security specifications precisely reflect their intended security requirements. That is, these tools can often determine the presence of flaws in the security policies that are contrary to the security objectives. Similar to how the security administrators' understanding of the policy language semantics affects the development of their policies, the tool writers' understanding of the semantics influences the development of their tools. This is an important indirect effect of the understanding of semantics on policy development. In other words, the semantics of policy languages have an influence not only on the way policy writers develop policies but also on the way policy-related tools facilitate policy development.

On the other hand, most policy languages do not have a fully formal semantics. Inconsistencies and contradictions are possible among specifications of the same policy language, often caused by the informal descriptive text that is used to relay the meaning of the semantics to language users (i.e., the colloquial nature of the description of the semantics of

the policy language because it is given in natural language). The specifications defined in these policy languages are often prone to ambiguous interpretations. Consequently, informal semantics of a policy language has a clear adverse impact on the correct understanding of the policy language, as using the language crucially depends on the understanding of individual policy developers of the semantic description of the language.

There are different types of access control mechanisms that are supported by policy languages. In this thesis, we focus on developing a new certified policy language, TEpla, for the Type Enforcement (TE) mechanism, which is a subset of the SELinux security module [National Security Agency, 2019] implemented in Linux distributions. In particular, we primarily focus on just one of the most important aspects of SELinux, Type Enforcement, and focus on the certification of this aspect. By *certified* policy language, we mean a policy language with formal semantics and mathematical proofs of important properties, which reflects the concept of certification in formal methods communities and programming languages [Chlipala, 2019]. In other words, the notion of certified for TEpla means having a mathematically defined semantics along with encoding it in a formal system and proving properties where proofs are *certificate*s that can be independently checked. Formal semantics of TEpla is mechanized in the Coq proof assistant [Bertot and Castéran, 2004, The Coq Development Team, 2019]. Developing a new access control language that focuses on Type Enforcement and builds in certification from the beginning as a central goal is a solution that we proposed after conducting a comprehensive study of the SELinux policy language [Eaman et al., 2017]. This survey was an important first step of our work.

SELinux is a Linux Security Module (LSM) that enables security developers to define security policies. Since Linux is widely used, and security is very important, the SELinux policy language plays a central role in access control. It implements the Mandatory Access Control (MAC) strategy, which allows policy writers to express whether a *subject* can perform an operation on an *object*. Thus, MAC security policies provide the central access control mechanism for enforcing security restrictions in SELinux.

The SELinux policy language encompasses an integration of Role-Based Access Control (RBAC), Type Enforcement (TE), and Multi-Level Security (MLS) [Mayer et al., 2006]. Partly as a result of the fact that its semantics are defined informally and partly due to the way the language has been designed, both SELinux security policy development and policy analysis are challenging tasks. In the study mentioned above [Eaman et al., 2017], we found that virtually all of the drawbacks are caused by ambiguity in the specification or the behavior of policy statements of the SELinux policy language. This problem stems from the SELinux policy language's lack of formal semantics and from not having any formally verified properties.

1.2 Motivation and State of the Art

In the study mentioned above [Eaman et al., 2017], we wanted to understand how informal semantics of a policy language can affect developing security policies, in particular, policies

developed using SELinux. This study includes studying SELinux policy-related tools (a large set) as well as evaluating the SELinux policy language with respect to formal language properties presented in [Tschantz and Krishnamurthi, 2006] to see if it satisfies properties claimed to be important. We found that the tools try to address problems and are beneficial for analyzing policies, but are limited by informal semantics and the complexity of the SELinux policy language. Furthermore, we found that SELinux does not satisfy all of the properties in [Tschantz and Krishnamurthi, 2006]. However, it is not clear that all of the properties as they are defined in [Tschantz and Krishnamurthi, 2006] are desirable. Our analysis led to an improved statement of the *Safety* property.

Consequently, following our study and the fact that understanding and analyzing SELinux policies are subject to different interpretations, as mentioned above, we proposed a potential solution which included designing TEpla, which is a new language with formal semantics. As a result, one of the main goals of the research is to show how formal semantics of a policy language can be used in proving particular behavior and properties of the language. To this end, we begin by exploring current challenges in access-control policy languages, such as SELinux policy language. We then describe and formalize TEpla with supporting proofs of several interesting properties. The language is then illustrated on a policy composed of 20 rules and a few constraints (and introducing many utility functions along the way), with the checking of several access requests.

Different studies of the SELinux policy language, such as [Quigley, 2007, Kuliniewicz, 2006], have been carried out in an attempt to put forward some possible solutions to the challenges mentioned above of the SELinux policy language. These solutions define other intermediate policy languages in order to express policies, which are then translated to the SELinux policy language. However, the proposed solutions, which aim to help users to specify SELinux security policies, are not complete solutions because the definitions of the intermediate languages still depend on the language designers' understanding of SELinux semantics. Moreover, the results of policy-related tools are not reliable since the implementation of such tools cannot be proven to agree with the semantics.

1.3 Benefits of Using Formal Semantics

Using mathematical and formal techniques and tools has proved its value in many different domains, such as verifying compilers and database systems [Pierce et al., 2008, Leroy, 2012, Benzaken et al., 2014]. Studies including [Abbas et al., 2018, Sengupta and Bhattacharya, 2008, Hausmann, 2005], for example, formalize various parts of Unified Modeling Language (UML) to facilitate validation and verification of software design. Formal semantics can tremendously improve the use of the language by constructing a precise reference for the underlying language. Formal semantics for a language thus provides a common understanding for the language users. Semantic-related tools which analyze or reason about specifications written in the language require formal semantics to process the language correctly. Moreover, as was mentioned earlier, the implementation of such tools can be

verified according to the formal semantics of the language, and thus verifying the results of the tools is another important consequence of formal semantics of TEpla.

As discussed, TEpla is a certified Type Enforcement policy language in terms of formal semantics, which enables policy developers or tools to rely on a language which behaves exactly as prescribed by the semantics. TEpla formally satisfies a particular set of properties that provide ease of reasoning and analysis. We stress here that the novelty of TEpla as compared to other policy languages that try to enhance SELinux policy language in terms of ease of reasoning, analysis, and so forth, is that it provides machine-checked proofs and formal semantics, and thus the language is not prone to different interpretations.

1.4 Objective and Contributions of this Thesis

The main contribution of this thesis is the design of a certified TE policy language (TEpla) with formal semantics and the proof of some important properties of TEpla. Here we state the contributions of this thesis in more detail.

- Root cause analysis of challenges of the SELinux policy language as an informal policy language and proposing a solution [Eaman et al., 2017]:
  Analyzing and evaluating SELinux policy language for identifying the root causes of problems related to policy development and proposing a solution (a new certified policy language) with the potential to lead to significant improvements.

- Designing a certifiably correct TE policy language:
  Designing a certified TE policy language to avoid language-introduced errors (i.e., errors that are introduced to IT systems due to multiple contradictory interpretations of policies) is one of the merits of TEpla. Policy writers can write security goals in TEpla as a high-trust policy language which guarantees that the introduced security goals are exactly as the specification in the policies; therefore, TEpla provides a TE policy language to IT security administrators to correctly express security goals that they want to implement.

- Designing a certified policy language which provides defining *predicates on-demand* (PoD):
  TEpla is flexible enough for defining complex security constraints through taking various user-defined predicates. This enables security administrators to define various security goals in security policies. This feature gives more expressive power to constraints in TEpla, in comparison to constraints in the SELinux policy language, for example, the set of TEpla constraints that we develop in Chapter 6 is expressible in TEpla, but not SELinux.

- Providing formal semantics for the TE policy language:
  Having a formal semantics allows policy writers to write formally verified security policies.

- Providing ease of reasoning and analysis for TE policies:
  Ease of reasoning or analysis of TEpla is guaranteed by a clear specification of TEpla's behavior and semantics as it satisfies most of the formal properties designed for this purpose [Tschantz and Krishnamurthi, 2006]. The *Non-decreasing* property of TEpla policies (discussed in Chapter 3), for example, provides a way to group queries based on their possible decisions. Having a common understanding of the language, through a formal semantics, helps to reach this end as well.

- Providing the initial step to the development of certified policy-related tools:
  Existing policy analysis tools, for example, are not certifiably correct due to the fact that they do not apply formal methods to their analysis of policies. A language such as TEpla lays the foundation for developing certified analysis tools or automated policy generating tools for policies expressed in TEpla.

- Providing an innovative proof of concept for developing a certified Type Enforcement policy language through formal methods:
  Establishing overall concepts for developing a certified TE policy language is provided. Following the same development procedure, researchers can develop certified formal languages and tools in different policy languages that are used in IT systems. These languages can formally support specific formal properties that are related to particular IT domains, such as semantic web or distributed systems, and provide facilities for the related language requirements.

1.5 Structure of this Thesis

Chapter 2 introduces the background for SELinux and the Type Enforcement access control mechanism. This chapter presents the challenges of the SELinux policy language as well as the details of our evaluation of policy analysis tools. In addition, we introduce the Coq proof assistant and some of its language features.

In Chapter 3, we design the infrastructure of TEpla consisting of syntax, semantics, and proofs that are used for developing TEpla. The syntax of TEpla is described by a BNF grammar. We develop some algorithms to specify TEpla semantics.

In Chapter 4, the Coq implementation of TEpla is described. We formalize the syntax and semantics of TEpla. This chapter provides more details on how policy developers can define predicates. Furthermore, we prove a set of formal language properties of TEpla in Coq.

In Chapter 5, we use TEpla to write a sample TE policy. We define two predicates and use them in TEpla constraints.

Finally, Chapter 6 concludes and presents future work.

Chapter 2

Background

This chapter explains access control and the SELinux policy language. The SELinux architecture and the main language constructs related to Type Enforcement are discussed. We then evaluate the SELinux policy language with a set of language properties that assess ease of reasoning and analysis. We choose the SELinux policy language because it employs Type Enforcement in its access decision-making process and it has informal semantics. Finally, this chapter introduces the Coq proof assistant, which we use in Chapter 4 to implement TEpla and prove properties about it.

2.1 Overview

Granting or rejecting access to resources is a crucial aspect of computer systems and vital for their security. Access control can be described as a security service that guards protected resources against unauthorized access while enabling access to authorized consumers [Bishop, 2002]. Security policies are expressed as a set of rules written in an access control policy language. Access requests are evaluated against a security policy to determine if the request is authorized or not. SELinux is an access control security module in Linux which allows administrators to develop security policies. Studying the SELinux policy language gives valuable insight into how informal semantics negatively influences SELinux policy development and analysis. Different SELinux policy-related tools, for example, are developed to help SELinux policy writers, however, there are limitations of all these tools because of the informal semantics, as discussed in Section 2.6.

In other words, while it is true that these tools are beneficial in developing or analyzing SELinux policies, their implementations cannot be validated to ensure that they are correct, as they are not based on a common formal understanding of what the SELinux policy language is. Because these tools are not amenable to formal reasoning and analysis, we do not work further with any tools beyond our thorough analysis. Our next step (in Section 2.7) is to analyze the SELinux language with respect to a set of mathematical properties.

We found that even this kind of analysis is too informal and SELinux does not satisfy all the properties. The properties we choose come from [Tschantz and Krishnamurthi, 2006, Tschantz, 2005], who argue that they are desirable properties for any policy language. Our analysis also led to a critique of one of the properties.

2.2 Access Control

Access control, as an IT security service, deals with three primary entities in a system: *Subject*s that require access to resources, *Object*s or resources that are accessed by subjects, and *Action*s that are performed by subjects on objects. Actions can range from being as simple as reading the data, sharing the data, or executing a file [Mayer et al., 2006]. The final protected system must satisfy information security measures, which consist of confidentiality, integrity, and availability (CIA triad).

2.3 Access Control Models

Access control models define the structure and language for describing system policies and relevant procedures for processing them. Four widely used models for different access control policy types include: Discretionary Access Control (DAC), Mandatory Access Control (MAC), Role-Based Access Control (RBAC), and Attribute-Based Access Control (ABAC) [Stallings and Brown, 2018]. We describe each of these access control model briefly as follows.

The traditional DAC model relies heavily on user identity, which can lead to a compromise of the whole system in the case when the attacker obtains root privileges on the system. The owner or creator of objects grants privileged access to objects. DAC defines an access control list for every object in the system. The only two kind of user identities are *admin* (i.e., owner) and *non-admin* (i.e., users who are not the owner). DAC-based systems are thus trivially coarse-grained; it is not possible to provide fine-grained controls using only identities of users as the basis of decisions.

MAC overcomes the drawbacks of DAC that result from restricting access to objects solely on user identity by introducing many resources and abstracting them into subjects and objects through security attributes. Security attributes are characteristics that define a set of properties of resources. In fact, MAC has more expressive power than any of the other access models because it introduces a larger set of security attributes, and the user can introduce new ones.

Different MAC security models target the preservation of different security objectives in the system, provided by defining security rules as their access control policies. Three important security models for MAC include the *Bell-LaPadula* (BLP) model preserving

confidentiality, the *Biba* model preserving integrity, and the *Clark-Wilson* model preserving integrity [Stallings and Brown, 2018]. We describe each in more detail below.

The Bell-LaPadula model ensures confidentiality of information by not allowing a subject to write objects of lower security level and not allowing a subject to read objects of higher security level. BLP security rules restrain the transfer of information from a higher security level subject to a lower security level object in a system.

The Biba integrity model protects the integrity of information by enforcing a policy defined by particular security rules. These security rules include rules that do not allow a subject to read objects of a lower integrity level and do not allow a subject to write objects of a higher integrity level.

The Clark-Wilson model focuses on the integrity of information and uses four security categories as the language for defining access control policy rules [Bishop, 2002]. Policy rules control the integrity of the system by ensuring the integrity of the security categories of the model. These categories are:

- Constraint Data Items (CDIs): objects that are integrity protected.
- Unconstrained Data Items (UDIs): objects that are not integrity protected.
- Integrity Verification Procedures (IVPs): verifiers to check CDI integrity.
- Transformation Procedures (TPs): certified procedures to transition CDIs or UDIs to other CDIs. TPs are supposed to be a filter to control information transfers from low or high integrity objects to high integrity objects.

The third access control model is RBAC which maps users to roles. Roles are sets of authorized permissions that are assigned to users. Thus, RBAC lumps users together in bunches and assigns permissions to these groups, thereby a user can have a specific permission if the permission is assigned to the role that is associated with the user. Finally, the ABAC access control model regulates access by evaluating different properties of entities as well as environmental conditions.

2.4 SELinux Overview

On Linux based systems, many security exploits attempt to target system daemons that often run with elevated or even unlimited privileges (e.g., as root). Once the attacker gets access to a daemon, the whole system is compromised since the attacker obtains permanent root privileges on the system. The traditional Discretionary Access Control (DAC) mechanism that Unix/Linux systems use leaves important security decisions up to the discretion of the individual users and administrators, resulting in an ad-hoc system where

some applications or daemons are well configured whereas others have too many unnecessary permissions. SELinux is a Linux-based access control framework developed by the United States National Security Agency (NSA) [National Security Agency, 2019]. SELinux is compiled into the kernel and supported through the Linux Security Module (LSM). LSM is a kernel-level security framework that provides the possibility of attaching various security mechanisms to the Linux kernel, such as SELinux, without directly depending on the kernel objects. SELinux implements the MAC model within Linux-based distributions and provides more granular control of security.

SELinux primarily involves labeling that divides system resources into subjects, which are processes, and objects, such as files and sockets. In SELinux, every system resource receives a label which is a combination of values of the user, role, type, and security level attributes. These values form the *security context* of system resources.

MAC, and in particular SELinux, mandates a central policy-driven approach to access control and regulates DAC's access decisions. SELinux works based on the principle of least privilege, and every grant of access must have the corresponding allow rule in the security policy to permit that access. This means that when DAC allows access to a subject, the access request still needs to be checked by MAC as well. If DAC denies an access request, MAC will not get involved.

2.4.1 SELinux Architecture

The SELinux security module implements the Flask architecture in a Linux environment [Loscocco and Smalley, 2001]. A feature of the Flask architecture is the separation of security policy logic from the enforcement mechanism. The Security Server is a kernel component responsible for making security decisions, and the Object Manager enforces these security decisions in the system. The Access Vector Cache (AVC), is another component of the SELinux architecture. AVC stores the security policy look-up results to improve the performance of the decision-making procedure. Searching the AVC is faster, so access requests that have been previously processed can be quickly answered without searching the entire security policy again.

Fig. 2.1 depicts the core decision-making architecture of SELinux, subjects send each access request (i.e., query) to the Object Manager who looks in the Access Vector Cache for information about the decision for the query. If the Object Manager fails to answer the query by searching for access information in AVC (i.e., a cache miss happens), then the Security Server has to determine the answer for the new query by evaluating the request against the security policy. This answer determines whether or not the subject can grant access to the intended object or not. The attributes used to determine the decisions of the SELinux access control mechanism are described in the following section.

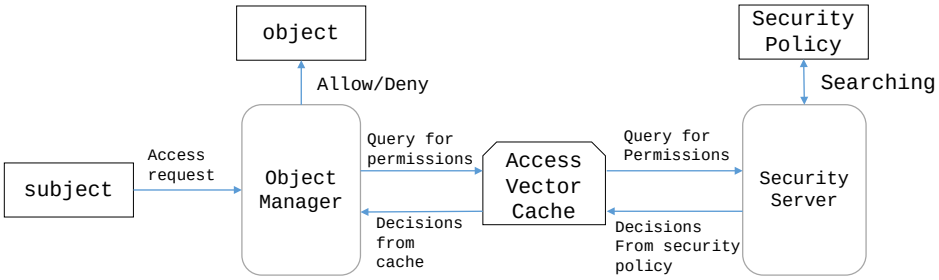

Figure 2.1: Core decision-making architecture in SELinux

2.4.2 SELinux Access Control Criteria

Labeling is the main functionality of SELinux with the goal of labeling all system resources with a proper security context. SELinux primarily focuses on Type Enforcement (TE) related to the type/domain field of security contexts. TE allows the creation of different domains in the system through assigning subjects to domains and subsequently associating them with objects. All of these authorized associations are stated in a SELinux security policy by using TE rules. In addition to TE, SELinux allows the expression of restrictions on the other fields of the security context.

The SELinux attributes include *user*, *role*, *type* or *domain*, and *security level*, each described below.

SELinux introduces its own user attribute, and the Linux user attribute is mapped to the SELinux one [Mayer et al., 2006]. The mapping of Linux users to SELinux users can be viewed using the Linux shell command "`semanage login l`."

The role attribute comes from RBAC. The SELinux security policy determines which users are authorized for each role. In particular, these roles are used for making role-based access control decisions. They specify which domains are authorized for which users and thus permit user entry into these domains. The Linux shell command "`seinfo -r`" lists roles that are available in the system.

The SELinux type attribute is the most important attribute within a security context. Terminologically, to help distinguish subjects and objects, types and domains mean the same thing, but domains classify subjects, while types classify objects. Note that in TEpla, the type attribute will be the only element of the security context.

The SELinux security level attribute is the final attribute in the security context. This attribute is used only in the Multi-Level Security (MLS) access control mechanism, which is not the main policy type of the SELinux access control framework. MLS policies use the security level attribute for expressing rules that restrict access requests, which makes it a suitable access control scheme for military type environments. The Biba integrity and

Bell-LaPadula models are based on MLS. SELinux can be loaded into the Linux kernel without accommodating MLS [Guttman et al., 2005].

2.5 The SELinux Security Policy Language

A SELinux security policy is a collection of statements that defines the threshold for accepting an access request. SELinux denies interaction of subjects and objects by default, in particular, with an empty SELinux policy every access request will be denied. Listing 2.2 lists important security policy rules in SELinux syntax.

```
allow SourceDType TargetType : class1 {perm1 perm2};
type_transition SourceDomain TargetType: class1 new_type;
type_transition SourceDomain TargetType: process new_type;
constrain classobject_list permission_list B(t1,r1,u1,t2,r2,u2)
```

Listing 2.2: Sample rules of a SELinux security policy

Here we describe some main rules of the SELinux policy language related to TE, and to user and role components of the security context.

Type Enforcement (TE) Rules of SELinux mainly include two kinds of rules [Mayer et al., 2006]: Access Vector (AV), and Type Rules, which consist of Object Transition Rules and Domain Transition rules. Access Vector (AV) rules allow, audit, or deny interaction between two types. AV rules include `allow`, `dontaudit`, `auditallow`, and `neverallow` statements [Loscocco and Smalley, 2001]. For example, consider the AV rule in listing 2.2 appearing on the first line. This rule allows the process with domain `SourceDType` to have actions `perm1` or `perm2` on the object of type `TargetType` and object class of `class1`. An object class specifies a possible instance of all resources of a certain kind, such as files, sockets, and directories.

Object Transition Rules in SELinux can be used to specify the type of objects that will be created at run-time. For example, consider the object transition rule on the second line in listing 2.2. This type transition means objects of type `TargetType` that are newly created by a process with the domain of `SourceDomain` will take the *default type* `new_type` instead of `TargetType`. The object class `class1` specifies the object category of `SourceDomain` and `new_type`.

Domain Transition Rules change the domain of a subject to a new domain. For example, consider the domain transition rule on the third line in listing 2.2. This domain transition states that if a process of the domain `SourceDomain` executes a file with the type `TargetType`, the new domain of the process will be `new_type`.

SELinux policies also include constraints. Software developers use constraints to introduce new criteria for granting access requests to objects. Constraints can refine an

```
require {
attribute domain;
attribute file_type;
attribute exec_type;
type sysadm_t;
attribute sysadm_r;
class process transition;
role sysadm_r; }
```
Adding types and attributes that are required by the rules

```
type app_t;
typeattribute app_t domain;
type app_exec_t;
typeattribute app_exec_t file_type;
typeattribute app_exec_t exec_type;
```
Declaring new types and classify them by attributes

```
role sysadm_r types app_t;
```
Assigning roles to types

```
type_transition sysadm_t app_exec_t : process app_t;
allow sysadm_t app_exec_t : file {getatr execute};
allow app_t app_exec_t : file entrypoint;
allow sysadm_t app_t : process transition;
```
Defining default transition and its required access

Listing 2.3: App program security policy rules in SELinux

explicitly allowed access request through enforcing extra considerations for certain users, roles, and types in the decision-making process of the access, expressed as boolean conditions. For example, consider the fourth line in listing 2.2. `B(t1,r1,u1,t2,r2,u2)` is a boolean expression expressing constraints on the type, role, and user of the source entity security context `(t1,r1,u1)` and target entity security context `(t2,r2,u2)`. This constraint defines the requirements under which the operations in `permission_list` are allowed for the class objects in `classobject_list`. If these requirements are not met by an access request, the operations in `permission_list` will be denied.

The policy language that is used to develop SELinux policies is a complex language consisting of a combination of RBAC, TE, and optionally MLS rules. As mentioned, SELinux policies typically include thousands of policy statements, which makes development and analysis of SELinux policies quite difficult. SELinux policy language statements enable security administrators to configure the required permissions for accesses. Sample policy rules for an application (called App here) are shown in listing 2.3. These rules define a single domain entry to execute App through a domain transition.

2.6 Survey of SELinux Policy Analysis Tools

2.6.1 Taxonomy for SELinux Policy Analysis Tools

Different studies have been carried out on the SELinux policy language, trying to put forward some possible tools for helping policy writers write policies that are more easily understood and reasoned about. Languages such as Lobster [Hurd et al., 2009], Seng [Kuliniewicz, 2006], Please [Quigley, 2007], and CDSFramework [Sellers et al., 2006] are intended

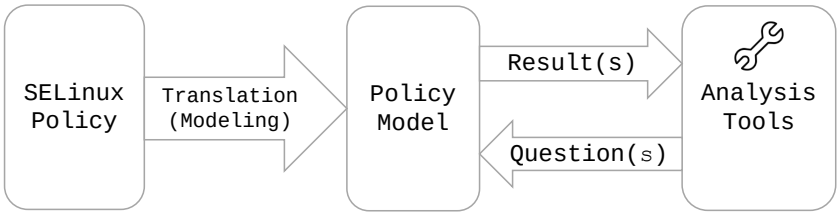

Figure 2.4: Typical structure of SELinux analysis tools

to enhance the SELinux policy language by providing easier syntax and more language features, such as defining object-oriented policy syntax, for example. Despite their attempt to help users to specify SELinux security policies, these languages give rise to limited results that cannot be verified, due to a lack of formalized definition of semantics and language behavior, which results in potentially contradictory interpretations and precludes correct reasoning. These issues contribute to the ongoing development of numerous policy-related tools that try to model SELinux policies without proving the correctness of the results and analyses, as each tool attempts to cover more features rather than verifying their properties and results.

Among existing tools, some are developed while others are at a prototype stage. The typical structure of policy analysis tools is demonstrated in Fig. 2.4. The complexity of the SELinux policy language makes analyzing SELinux policies and even implementing policies very difficult. As a result, virtually all analysis tools and studies that try to develop efficient policy model for SELinux (such as [Zook, 2016]) provide some kind of other intermediate languages or models for SELinux security administrators, as shown in the figure. Note that analysis tools, such as [Radhika et al., 2018], for Security Enhancements for Android (SEAndroid), which is an Android port of the SELinux MAC mechanism [Reshetova et al., 2015], use the same structure as SELinux analysis tools demonstrated in Fig. 2.4 to analyze SEAndroid policies.

It is useful to evaluate SELinux tools in terms of their techniques and capabilities. Table 2.1 compares eighteen SELinux analysis tools. The comparison considers features and techniques utilized in the tools. The table shows that different analysis tools have different capabilities in terms of providing safety, completeness, integrity, and Separation of Duty (SoD) analyses (see the description of these analyses at the end of this section; they are the first four columns of the table). The other features that are compared in Table 2.1 are browsing a policy, rewriting a policy, and building customized queries. The analysis tools employ various forms of query language syntax to allow security administrators to make queries for checking specific properties of the security policy. Various techniques are utilized as methods of analysis; they model the security policy with well-known concepts such as mathematical sets [Zanin and Mancini, 2004], information visualization [Marouf and Shehab, 2011, Xu et al., 2009], and computer security models [Amthor et al., 2011].

Some analysis methods expand all macros, while some perform on-demand expansion of macros [Archer et al., 2003] in the policies. SELint [Reshetova et al., 2017] goes further and replaces policy rules with proper macros of the policy rules, which provides the capability to suggest improvements. The last three tools in Table 2.1—SEAL, EASEAndroid, and SELint—are for analyzing Security Enhancements for Android (SEAndroid), which is an Android port of the SELinux MAC mechanism [Reshetova et al., 2015]. Because most of the tools in Table 2.1 are not available publicly, the information provided here is based on the studies conducted for the tools as presented by the authors. This analysis also appears in our paper [Eaman et al., 2017]. We describe these tools in more details in Section 2.6.2.

We describe the columns of Table 2.1 in more detail. *Safety Analysis* means that policies provide authorized access requirements for a particular entity, such as making sure that only authorized resources reach the entity. *Completeness Analysis* indicates whether or not a resource has all the required accesses as specified by its requirements. *Integrity Analysis* indicates whether or not actions to or from categories of resources are authorized. *SoD Analysis* indicates whether or not entities that perform different actions on the same object are different from each other. *Information Flow Analysis* indicates whether or not there is any access between two entities. *Method of Analysis* describes the analysis algorithms that tools use to analyze policies. These methods are applied to the policy models that each tool has developed (see Fig. 2.4). *Policy Browsing* is about whether or not tools can display different rules of policies. *Policy Rewriting* represents the ability of tools to modify policies. *Method of Modeling* means the methods that tools use to model SELinux policies so that they can apply their analysis algorithms (see Fig. 2.4). *Query language* is the language that tools exploit to question their policy model of SELinux policies. *Macro Expansion* indicates the capability of tools to provide a facility for realizing the rules that can be derived from SELinux macros in policies.

2.6.2 Other SELinux Policy Tools and Languages

In this section, we briefly describe the tools and their related languages which are listed in Table 2.1.

APOL [Tresys Technology, 2019] is a member of the SETools suite [Mayer et al., 2006]. A user loads a SELinux security policy file or a compiled binary policy file to APOL to begin the analysis procedure. By loading the policy file, the user can select attribute items from enabled lists, which are loaded according to the rules in the SELinux security policy file. Then, the user can use regular expressions to specify a search in several analysis modules for particular attributes. A great number of SELinux analysis tools (e.g., [Xu et al., 2013, Xu et al., 2009, Reshetova et al., 2015]) use APOL libraries for their development and often a comparison of the ease of use as compared with APOL is carried out.

Guttman and Herzog [Guttman et al., 2005] describe a four-step procedure used in the SLAT tool for verifying security goals in SELinux configurations. These steps include

Table 2.1: Analysis tools for SELinux security policies

Analysis Tool	Safety Analysis	Completeness Analysis	Integrity Analysis	SoD Analysis	Information Flow Analysis	Method of Analysis	Policy Browsing	Rewriting the Policy	Method of Modeling	Query Language	Macro Expansion
APOL	✓		✓	✓	✓	Information Flow	✓		Syntactic Analysis	Selecting attributes from menus	✓
SLAT	✓	✓			✓	Information Flow–Model Checking			XSB Logic	✓(Regular Expressions)	✓
XcelLog	✓	✓			✓	Information Flow– Deductive Spreadsheets			Sets of Values	Writing Set Formulas	✓
GOKYO	✓	✓	✓		✓	AC Spaces– TCB			AC Spaces – Graphcal AC Model (Sets)	✓	✓
PAL	✓	✓	✓	✓	✓	Logic Programming			XSB Logic	✓	
SELAC	✓				✓	AC Spaces– Mathematical Set			25 Sets		✓
SPTrack	✓				✓	Data Visualization			Graphs		✓
SEGrapher	✓			✓	✓	Data Visualization – Clustering			Graphs– Clustering of Nodes	Selecting types from Menus	✓
SEAnalyzer	✓			✓	✓	Colored Petri Nets			Diagrams & Sets	✓	✓
LOPOL	✓				✓	Deductive Database		✓	Logical Relations	Datalog Query	✓
SEEdit	✓				✓	Higher Level Language, SPDL		✓	Grouping Permissions– Access Log – User Decision		
PVA	✓	✓	✓	✓	✓	Information Visualization Techniques	✓		Semantic Substrates – Adjacency Matrix	Graphical User Interface	
GPA	✓		✓		✓	Information Visualization Techniques			Semantic Substrates –Adjacency Matrix	Graphical User Interface	✓
Sepol2HRU	✓		✓			HRU Security Model			HRU Model Simulation		✓
SCIATool	✓		✓		✓	Colored Petri Nets, Information Flow, AC Space		✓	NA	Wizard– Style Query	✓
SEAL	✓		✓		✓	Information Flow			Syntactic Analysis		✓
EASEAndroid						Semi– Supervised Learning		✓	Parsing the Audit Log		✓
SELint						Information Flow			Syntactic Analysis		✓

modeling, expressing goals, enforcing goals of the model, and implementation. The language that encodes the security goals is based on information flow diagrams, and security goals are expressed using a language similar to regular expressions. Five different access control relations are defined to model SELinux configurations, which are based on key concepts of SELinux. The authorization relation uses access control relations to authorize class-permission pairs for a process against a resource. Finally, a model checker verifies the establishment of security goals of the policy.

XcelLog [Singh et al., 2007] combines policy rules and deductive spreadsheets (DSS) for taking advantage of deductive reasoning. The transformation of policy rules to deductive spreadsheets is a semi-automated process. Cells of the deductive spreadsheets are capable of containing a set of values or recursive formulas.

GOKYO [Jaeger et al., 2003a] tries to reveal various conflicts in the policy and find missing or incorrect constraints. GOKYO resolves these constraints, according to the concept of *Access control Space*s (AC spaces) [Jaeger et al., 2003a, Eaman et al., 2017], which can reduce the complexity of the policy. The process of resolving conflicts is based on removing the unknown subspace and performing a kind of balancing among different kinds of rules in the policy. The approach creates a near-minimal Trusted Computing Base (TCB) in the SELinux policy model and verifies whether the TCB is integrity-protected.

PAL [Archer et al., 2003] (Policy Analysis using Logic-Programming) is implemented using the XSB logic programming language. A XSB program translates a policy to a set of facts and builds queries that are answered from these facts. This technique is macro preserving, which means that the macros in the policy get expanded on demand. As stated in [Archer et al., 2003], PAL's use of macros that are not fully expanded is efficient and unique in contrast to other tools such as SLAT, APOL, and GOKYO.

SELAC [Zanin and Mancini, 2004] (SELinux Access Control) models each language construct in the security policy language as a mathematical set. A collection of sets is constructed incrementally from the specification. As stated in the paper, SELAC has removed the redundant space, unknown space, and general subspaces that are used in GOKYO.

SPTrack [Clemente et al., 2012] represents SELinux security policies as interaction graphs. The nodes in an interaction graph are security contexts made up of subjects or objects. The edges in this graph are possible interactions among nodes, all of which are included according to rules in the policy. The edges of the graph are colored based on the criticality levels of paths between nodes.

SEGrapher [Marouf and Shehab, 2011] begins its analysis with data visualization of the SELinux policy and then generates optimized graphs using the concept of clustering. Cluster-based graphs represent policy analysis results, which have been simplified by the use of clusters. To model a SELinux policy, the tool focuses on the access vector rules within it. These rules are represented as edges in a directed graph. The building block for clustering the nodes is a focus-graph, based on an object-type set. An object-type set is the set of all types that an object can access [Marouf and Shehab, 2011].

SEAnalyzer [Chen and Kao, 2006] utilizes Colored Petri Net (CPN) diagrams for representing SELinux security policies and security goals. A rather complex query language for expressing security goals has been developed for SEAnalyzer with a smaller character count in comparison to PAL and SLAT.

Lopol [Kissinger and Hale, 2006] takes advantage of deductive database analysis and Datalog queries. Lopol policy analysis includes analyzing a collection of logical relations and inference rules. Lopol is capable of rewriting the policy. Rewriting a policy is performed through goal-projection, which involves reverse compilation of the inference rules to the policy.

SEEdit [Nakamura et al., 2009] uses the concept of integrated permissions to reduce the number of configuration elements. Integrated permissions group related permissions into a single unit, which causes the removal of the macro entities from the policy. SEEdit creates security policies using a higher-level language called SPDL. The SPDL tool consists of two sections including an allow generator and a template generator. The former reads the access log to generate an SPDL based specification for permitting access. The latter uses the user's knowledge to generate an SPDL configuration to make a program that is problematic due to access control restrictions run correctly. Finally, an SPDL converter generates the policy file.

The PVA [Xu et al., 2013] and GPA [Xu et al., 2009] tools use a visualized-based framework for analyzing policies, expressing policy queries, and identifying policy violations of a SELinux policy. The concepts and proposed framework in GPA have been slightly enhanced in PVA. The framework begins by representing the policy layout using two visual mechanisms: Semantic Substrates and Adjacency Matrices. The framework provides a visual query formulation that helps system administrators specify precise queries on the policy. Subsequently, the framework generates a policy violation graph to represent the violations that are identified by the integrity model. The integrity model is based on Biba and the concepts of Trusted Computing Base (TCB) and Transaction Procedure in the Clark-Wilson security model. The framework introduces some approaches, such as filtering and ignoring, to modify the policy graph in order to remove any policy violations. GPA proposes identifying and protecting the TCB of a system using the Information Domain.

Sepol2HRU [Amthor et al., 2011] establishes an isomorphic mapping between a SELinux access control system and a HRU security model as defined in [Harrison et al., 1976]. Transforming the SELinux security policy to a HRU model allows the application of the analysis tools available for the HRU model to SELinux security policies. Transforming a policy to a HRU model is a three-step procedure. 1) The elements in a SELinux access control system, such as rules and types, are mapped to heterogeneous mathematic standard concepts like sets, matrices, and functions. 2) These elements are rewritten to a single composed matrix. 3) The authorization scheme is inferred. Sepol2HRU outputs the SELinux security policy as an HRU model description in a single file in a XML-based format.

SCIATool [Zhai et al., 2014] integrates three policy analysis methods including access control spaces, information flows, and colored Petri-nets. The architectural design of the

SCIAtool is based on the modularity principle. SCIATool's approach to integrity analysis is the use of a TCB which means that integrity analysis verifies that subjects inside the TCB are prohibited from reading incorrect information from non-trusted objects.

SEAL [Reshetova et al., 2015] is a tool for SEAndroid policy analysis. Finding problematic patterns in SEAndroid policies is the main purpose of the study in [Reshetova et al., 2015]. The identified patterns consist of overuse of default types, overuse of predefined domains, forgotten or seemingly useless rules, and potentially dangerous rules.

EASEAndroid [Wang et al., 2015] proposes a semi-supervised learning approach to refining SEAndroid security policies. SEAndroid security policies require continuous refinements due to continuous updates to Android and to emerging new attacks. A policy is refined based on analyzing the audit log and information in one access event. The tool parses information available on access events, which provides information for building access patterns. These access patterns act as a knowledge base for the learning process of the approach.

SELint [Reshetova et al., 2017] helps Original Equipment Manufacturers (OEMs) to produce better SEAndroid policies by optimizing the security policy. SELint has several plugins, including simple macro expansion, parameterized macro expansion, risky rules, unnecessary rules, and user *neverallow* rules. Plugins that operate on macros try to replace certain kinds of rules with macros. In contrast, other analysis tools seek to remove macros because the semantics of the m4-based language, i.e., macro language, is uncertain [Hurd et al., 2009].

2.7 Ease of Reasoning about SELinux Policies

A particular set of properties which may be used as a basis for formally comparing and contrasting access control policy languages include *Safety*, *Independent Composition*, *Monotonicity* and *Determinism* [Tschantz and Krishnamurthi, 2006]. An access control policy language that is safe, independently composable, monotonic, and deterministic is said to be most amenable to reasoning as compared to one that does not have any of these properties. In addition, these properties and others mentioned in [Sistany, 2016] can be used to classify different access control policy languages along the reasoning spectrum. Being able to reason about policies written in an access control policy language directly leads to another property that is desirable in a policy language. Such a policy language has the property that formal analysis and verification of specific policy statements can determine whether or not the policy meets the high-level goals of the system. In this section, we carry out this kind of formal analysis for SELinux.

Using the definition of an access control policy language as presented in [Tschantz and Krishnamurthi, 2006], the SELinux access control policy language can be considered as a tuple $L = (P, Q, G, N, \ll . \gg)$ where P is a set of SELinux policies, Q is a set of requests or queries, G is the granting decisions, and N is the non-granting decisions, with

the constraint $G \cap N = \emptyset$. Let D denote the set of decisions $G \cup N$. The last element of L, $\ll . \gg$, is a function taking a policy $p \in P$ to a relation between Q and D. Given a policy $p \in P$, a query $q \in Q$ is assigned a decision of $d \in D$, which is denoted as $q \ll p \gg d$. L also defines the partial order $\leq$ on decisions d, d' such that $d \leq d'$ if either $d, d' \in N$ or $d, d' \in G$ or $d \in N$ and $d' \in G$. Note that for SELinux, $D = \{Granted, Denied\}$, $G = \{Granted\}$ and $N = \{Denied\}$. Thus for SELinux, non-granting decisions are all the same, granting decisions are all the same, and all granting decisions are considered greater than all non-granting decisions. In TEpla, we add the *UnKnown* decision, which is for all the cases when neither granting nor denying decisions is applicable, a situation that results from authorizing accesses that fail to satisfy security goals (refer to the BNF grammar of TEpla in Fig. 3.1). In other words, as we will explain in Chapter 3, TEpla policies have two components including authorizing access rule component and constraint component, *UnKnown* decisions are the answer for queries that are authorized by the access rule component of policies, however, they fail to satisfy the constraint component of the policies. As a result, *UnKnown* decisions signify *conflicts* [Stepien and Felty, 2016] in policies.

Let DC, TC, CLS, and PRM be the set of all domains, types, object classes, and permissions, respectively, available in a system. Queries are of the form $(dc, tc, cls, prm, envirn)$ where $dc \in DC$ is the domain type of the subject, $tc \in TC$ is the type of the resource, $cls \in CLS$ is the class of the resource, $prm \in PRM$ is the permission or permissions, and *envirn* expresses some conditions, such as properties related to DAC or MLS mechanisms, that are not about the Type Enforcement mechanism of SELinux. Two queries $q = (dc, tc, cls, prm, envirn)$ and $q' = (dc, tc, cls, prm, envirn')$ have relation $q \sqsubseteq q'$ if they have equal *dc*, *tc*, *cls*, and *prm* components and we can deduce all the information in the $envirn'$ of the q' from the *envirn* of the q, $envirn \Longrightarrow envirn'$ (" $\Longrightarrow$ " denotes logical implication), in other words, if the first four components of the queries are equal and the *envirn* of the q is a subset of the $envirn'$ of q'. The authors in [Tschantz and Krishnamurthi, 2006] use this definition of $\sqsubseteq$ for comparing queries in the *FOL* policy language. In the rest of this section, we assess SELinux with regard to its Type Enforcement (TE) mechanism to determine if it satisfies the four properties mentioned earlier. The TE mechanism is based on TE rules available in SELinux policies. SELinux policies are organized into modules of policies and sub-policies, which is important for expressing and proving some of the properties, and which allows dynamic loading of policy modules as needed. Each policy module has its own set of rules.

An access control policy language is considered *Safe* if a request with less information will lead to a decision that is less than the decision reached for a request with more information, according to the defined partial order on decisions [Tschantz and Krishnamurthi, 2006]. For example, requests with incomplete information should only result in a grant of access if a request with more complete information results in a grant of access. Based on this definition, safety can be defined as the following formula (the notation "$\wedge$" denotes logical conjunction):

Definition 1 *Safety:*

$$\forall(p \in P), (q, q' \in Q), (d, d' \in D),$$
$$q \sqsubseteq q' \quad \wedge \quad q \ll p \gg d \quad \wedge \quad q' \ll p \gg d' \implies d \leq d'.$$

Theorem 1 *The SELinux access control policy language is not safe with respect to* $\sqsubseteq$.

Proof. Consider the policy module p_a below along with requests q_a and q_b:

```
p_a  :  role sCrole_r type sAtype_t
        allow sAtype_t mytype_t :  file read
q_a  =  (sAtype_t,mytype_t, file, read, {})
q_b  =  (sAtype_t,mytype_t, file, read, {role ∈ sDrole_r})
```

In the policy p_a, for example, we have a user with the role `sCrole_r`, and the type `sAtype_t` is an associated type to this role. For example, a web server with the type `sAtype_t` needs to access to read the type `mytype_t` that is applied to a data file. For request q_a, p_a produces *Granted*, while for q_b, it produces *Denied*. Note that $q_a \sqsubseteq q_b$, $q_a \ll p_a \gg$ *Granted*, $q_b \ll p_a \gg$ *Denied*, but *Granted* $\not\leq$ *Denied*, which contradicts safety.

In this property, it must be the case that if a query is granted, it continues to be granted when more information is added. Note that this example illustrates that this is not the case in SELinux. However, it is not likely that later queries will provide more information than needed to gain access. In Chapter 3, we improve the statement of this property based on what we find to be more important and applicable.

An access control policy language has the *Independent Composition* property if taking into account all policy modules and rendering a decision gives the same result as combining the decisions obtained from each primitive policy in isolation. As a result, the independent composition can be defined as the following formula, in which $\square$ is the decision composition operator for combining policy decisions and $\oplus$ is the composition operator defined in the language for combining policies. Some policy languages, such as FOL [Tschantz and Krishnamurthi, 2006], allow more than one interpretation of the operator that combines policies, thus preventing them from having the independent composition property.

Definition 2 *Independent Composition:*

$$\forall(p_1, \ldots, p_n \in P), (q \in Q), (d_1, \ldots d_n, d^* \in D),$$
$$q \ll p_1 \gg d_1 \quad \wedge \cdots \wedge \quad q \ll p_n \gg d_n \quad \wedge \quad q \ll \oplus(p_1, \ldots, p_n) \gg d^* \implies$$
$$\square(d_1, \ldots, d_n) = d^*.$$

Composing policies in SELinux simply means adding them together to form one big policy. A request is denied if any *one* of the individual policies produces `Denied`. Trivially, SELinux access control always reaches a single decision when combining all policy modules or decisions.

Theorem 2 *The SELinux access control policy language has the independent composition property.*

Proof Sketch. By definition, $\square(d_1, \ldots d_n)$ is *Denied* if any of $d_1, \ldots, d_n$ is *Denied*. In this case, the combined policy decision d^* will also be *Denied* by the definition of SELinux policy composition. Otherwise, $d_1, \ldots, d_n$ are all *Granted*, and in this case, both $\square(d_1, \ldots d_n)$ and d^* will be *Granted*, again by definition.

An access control policy language is *Monotonic* if adding another primitive policy does not change the combined decision from granting to non-granting.

Definition 3 *Monotonicity:*

$$\begin{array}{l} \forall(p_1, \ldots, p_n, p_{n+1} \in P), (q \in Q), (d_1, d_2 \in D), \\ q \ll \oplus(p_1, \ldots, p_n) \gg d_1 \quad \wedge \quad q \ll \oplus(p_1, \ldots, p_n, p_{n+1}) \gg d_2 \implies \\ \qquad d_1 \leq d_2. \end{array}$$

Theorem 3 *The SELinux access control policy language is not monotonic.*

Proof. Consider policy modules p_c and p_d below along with request q_c :

$$\begin{array}{lcl} p_c & : & \texttt{allow Dtype1\_t type2\_t : file open} \\ & & \texttt{allow sAtype\_t mytype\_t : file read} \\ p_d & : & \texttt{neverallow Dtype1\_t type2\_t : file open} \\ q_c & = & \texttt{(Dtype1\_t, type2\_t, file, open, \{\})} \end{array}$$

In the policy p_c, for example, we suppose that a mail server with the type `Dtype1_t` is allowed to open a user data file with the type `type2_t`, and a printer's process with the type `sAtype_t` is allowed to read a file with the type `mytype_t`. The policy p_c will result in *Granted* for the request q_c and adding policy module p_d will result in *Denied*, which changes the decision from *Granted* to *Denied*.

An access control policy language is *Deterministic* if the decisions for a specific query and policy are the same for all inquiries.

Definition 4 *Determinism:*

$$\begin{array}{l} \forall(p_1 \in P), (q \in Q), (d_1, d_2 \in D), \\ q \ll (p_1) \gg d_1 \quad \wedge \quad q \ll (p_1) \gg d_2 \implies d_1 = d_2. \end{array}$$

Theorem 4 *The SELinux access control policy language is deterministic.*

Proof Sketch. As SELinux always returns the same decision for a particular query and policy, the SELinux policy language is deterministic.

The verification of properties that we presented in this section are open to discussion because they are based on our understanding of the SELinux policy language. Moreover, they are not formally proved in Coq. We analyze them further, including some problems with them, when we formalize these properties for TEpla. Apart from the fact that policy analysis tools as well as policy developers can reap the benefits of policy languages with formal semantics and a verified set of properties, it is essential that the set of properties be related to the application domain of the policy language, focusing on the particular aspect of IT systems they cover. For instance, ACCPL (A Certified Core Policy Language) [Sistany, 2016, Sistany and Felty, 2017] was developed with the same goal in mind, to develop a language with formal semantics along with its formalization in Coq, but in the domain of web services and digital resources. Some initial properties were proved in Coq about ACCPL, but not yet the properties from [Tschantz and Krishnamurthi, 2006] that we have considered in this section. Although our approach to TEpla is the same as for ACCPL, it is in the domain of operating systems security, and we have focused on the properties in [Tschantz and Krishnamurthi, 2006].

2.8 Summary of Our Study

As a result of our study [Eaman et al., 2017], we can summarize the many gaps for using the SELinux policy language and developing verified security policies:

- For having a verified security policy, it is crucial to formally reason about the policy language in which the policy is written. We can conclude this as it is stated in [Jaeger et al., 2003a] that access control policies can be developed like programs. In [Pierce et al., 2008], it is stated that in order to verify programs, it is important to be able to reason about the programming language used to write the programs. This is one of the reasons that we argue that it is important to reason about policy languages.

- The SELinux policy language requires third-party analysis tools to help security administrators write policies and check various properties.

- The inherent complexity of the SELinux policy language as well as its lack of formal semantics have led to the development of many policy analysis tools to try to translate SELinux policies to another intermediate language constructs (see the translation arrow in Fig. 2.4). Therefore, this procedure entails a translation into another sophisticated language that requires equally complex semantics and syntax. Not only is it the case that these translations cannot be verified, they also map to another language, i.e., the intermediate language, which itself requires evaluation.

- Software developers continually add new rules to SELinux security policies, while fine-tuning the policy to handle access problems of newly installed applications, using the system audit log file. The practice of making every deny access found in the SELinux audit log into new rules in the policy is extremely error-prone and can lead to compromising the safety of the system. This is caused by the fact that root-cause analysis of the SELinux policy language is limited by the lack of a formal definition of its semantics.

- There is no proof for the correctness of policy analysis tools to make sure their results are reliable. There are informal justifications for results, but no formal justification of results. Moreover, the implementation of the SELinux semantics in one tool may deviate from that of another tool as there is no common formal understanding for the SELinux policy language.

- Overall, SELinux lacks clarity as an access control language. The clarity of an access control policy language can provide better decision making for incremental policy writing, ease of analysis, and ease of reasoning.

- In addition to studying the tools, we studied SELinux with respect to the properties in Section 2.7, which allowed a more mathematical analysis and showed that not all properties hold.

In the next section, we briefly introduce the Coq theorem prover since we use it in Chapter 4 to implement TEpla and provide machine-checked proofs for the language's behavior and properties.

2.9 The Coq Proof Assistant

Coq is an interactive theorem prover used to develop machine-checked proofs and often used to verify the correctness of programs. Such programs are called *certified* due to the fact that their conformance to their specification is formally verified. The Coq Proof Assistant is used in this work to verify theorems for the certification of properties of TEpla. These proofs are developed in a semi-interactive manner which depends on human guidance.

In Coq, properties and proofs are formalized in a specification language named *Gallina*, based on the *Curry-Howard isomorphism* [Bertot and Castéran, 2004]. This correspondence states that the relationship between types and programs is the same as the relationship between well-formed formulas and mathematical proofs. Therefore, type verification is the characteristic procedure to verify a proof. The *Calculus of Inductive Constructions (CIC)*, a λ-calculus with a rich type system, is the formal language of Coq [The Coq Development Team, 2019].

The Coq proof engine applies predefined *tactics* to manipulate local contexts and available goals to construct a proof. In other words, Coq's tactics are specific tools that break

down a goal into simple goals or sub-goals. It is possible to define new tactics through using a language called *Ltac* [Bertot and Castéran, 2004]. In the next parts of this section, we review some language constructs of Coq.

2.9.1 Basics

The standard library of Coq provides different data structures such as Booleans, lists, pairs, and hash tables. Considering that the set of these data structures is extremely small [Pierce et al., 2019], Coq provides the capability to define new datatypes through inductively defined types. Additionally, we can override provided data structures by building our own from scratch. We illustrate Coq's basic capabilities using examples from our application, which is presented in detail in Chapter 4. We, for example, define possible decisions in TEpla (see Chapter 4 and Section 3.2.1) using an inductive set as follows:

```
(* TEpla decisions *)
Inductive answer : Set :=
             | Permitted
             | NotPermitted
             | UnKnown.
```

Listing 2.5: The inductive datatype `answer`

`answer` is an inductively defined set with three constructors consisting of `Permitted`, `NotPermitted` and `UnKnown`. As we will see, constructors can take arguments; the constructors of `answer` take no arguments (see the following section for more details on inductive types). More details about the logical foundation and the basic language constructs of Coq are available in [Pierce et al., 2019, St-Marting, 2012].

2.9.2 Inductive Types

Defining new datatypes by providing different cases is possible using inductive types. An inductive type T has different cases (a vertical bar | is written before each case) in the form of $(CaseName : t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n \rightarrow T)$ in which *CaseName* is a *constructor* of the inductive type T, $(t_1, t_2, \ldots, t_n)$ are well-typed arguments, and type T is the inductive type that we are defining. Note that $n \geqslant 0$, which means that we can define cases that take no arguments. Provided that a case has no arguments, we can remove " $\rightarrow T$ " in defining the case, making its form as $(CaseName)$.

We defined three cases with no arguments in listing 2.5 to define the new type `answer`. That is, as mentioned earlier, `Permitted`, `NotPermitted` and `UnKnown` are constructors of inductive type `answer`, which have no arguments. As an example for constructors that take arguments, in listing 4.7, we define the inductive type `TErule` with two constructors, i.e., `Allow` and `Type_Transition`, which take 5 and 3 arguments, respectively (refer to Chapters 3 and 4).

2.9.3 Pattern Matching

Inductive types are used in Chapter 4 datatypes to encode TEpla in Coq. In defining many functions, we use the *pattern matching* technique to recognize how an inductive value is formed from the constructors of its datatype. For example, in listing 2.6, we perform pattern matching on input parameters c, d of type `answer`, which enables us to produce a Boolean value according to the formation of the values of these input parameters. Note that this pattern matching is exhaustive and sequential, that is, the pattern matching begins from the first case (i.e., `NotPermitted,_`) and continues to the last case (i.e., `_,_`) which is for all other cases that are not covered by the first three patterns and also it covers all the possible values of input parameters c, d. In listing 2.6, the underscore character "_" is Coq's syntax for considering all possible values of a particular variable.

2.9.4 Functions

After implementing datatypes needed for our application, we can define functions that operate on the datatypes and return values of some type. For instance, listing 2.6 is the definition of the function `compare_decisions` that takes two arguments of type `answer` and returns a value of type `bool`. This function implements the comparison operator for decisions in TEpla, as described in Section 3.4.1.

```
Definition compare_decisions (c d : answer) : bool :=
match c,d with
   | NotPermitted, _ ⇒ true
   | Permitted, Permitted ⇒ true
   | _, UnKnown ⇒ true
   | _, _ ⇒ false
 end.
```

Listing 2.6: The function `compare_decisions` to compare decisions

To manipulate data structure, we use the *pattern matching* feature of Coq, which is based on the inductively defined structure of datatypes. Function `compare_decisions`

manipulates the input arguments by performing pattern matching on the structure of the `answer` datatype by using keywords `match...with...end`.

2.9.5 Tactics

In Coq, theorems and lemmas are generally written as follows:

$$Hyp\ (H_1, H_2, \ldots, H_n) \vdash\ Conclusion\ (G).$$

We have hypotheses $H_1, H_2, \ldots, H_n$ as *the context* of the proof and at any point during the proof, we are proving one of the possible sub-goals (G). The following listing demonstrates one of the lemmas that we proved for TEpla in Coq (see Chapter 4 for the Coq encoding of TEpla). Coq uses "<>" as the operator for inequality, and it is defined in terms of Coq's built-in equality, e.g., `l1<>nil` is an abbreviation for `l1=nil -> False`. The infix operator "++" is defined in Coq to concatenate two lists. The inductive datatype `type` of TEpla will be defined in Chapter 4. Empty lists in Coq are denoted `nil` or `[]`.

```
Lemma not_nilList_concate_prop:
      ∀ (l1 l2: list type),
              l1 <> [ ] → l2 ++l1 <> [ ].
```

Here we explain some common tactics that help us to make progress in proofs and write proofs by decomposing a goal into sub-goals or transforming a (sub)-goal.

The `intro` tactic adds the current variable or premise of the goal as a new variable to the context. We can provide a new name for this variable by writing `intro new_name`. Moreover, we can use the tactic `intros` to introduce all variables or propositions on the left side of an implication as assumptions. Similar to the `intro` tactic, we can assign names to the assumptions which will be introduced after applying the `intros` tactic, through providing names as arguments to this tactic such as `intros new_name1 new_name2 ....` Starting the proof of lemma `not_nilList_concate_prop`, using the tactic `intros l1 l2 H1` introduces the assumptions `l1`, `l2`, and `H1: l1 <> [ ]` into the context, making the new goal the proposition `l2 ++ l1 <> [ ]`. Therefore, the following proof state demonstrates the context, which includes the hypotheses above the dashed line, and the new goal, which is the goal below the dashed line, after applying the tactic `intros l1 l2 H1`.

```
l1 :  list type
l2 :  list type
H1 :  l1 <> [ ]
------------------------------------------------
l2 ++ l1 <> [ ]
```

The destruct tactic helps us to perform case analysis on inductively defined datatypes. Using this tactic on a hypothesis containing a term belonging to such a datatype, it will generate sub-goals for each possible constructor of this inductively defined type. Since we defined answer with three constructors, if the term anwr is of type answer, applying the tactic destruct anwr to the current (sub)-goal will generate three sub-goals replacing the term anwr with its possible values (i.e., Permitted in one subgoal and NotPermitted, UnKnown in the others). As an example for using destruct on lists, we can continue to prove the lemma not_nilList_concate_prop by applying tactic destruct l2. (after applying the tactic intros l1 l2 H1). In this case, we will have two sub-goals as [] ++ l1 <> [] in which l2 is replaced by an empty list, and (t::l2) ++ l1 <> [] in which l2 is replaced by t::l2. The notation "::" is used for the constructor cons of the list datatype, from the Coq library Coq.Lists.List, which appends one element to the head of a list.

The apply tactic allows us to manipulate the current goal by one of the possible logical implications that are currently in the context. If we have the hypothesis H1: A -> B in the context and the current goal is B, we can change the current goal to A by writing the tactic apply H1.

The rewrite tactic transforms one term of the current goal into the equivalent term in the context. Suppose we want, for example, to prove that C= B + B and the hypothesis H1: A = B is in the context of our proof and the current goal is C= A + B, we can transform the current goal to C= B + B by using the tactic rewrite H1.

The cut tactic enables us to add a new hypothesis to the context. In this case, first, we prove the goal using the new hypothesis and then we have to prove that the introduced hypothesis is true as well. For example, the tactic cut (A = B) allows us to add the proposition (A = B) to the context, as long as (A = B) is provable from the current context.

The reflexivity tactic proves a goal if it is in the form of an equality A = B, where A and B are exactly the same term, possibly after some simplification.

The induction tactic generates subgoals for every constructor of an inductively defined type. In addition, it also generates an induction hypothesis to use for proving our goal. Therefore, the tactic induction is similar to the tactic destruct except for providing an induction hypothesis for recursively defined constructors.

2.9.6 Proof by Reflection

Proof by reflection is considered as one of the main features of Coq [The Coq Development Team, 2019, Grégoire and Tassi, 2016]. Proof by reflection enables us to write proofs that exploit both the boolean interpretations and the propositional representations of facts. That is, proof by reflection takes advantage of combining computations with logical facts in proving proofs. In particular, boolean interpretations provide computations and propositional representations provide logical facts for proofs. Accordingly, by proving that a

proposition holds when a boolean expression is true, we can use a combination of these interpretations in proofs because they form *rewritable equations* [Gonthier and Mahboubi, 2010] that can be used in a case analysis of inductive types.

Here, for example, we prove by reflection that the logical disjunction is decidable. The definition `orLogicProp` is the propositional interpretation of logical disjunction. The function `orLogicProp` returns a proposition to indicate whether or not disjunction between input argument of type `bool` holds.

```
Definition orLogicProp (A B:bool) : Prop :=
 match A with
    | true ⇒ True
    | false ⇒ (B=true)
 end.
```

On the other hand, the definition `orLogicBool` is the boolean expression of logical disjunction. The function `orLogicBool` has the same logical content as `orLogicProp` because both are related to logical disjunction. The difference is that `orLogicBool` returns a `bool`.

```
Definition orLogicBool (b1 b2:bool) : bool :=
     match b1 with
    |true ⇒ true
    |false ⇒ b2
 end.
```

We use the *reflect* predicate [Gonthier and Mahboubi, 2010] (depicted in listing 2.7) to express that `orLogicProp` holds when `orLogicBool` returns true, defined in Theorem `OR_PropBoolreflection`.

```
Inductive reflect (P: Prop): bool → Type :=
  | Reflect_true: P ⇒ reflect P true
  | Reflect_false: P ⇒ reflect P false.
```

Listing 2.7: The reflect predicate

As a result of Theorem `OR_PropBoolreflection`, there is a reflection relation between these two characterizations, i.e., the boolean disjunction (`orLogicProp d1 d2`) and the logical disjuction (`orLogicBool`). The reflection predicate in the Theorem `OR_PropBoolreflection` asserts that `orLogicProp d1 d2` is logically equivalent to the proposition (`orLogicBool d1 d2` = true).

```
Theorem OR_PropBoolreflection: ∀ d1 d2,
           reflect (orLogicProp d1 d2) (orLogicBool d1 d2).
```

After proving the reflection relation between `orLogicProp` and `orLogicBool`, we can use this relation in proving the decidability of the relation `orLogicProp`, for example. Theorem `OR_Decidable` expresses the decidability of the relation `orLogicProp`.

```
Theorem OR_Decidable d1 d2 :
   { orLogicProp d1 d2 } + { orLogicProp d1 d2 }.
```

Chapter 3

Infrastructure of TEpla

This chapter outlines the key language concepts of TEpla. We discuss the TEpla language structures and the meaning of these structures, that is, the syntax and semantics of TEpla. In addition, this chapter sets out to present language-level features of TEpla through a set of formal properties. These properties, which depend on the formal semantics of TEpla, positively impact policy development and analysis, as will be discussed later. In the next chapter, we go a bit deeper and go through the mechanized formalization of infrastructure and proofs of properties of TEpla using the Coq proof assistant, and thereby provide more details for how we implement TEpla (see Chapter 4).

3.1 Overview

The main building block of TEpla is *type* which is the core language concept in TEpla's syntax and semantics. *Type*s act as labels for elements of computer systems, which enables policy administrators to regulate access between entities. Virtually all language constructs and semantics of TEpla revolve around the concept of *type*. One of the main design decisions in developing TEpla is to develop a policy language that is easy to comprehend, which is attained by keeping the core of language simple and basing it on the concept of *type*. This feature and the provided insights into TEpla policy behaviors, presented by the formal properties of TEpla, contribute to having a correct common understanding of the language for developing and analyzing security policies. It also provides certain guarantees. TEpla thus helps administrators to better understand upfront how policies and access requests will act when they are deployed.

The semantics of TEpla is expressed as denotational mappings through translation functions from access requests and policy specifications to decisions. All the translation functions together act as a decision-making chain, which are dependent on each other. The final decision for an access request thus is a combination of various parts of the semantics of TEpla.

In Section 3.2, we present the syntax of TEpla and in Section 3.3, we present its semantics. Section 3.4 presents an overview of a set of main properties of TEpla. These properties are based on the idea that we have a *set* interpretation over policies, access requests, and decisions of TEpla. Owing to this *set* interpretation, we define *ordering relations* [Walick, 2016] on these sets and investigate how TEpla's semantics act on them.

3.2 Syntax

The basic data for making access control decisions in TEpla are primitive attributes, the core of which is *type*. In TEpla, the primitive attributes include *type*, *Object Class*, *Permitted Action*. All of these entities are used likewise in the SELinux decision-making process. *Object class*es specify possible instances of all resources of a certain category, such as files, sockets, and directories. In particular, primarily, this grouping is used to define a list of *permitted action*s for each group (i.e., object class). *Permitted action*s specify the actions that subjects are authorized to perform on the objects. Note that the concept of `types` is different from *Object Class*es because the former act as labels for elements of computer systems (single element or aggregate group of elements for identifying them), which are for identifying elements in a system, however, the latter determines the category of objects that each element belongs. Accordingly, SELinux defines a set of particular *permitted action*s for each of *object class*es. TEpla uses *permitted action*s defined in access rules to authorize queries, and we assume that the security framework of the system controls, instead of the policy language, whether or not performing an action on an *object class* is allowed, however, we leave this feature (i.e., defining a valid list of *permitted action*s for each of *object class*es) as a future work. These data types provide a straightforward syntax for policy writers. Other language constructs of TEpla exploit primitive attributes to express how they render decisions for access requests. Moreover, primitive attributes can be used to define additional restrictions on policies (see Section 3.2.6). In this thesis, we assume all the resources in the system have a *type* attribute assigned to them, and thereby, TEpla policies can enforce access decisions on system resources.

The BNF grammar of TEpla is defined in Fig. 3.1. We use { } notation to represent the Kleene operator meaning 0 or more occurrences. In this grammar, the primitive attributes *type*, *Object Class*, *Permitted Action* are denoted by *type*, *cls*, *prm* respectively. *Object classes* specify the classes in which objects are included (i.e., the possible instances of classes) and *Permitted Actions* define the authorized actions to be performed on the objects (see Section 2.5 for more details).

In the BNF grammar of TEpla, *TEpredicate_ID* represents a set of functions having eight arguments and returning a boolean. This set of functions will be explained in Section 3.2.7.

3.2.1 TEpla Decisions

TEpla has a three-valued decision set for access requests including *NotPermitted*, *Permitted* and *UnKnown*. The *NotPermitted* decision is for denying an access request, and the *Permitted* decision for granting an access request. The *UnKnown* decision arises from conflicts in TEpla policies. Conflicts are caused by rendering a decision for access requests in a part of security policies that is different from an already taken decision according to other policy statements. More specifically, when the result is *UnKnown*, it is the job of the administrators to fix it, using their discretion. Refining TEpla's policies or revisiting the security framework of the system are two possible options for administrators to address this issue. In TEpla, policies have two parts: rules (*TErules*) and constraints (*TEconstraints*). Constraints add conditions on the rules that may replace a decision obtained from considering only the rules alone. In TEpla's semantics (see Section 3.3), conflicts are the result of such a replacement (see subsections 3.2.6 and 3.3.2). TEpla's decisions are based on a *Closed World Assumption* [Bishop, 2002] that denies permissions of access queries by default. In view of the *Closed World Assumption*, every grant access request has to be explicitly authorized through security policies.

We use *dcs* to represent decisions in the TEpla BNF grammar in Fig. 3.1. The semantics of TEpla (see Section 3.3) ultimately will return one of the possible decisions in TEpla (i.e., *NotPermitted*, *Permitted* and *UnKnown*). In our definitions as well as in the text, the word *answer* is used interchangeably with *decision*.

cls ::= ; object class
    net_socket | $filesystem$ | ; classes
    tcp_socket | ...

prm ::= ; permitted action
    $read$ | $signal$ | $setattr$ | ; actions
    $entrypoint$ | ...

$basictype$ ::= ; basic type
    $print\_exec\_t$ | $http\_t$ | ; basic types
    $mail\_t$ | ...

$attribute$ ::= ; attribute
    $basictype$ {$basictype$}

$type$ ::= ; TEpla type
    $basictype$ | $attribute$; types

$subject$ ::= $type$; subject type

$object$ ::= $type$; object type

$target$ ::= $type^*$; target type (type* is a basictye or an attribute with one basictype)

$cond\_bool$ ::= ; bool
    $true$ | $false$ | $\cdots$; bool value or expression

$Allow$ ::= ; allow rule
    ($subject, object, cls, prm, cond\_bool$) ; allow rule

$Type\_Transition$ ::= ; type transition
    ($subject$, $target$, cls) ; type transition rule

$TErule$ ::= ; policy rule
    $Allow$ | $Type\_Transition$; rules

$TEpredicate\_ID$ ::= ; identifiers for predicates in TEpla (*TEpredicates*)
    $identifier$; a *TEpredicate* function name

$TEconstraint$ ::= ; policy constraint
    (cls, prm, $type$, $type$, {$type$}, $TEpredicate\_ID$) ; constraint

$TEpolicy$ ::= ; TEpla policy
    ({$TErule$}, {$TEconstraint$}) ; policy

qr ::= ; query
    ($subject$, ($object$ | $target$), cls, prm) ; query specifics

dcs ::= ; decisions of TEpla
    $UnKnown$ | $Permitted$ | $NotPermitted$;decisions

Figure 3.1: The BNF grammar of TEpla

3.2.2 Type Enforcement in TEpla

Type Enforcement exploits the security context of resources to regulate accesses. Recall that the security context is a set of values of particular attributes assigned to resources [Eaman et al., 2017], as also defined in Section 2.4. Type Enforcement uses the security context to identify different entities of a system. Access control in Type Enforcement mechanism thus is based on security contexts associated with resources.

We use the *basictype* syntax class in TEpla as the security context of resources. That is, every system resource is labeled with a value from the *basictype* syntax class. This makes TEpla a fine-grained policy language as administrators can apply different access regulations on different entities of a system. Note that we have assumed that all resources of the system are assigned a particular security context, and thus the security context for resources is always defined. For example, for two resources of a system, *file_web* and *port_protocol*, we can assign the security contexts `mail_t` and `http_t` respectively.

As mentioned in Chapter 2, in computer systems, resources can be grouped into *subjects*, which attempt to perform an action on other resources, and *objects*, which are accessed by *subjects*, and thus *subjects* act upon *objects*.

In TEpla policy statements, source and destination entities of accesses, which are *subjects* and *objects* respectively, are specified by *type*s. In contrast to *domains* and *type*s in SELinux, we simply use *type* in defining both *subjects* and *objects*. The security contexts `http_t` or `mail_t`, mentioned above are *subjects*, which try to access objects in the system. Similarly, as another example, `print_exec_t` can specify an *object*, which is being accessed by *subjects* such as `http_t` or `mail_t`. As these examples are values of the *basictype* syntax class which is one of the elements of the *type* syntax class, they can determine source or destination entities in policy statements, i.e., subjects or objects. Note that in policies a resource can be viewed as a subject in one rule and an object in another.

Taking into account the fact that computer systems include numerous elements that require many rules to cover them in policies, we need to logically relate elements (i.e., computer resources) to each other, enabling the policy developers to address a group of resources under a single identifier. Similar to *attributes* in SELinux, we add this feature to *type*s in TEpla. *Type*s in TEpla thus consists of two kinds: *basictype*s and *attribute*s (see the BNF grammar of TEpla). *Basictype*s are a sort of *type*s which are assigned to every basic element of resources. By associating a *basictype* to all resources, we can logically group these *basictype*s through assigning shared identifiers to the group of *basictype*s. *Attribute* thereby implies there exists a conceptual relationship among a set of *basictype*s. The attribute `program_type`, for example, can include two *basictype*s such as {`mail_t, http_t`}, which can be used to specify the source or destination of accesses in policy statements.

Therefore, in TEpla, access rules can use two kind of *type*s to specify source and destination entities: *basictype*, which is the single label for a particular element and *attribute*,

which aggregates a group of *basictype*s in a single identifier. We use *Source Type* and *Destination Type* to represent source and destination entities in policy statements.

In TEpla policy specifications, by convention, we use the suffix "`_t`" to denote *basictype*s and "`_type`" to denote *attribute*s. This is not enforced by the grammar. The access rules of TEpla (denoted by *TErule*) consist of *Allow* and *Type_Transition* rules. These rules grant access requests according to the security context of elements in TEpla. In the following subsections, we introduce other features of the TEpla policy language and provide details of the mentioned access rules.

3.2.3 Allow Rules

Allow rules enable policy writers to express eligible access from a source type to a destination type. The components of *Allow* rules include *Source Type* (*subject*), *Destination Type* (*object*), *Object Class* (*cls*), *Permitted Actions* (*prm*), and *Conditional Boolean* (*cond_bool*), respectively as shown in Fig. 3.1. *Source Type* and *Destination Type*, as their names indicate, identify the subject and object type of the rules. *Object Class* specifies the object class of the *Destination Type* and *Permitted Actions* determines possible actions which *Source Type* can perform on the *Destination Type*. The last component of *Allow* rules is a Boolean condition. Granting the specified access depends on the value of the condition expressed by *Conditional Boolean* as it is only granted if the value is true. This feature enables policy writers to specify conditions to control granting accesses by *Allow* rules.

By *Allow* rules, for example, we can express that the *basictype* `http_t` is authorized to access the *basictype* `print_exec_t` of *object class* `tcp_socket` to perform a `read` operation, resulting in a *Allow* rule, named *TEr1* for example, of the form *Allow* (`http_t,print_exec_t,tcp_socket,read,true`). In the same way, another *Allow* rule can express the same access specification for the *basictype* `mail_t` (i.e., authorizing the *basictype* `mail_t` to access the *basictype* `print_exec_t` of *object class* `tcp_socket` to perform a `read` operation), however, we can use the same *Allow* rule (i.e., *TEr1*) but replacing the *basictype* `http_t` with the *attribute* `program_type` which we defined earlier as an *attribute* that includes the two *basictype*s `http_t` and `mail_t`, resulting in a *Allow* rule named *TEr2* of the form *Allow* (`program_type,print_exec_t,tcp_socket,read,true`). In this rule, it is assumed that the `print_exec_t` object is in the `tcp_socket` *object class* and that `read` is a `permitted action` that is allowed on this *object class*. Note that SELinux enforces these constraints; enforcing them in TEpla is left for future work.

3.2.4 Type Transition Rules

TEpla supports transition of *type*s in security contexts. *Type_Transition* rules are policy statements which determine *type*s that can switch to other *type*s. *Type_Transition* rules

include three components: *Source Type* (*subject*), *Target Type* (*type*), and *Object Class* (*cls*). The *Source Type*, *Target Type* state the initial *type* of the context and the new value for it, respectively. Similar to source *type* of *Allow* rules, we can use *attributes* in *Source Type* of *Type_Transition* rules. However, *Target Type* of *Type_Transition* rules has to precisely specify the intended *type* (i.e., a *basictype* or an *attribute* with one *basictype*) because using multiple *basictype*s in *attribute*s for the *Target Type*, the transition rule would be ambiguous. This is not enforced by the BNF grammar of TEpla and adding a constraint that an attribute must be a list of at least length two, with different *basictype*s, is left for future work.

By *Type_Transition* rules, for example, we can express that the *basictype* `mail_t` is authorized to transition to the *basictype* of the *object class* `process`, such as *basictype* `print_t`, which can be expressed by the rule *Type_Transition* (`mail_t,print_t,process`). Note that the concept of *entrypoint* [Mayer et al., 2006] in SELinux is left for future work in TEpla. However, TEpla constraints can add extra regulations on type transitions.

3.2.5 Access Requests

In TEpla, *access requests* or *queries*, represented as *qr* in the TEpla BNF grammar, consist of four components *Source Type*, *Destination Type*, *Object Class* and *Permitted Action*, respectively as shown in Fig. 3.1. Access requests are inquiries into the policy to check the possibility that *Source Type* is allowed to perform the action *Permitted Action* on the object *Destination Type* of the object class *Object Class* (see the BNF grammar of TEpla in Fig. 3.1). Processing of a query with respect to a policy involves an attempt to check the authorization of a subject element to carry out a specific access on an object element of a particular class; both the subject and object belong to the *type* syntax class. Translation functions for semantics of TEpla (see Section 3.3) receive access requests as one of their inputs.

In addition to the above description for queries, access requests can ask for authorization of transition of *type*s to another specific *type*s (refer to subsection 3.2.4). In this view, queries (*qr*) ask for authorization of a *type* transition specified by *Source Type* and *Target Type* with a specific *Object Class* (*cls*) for the *Target Type*. As *Type_Transition* rules do not authorize access and instead they permit transition of *type*s, the *Permitted Actions* (*prm*) part of queries that ask about transition of *type*s is unchangeable in all the queries of this kind. Thereby, a unique *Permitted Action*, such as *transition* will be appropriate for the queries intended to authorize the transition of *type*s.

3.2.6 Constraints as Higher-Order Functions

Although *TErule*s can express the regulations for authorizing access requests, they cannot accommodate the security requirements of systems precisely enough. TEpla's constraints

(*TEconstraint*s) can complement *TErule*s with the goal of providing administrators a feature to precisely express detailed aspects of safe systems. *TEconstraint*s represent one of the powerful features of TEpla, which can be tailored to different security requirements. In comparison to other languages which lack this feature, *TEconstraint*s allow policy writers not only to rely on conditions or constraints defined in the language but also to define their complementary security logic.

*TEconstraint*s in TEpla, (see the BNF grammar of TEpla), have six arguments. The first two arguments are *Object Class* (*cls*) and *Permitted Action* (*prm*). These two arguments are compared to the *Object Class* (*cls*) and *Permitted Action* (*prm*) of a query to check if the *TEconstraint* is applicable to the query. The next three arguments are *type*, *type*, and {*type*} (i.e., a list of *type*s) whose values are provided by the policy writers, explained later when presenting semantics.

*TEconstraint*s in TEpla include a function as their last argument. As mentioned, these functions return a Boolean. Therefore, a *TEconstraint* in TEpla is a *Higher-Order Function* (HOF) [Abelson and Sussman, 1996, Chlipala, 2019]. This is because a *TEconstraint* has one function argument that returns a Boolean. To express specific security goals, administrators can define the body of this function by using various arguments provided for the function (see the next subsection).

Using *TEconstraint*s, for example, we can express that the *basictype* `http_t` can access the *basictype* `print_exec_t` of *object class* `file` to perform a `read` action provided that the set of *type*s that access both `http_t` and `print_exec_t` are distinct and there are no common *type*s in these sets, i.e., their intersection is empty. As a result, we need to encode this check in a function that returns *true* or *false* if the intersection is empty or not. The function computes the two sets that access `http_t` and `print_exec_t` and evaluates their intersection. This function is a *predicate* that we describe in the next subsection. Note that in order to compute these sets, the predicate (function) must take as an argument the list of *Allow* rules of the policy, because only by searching this list can the function compute the set of *type*s that access `http_t` and `print_exec_t`. In this example, the constraint provides the arguments `print_exec_t`, `http_t`, `read`, and `file` to the predicate. Note that other constraints can use the same predicate to apply the same security goal but with a different arguments which come from the new constraint.

Consequently, we define the constraint *TEcstr1* as `TEconstraint (file,read,-http_t,print_exec_t,[],predicate_checkSoD)` in which `predicate_check-Sod` is the name of the predicate described above. In particular, the predicate `predicate_checkSoD` has to receive as arguments the list of *Allow* rules of the policy (*TEr1* and *TEr2* defined in section 3.2.8, for example). In the following subsection, we introduce TEpla *predicate*s, which are used to encode different security goals for constraints.

The security goal that we just described is called *Separation of Duty*, and is described and encoded more fully in the next chapter. Note that the fifth component of the constraint *TEcstr1* is empty, which is represented by an empty list in the form of [ ], since the `predicate_checkSod` does not use this argument. In this example, the predicate must

return *true* because the set of *type*s which access both `http_t` and `print_exec_t` is empty, which is computed from the access rules *TEr1* and *TEr2*. In the access rules *TEr1* and *TEr2*, the set of *type*s that access `http_t` is empty, and the set of *type*s that access `print_exec_t` is {`mail_t`,`http_t`}. This *true* result of the predicate leads to the *Permitted* decision for the query, named *Qr1*, (`http_t`,`print_exec_t`,`file`,`read`). If the intersection was not empty, the predicate would return *false*, which would lead to the *UnKnown* decision for the same query.

3.2.7 Predicates

As just mentioned above, the last input argument of a *TEconstraint* is a function that returns a Boolean, which makes these functions act as *predicates* [Harzheim, 2005]. Recall that TEpla predicates are denoted by *TEpredicates_ID*. When a query is evaluated against a policy (made up of *TErules* and *TEconstraints*) the *TEpredicate* is evaluated, provided that the *TEconstraint* is applicable to the query, with specific arguments provided by the *TEconstraint* and the query.

As mentioned earlier in Section 3.2, *TEpredicates_ID* has eight arguments. The first argument is a list of *TErules*. The next argument of *TEpredicate* is a list of *type*s. The next four arguments of *TEpredicate*s are *Object Class* (*cls*), *Permitted Actions* (*prm*), *type*, and *type* respectively. The remaining two arguments are *type*. All of these input arguments supply a comprehensive set of values by which policy developers can define the required security criteria.

For encoding the example security goal that we expressed in Section 3.2.6, we need a predicate to check the condition that there is no common *type* that accesses both `http_t` and `print_exec_t`. A predicate can express this condition by comparing the set of source *type*s (i.e., subjects) that access `http_t` with the set of source *type*s that access `print_exec_t`. Here we express the pseudo-code of such a predicate for the example security goal that we expressed in Section 3.2.6.

procedure PREDICATE *predicate_checkSoD*(arg1: list *TErule*, arg2:list *type*, arg3:*cls*, arg4:*prm*, (arg5 arg6 arg7 arg8: *type*))
  if (*arg5* $\subset$ *arg7*) $\wedge$ (*arg6* $\subset$ *arg8*) **then**
    sourceTypefor_arg7 $\leftarrow$ search *arg1* for finding subject *types* that access *arg7*
    sourceTypefor_arg8 $\leftarrow$ search *arg1* for finding subject *types* that access *arg8*
      return ((*sourceTypefor_arg7* $\cap$ *sourceTypefor_arg8*) $= \varnothing$)
  else return true

Continuing our example with *TEcstr1* and *Qr1*, this predicate is applied to arguments as follows: `predicate_checkSoD` ([*TEr1*,*TEr2*], [ ],`file`,`read`,`http_t`,`print_exec_t`,`http_t`,`print_exec_t`). In the above procedure, *arg1* contains all the *Allow* rules of the policy. Note that this predicate does not use the second input argument because it is not required in computing the intersection of the two lists. The fifth argument

of constraints provides the second input arguments of predicates. In our example, it is [ ] which is the fifth argument of the constraint *TEcstr1*. The third and fourth arguments of the predicate (i.e., *arg3*, *arg4*) are not used in this example. The first and second arguments of constraints provide these input arguments of predicates, respectively. The fifth and sixth arguments (i.e., *arg5*, *arg6*) are the subject and object *type*s of queries. Finally, the seventh and eighth arguments are the subject and object *type*s whose values are specified by the third and fourth arguments of constraints. The body of this predicate function searches the first input argument, i.e., *arg1*, to find all the subject *type*s that access the seventh and eighth input arguments. If these two sets share common *type*s, the predicate returns `false`, otherwise, `true`. To check if the predicate is applicable to a query, we compare the fifth input argument (i.e., the subject *type* of the input query) with the seventh input argument (i.e., the subject *type* received from the constraint). Similarly, we check the same relation between the sixth and eighth input arguments (i.e., the subject *type* of input queries and the subject *type* received from the constraint, respectively). See Section 3.3 for more details about the semantics of TEpla.

Administrators using this feature (i.e., defining *TEpredicates*) must make sure their definitions are compatible with TEpla's properties. We provide the details of the requirement in the next chapter. This feature of TEpla for defining different predicates increases the expressive power of the language; however, checking that predicates satisfy particular conditions can lead to the impression that the language is complex. Despite the fact that defining predicates requires checking for compatibility with some conditions (see Section 4.3), policy developers perform this checking statically, and there is no other overhead or slow down in the execution process in exploiting predicates in policies. A detailed description of the expressive power of predicates is available in Section 4.3.1.

3.2.8 TEpla Security Policies

Security policies in TEpla allow administrators to express access permissions as well as security conditions which specify additional restrictions based on security requirements of systems. TEpla policies (*TEpolicy* in the BNF grammar) consist of a pair of a *TErules* sequence and a *TEcontraints* sequence (see Sections 3.2.2 and 3.2.6). Policies thus can be formed by any sequences of *TErules* and *TEconstraints*.

3.3 Semantics

We define the semantics of TEpla as a mapping from policies and access requests to decisions. The output of this mapping is specified by translation functions. These translation functions involve all possible parts of TEpla policies. As TEpla policies are made up of sequences of *TErules* and *TEconstraints*, we need to provide translation functions for a single *TErule* and *TEconstraint* and then show how to compose them. There are six translation

functions which cover all aspects. In the following subsections, we present pseudo-code for these translation functions as Algorithms (1 – 6).

3.3.1 Semantics for *TErule*

The first translation function defining the semantics of TEpla, shown in Algorithm (1), determines how an *Allow* rule leads to a decision for an access query. The algorithm uses the last component of the input elements of *Allow* rules to check if the conditional element of the *Allow* rule is *true* or *false*. This condition is evaluated according to the current state of the system. Following this evaluation, the algorithm checks the requirement that the source and destination *type* of queries are a subset of the source and destination *type* of the *Allow* rules, respectively.

We define the following subset relation, called *isSubset*, between TEpla types (*type* in the grammar). Here lists are used to represent sets. This is achieved by ignoring duplication and ordering of elements in the definition.

- A *basictype* is a subset of another *basictype* if they are the same.
- A *basictype* is a subset of an *attribute* if the *basictype* is one of the elements of the *attribute* list.
- An *attribute* is a subset of a *basictype* if every member of the *attribute* list is the same as *basictype*.
- An *attribute* is a subset of another *attribute* if every member in the first *attribute* list also appears in the second.

For making the final decision for an *Allow* rule, the *Object Class* (*cls*) and *Permitted Action* (*prm*) components of the input query and the *Allow* rule are matched to check whether they are equal or not. As Algorithm (1) shows, failing to satisfy any conditions of the algorithm results in *NotPermitted* since the *Allow* rule is not applicable to the query, and thus cannot authorize it.

Algorithm 1 Algorithm for evaluating a TEpla Allow rule (*Allow*) and a Query

```
1: procedure TRANSLATION FUNCTION 1 (TRFU1) OF TEPLA ALLOW RULE (allowRule, query)
2:     sourceTypeRule ← first component of allowRule
3:     destinationTypeRule ← second component of allowRule
4:     sourceTypeQuery ← first component of query
5:     destinationTypeQuery ← second component of query
6:     condition_bool ← last component of allowRule
7:     cls_AllowRule ← third component of allowRule
8:     perm_AllowRule ← fourth component of allowRule
9:     cls_Query ← third component of query
10:    perm_Query ← fourth component of query
11:    if condition_bool isEqual true then
12:        if sourceTypeQuery isSubset sourceTypeRule then
13:            if destinationTypeQuery isSubset destinationTypeRule then
14:                if (cls_Query isEqual cls_AllowRule) then
15:                    if (perm_Query isEqual perm_AllowRule) then return Permitted
16:                    else return NotPermitted
17:                else return NotPermitted
18:            else return NotPermitted
19:        else return NotPermitted
20:    else return NotPermitted
```

The next translation function is for the second kind of TErules, i.e., *Type_Transition* rules, shown in Algorithm (2). It receives a *Type_Transition* rule and a query. The algorithm uses the source and destination *type* components of queries to check if they are subsets of the source and destination *type* of *Type_Transition* rules, respectively. In addition, the *Object Class* components of both access requests and *Type_Transition* rules have to be equal.

Algorithm 2 Algorithm for evaluating a TEpla Type_Transition rule and a Query

```
1: procedure Translation Function 2 (TrFu2) of TEpla Type_Transition
   Rule (typeTransitionRule, query)
2:     sourceTypeRule ← first component of typeTransitionRule
3:     targetTypeRule ← second component of typeTransitionRule
4:     cls_TransitionRule ← third component of typeTransitionRule
5:     sourceTypeQuery ← first component of query
6:     targetTypeQuery ← second component of query
7:     cls_Query ← third component of query
8:     if sourceTypeQuery isSubset sourceTypeRule then
9:         if targetTypeQuery isSubset targetTypeRule then
10:            if cls_TransitionRule isEqual cls_Query then    return Permitted
11:            else return NotPermitted
12:        else return NotPermitted
13:    else return NotPermitted
```

Algorithms (1) and (2) map a *TErule* to a decision. Taking into account that policies have a sequence of *TErules*, we need another translation function to translate a sequence of *TErules*. This third translation function, depicted in Algorithm (3), shows how this translation function produces a list of decisions as specified by the sequence of *TErule*. To denote a sequence of entities, when required in the algorithms of this section, we use $[n]$, which represents a list of size n. We use lists of size n to keep track of *TErules* and their related decisions. This list of decisions is used in another translation function to make a final decision for the query. Note that indices for lists go from 0 to n where n is the length of list.

Algorithm 3 Algorithm for evaluating a list of TErules

```
1: procedure Translation Function 3 (TrFu3) of TEpla list of
   TErules (list_TErule, query)
2:     n ← number of TErules in list_TErule
3:     TErule_sequence [n] ← list_TErule
4:     decisionList [n] ← output list of decisions of size n
5:     i ← n
6: while (i) > (0) do
7:     if TErule_sequence [i] is Allow rule then
8:         decisionList [i] ← TrFu1 on (TErule_sequence [i],query)
9:     else
10:        decisionList [i] ← TrFu2 on (TErule_sequence [i],query)
11:    i ← i − 1.
12:    end-do.
13: return decisionList
```

3.3.2 Semantics for *TEconstraint*

The fourth algorithm, shown in Algorithm (4), shows the decision-making steps for a *TEconstraint*, a query and a list of *TErule*s. The algorithm first checks if the *TEconstraint* is applicable to the query. Specifically, the algorithm inspects whether or not the *Object Class* and *Permitted Action* parts of the query and the *TEconstraint* are identical. After this evaluation, the predicate function *TEpredicate* (here *PredicateFunc_ConstraintRule*) within the *TEconstraint* (here *constraintRule*) is evaluated to determine whether or not the query satisfies the security requirements of the *TEpredicate*. Note that when this function is applied, the list of *TErule*s (*list_TErule*) is one of the arguments. Thus, the information in the entire list of rules can be used to evaluate a constraint, as described further below.

As mentioned earlier, a list of *TErule*s and a list of *TEconstraint*s constitute a TEpla policy. This means that *TEcontraint*s append additional conditions on top of eligible accesses which are granted by the *TErule*s component of the policies. Taking into account this fact, administrators usually need to know access information from eligible accesses to express different security goals in predicates. Access information obtained from a list of *TErule*s can include, for example, the list of all destination *type*s from a particular source *type*, or the list of all source *type*s that can perform a specific *Permitted Action* on a particular destination *type*, or the list of all *Permitted Action*s on a specific destination *type*. Accordingly, policy writers need to have the list of *TErule*s of the policy from which they can extract valuable access information.

As a result, Algorithm (4) receives a list of *TErules* as its third argument (here *listTErule_Policy*) which includes all the *TErule*s of the policy whose second component includes the *TEconstraint* received as the first argument in the algorithm (i.e., *constraintRule*). As the algorithm shows, the list of *TErule*s (i.e., *listTErule_Policy*) is passed to the predicate function (here *PredicateFunc_ConstraintRule*) within the received *TEconstraint* so that administrators can acquire their intended access information in *PredicateFunc_ConstraintRule*. In other words, having this list of *TErules* contributes to providing enough information for policy writers to use when they are defining predicates in TEpla.

It is worth mentioning that the extracted information from a list of *TErule*s can form various *set*s of values which support basic set operations; such as intersection and union, and thus enable policy writers to exploit *set comparison* to express security conditions, which is a suitable formalism to express constraints [Jaeger and Tidswell, 2001]. Set comparison, for example, includes the concept of subset or set equality, or generally includes set operators, i.e., binary relationships on pairs of sets. The authors of [Jaeger and Tidswell, 2001] show that complex access control constraints such as *Separation of Duty* (*SoD*) [Stallings and Brown, 2018, Jaeger et al., 2003b], can be expressed by basic set operators. We will see an example of *SoD* constraints in Chapter 5.

After *listTErule_Policy*, the next argument to *PredicateFunc_ConstraintRule* is *typeList_ConstraintRule*, which is a list of *type*s that comes from *constraintRule*. The next four arguments of *PredicateFunc_ConstraintRule* are *cls_ConstraintRule*, *perm_ConstraintRule*,

sourceTypeQuery, and *targetTypeQuery*, which are equal to the *Object Class*, *Permitted Action*, and the source and destination *type* of queries respectively. These elements provide information from both the constraint and the query that can be used by the predicate. The two remaining arguments *typeArgn1_ConstraintRule* and *typeArgn2_ConstraintRule* are *type* which come from the *constraintRule*.

In Chapter 5, we use this feature in developing two sample predicates for a TEpla policy that express specific security goals which must be checked. These predicates search the list of *TErule*s, which they receive as one of the arguments, to find particular access information.

In the first example, we develop a predicate for *SoD* constraints, which ensure that two specific *type*s cannot be accessed by the same entities (i.e., the security contexts of the resources that access these *type*s have to be different). This ensures that distinct *type*s can access these two different destination *type*s.

In the second example predicate, we develop a predicate for ensuring that a *type* T_1 can access to another type T_2 if the common set of types that T_1 can use (e.g., by action read) and modify T_2 (e.g., by action write) is restricted to a specified set of elements of *type*. In other words, the predicate checks whether or not the intersection of the specified set of types that are accessible from *type* T_1 and the set of elements of *type* that change *type* T_2 is a subset of the authorized set of elements of *type*.

Algorithm 4 Algorithm for evaluating a TEpla TEconstraint rule and a Query

```
1: procedure TRANSLATION FUNCTION 4 (TRFU4) OF TEPLA TECONSTRAINT
   RULE (constraintRule, query, listTErule_Policy)
2:     cls_ConstraintRule ← first component of constraintRule
3:     perm_ConstraintRule ← second component of constraintRule
4:     typeArgn1_ConstraintRule ← third component of constraintRule
5:     typeArgn2_ConstraintRule ← fourth component of constraintRule
6:     typeList_ConstraintRule ← fifth component of constraintRule
7:     PredicateFunc_ConstraintRule ← sixth component of constraintRule
8:     listTErule_Policy ← first component of TEpolicy
9:     sourceTypeQuery ← first component of query
10:    targetTypeQuery ← second component of query
11:    cls_Query ← third component of query
12:    perm_Query ← fourth component of query
13:    if cls_ConstraintRule isEqual cls_Query then
14:        if perm_ConstraintRule isEqual perm_Query then
15:            if PredicateFunc_ConstraintRule(listTErule_Policy, typeList_ConstraintRule,
                       cls_ConstraintRule, perm_ConstraintRule,
                   sourceTypeQuery, targetTypeQuery, typeArgn1_ConstraintRule,
                   typeArgn2_ConstraintRule) isEqual true then
                return Permitted
               else return UnKnown
17:        else return NotPermitted
18:    else return NotPermitted
```

In Algorithm (4), if the *TEconstraint* (here *constraintRule*) is not applicable to the query, then the algorithm returns *NotPermitted*. Algorithm (4) returns *Permitted* if *constraintRule* is applicable to the query and the query satisfies the condition expressed in the predicate function (here *PredicateFunc_ConstraintRule*) of *constraintRule*. However, if *constraintRule* is applicable but the query does not satisfy the condition expressed in the predicate function of the *constraintRule*, the algorithm returns *UnKnown*. Note that the function *PredicateFunc_ConstraintRule* is a function which is the sixth component of the input constraint *constraintRule*. Moreover, note that there is just one level for constraints (no nested constraints).

The fifth translation function, depicted in Algorithm (5), shows how to produce a list of decisions as specified by the list of *TEconstraints*. Each *TEconstraint* in the list is evaluated and a list of decisions is generated for the input query. This list of decisions is used to make the final access decision using the sixth algorithm.

Algorithm 5 Algorithm for evaluating a list of TEconstraints

1: **procedure** TRANSLATION FUNCTION 5 (TRFU5) OF TEPLA LIST OF TECONSTRAINT (*list_TEconstrain, query, listTErule_Policy*)
2: $n \leftarrow$ number of TEconstraints in *list_TEconstraint*
3: *TEconstraint_sequence* $[n] \leftarrow$ *list_TEconstraint*
4: *decisionList* $[n] \leftarrow$ list of decisions of size n
5: $i \leftarrow$ n
6: **while** $(i) > (0)$ **do**
7: *decisionList* $[i] \leftarrow$ TrFu4 *(TEconstraint_sequence* $[i]$*, query, listTErule_Policy)*
8: $i \leftarrow i - 1$.
9: **end-do.**
10: **return** *decisionList*

3.3.3 Semantics for TEpla Policies (*TEpolicy*)

The semantics algorithms, which we represented, map different parts of policies into decisions. It is thus needed to combine these decisions as to the final decision for an access request. The sixth algorithm, shown in Algorithm (6), describes the steps for reaching the final decision based on the results of two parts of a *TEpolicy*. The algorithm takes a policy and a query and maps them to a decision. We present *MaximalElement* in the next section.

Algorithm 6 Algorithm for evaluating a TEpolicy and a Query

1: **procedure** TRANSLATION FUNCTION 6 (TRFU6) OF TEPLA POLICY (*TEpla_policy, query*)
2: *TErule_Policy* $\leftarrow$ first component *TEpla_policy*
3: *TEconstraint_Policy* $\leftarrow$ second component of *TEpla_policy*
4: *maxTErule* $\leftarrow$ MaximalElement of (TrFu3 (*TErule_Policy, query*))
5: *maxTEconstraint* $\leftarrow$ MaximalElement of
(TrFu5 *(TEconstraint_Policy, query, TErule_Policy)*
6: **if** *maxTErule* isMemberOf $\{UnKnown, Permitted\}$ **then**
7: **return** MaximalElement*(maxTErule, maxTEconstraint)*
8: **else** **return** *NotPermitted*

3.4 Formal Language Properties of TEpla

By having a formally defined syntax and semantics for TEpla, we can study the behavior of TEpla policies in terms of different properties. In this section, we focus on introducing a set

of formal properties which analyze the semantics of TEpla under different circumstances. These properties evaluate how decisions in TEpla can evolve as a result of certain changes in queries or policies. Furthermore, we assess the behavior of composition of TEpla policies as there are often plenty of demands to combine separate security policies of computer systems into a central security policy. Adding extra policy rules to regulate new resources can be another reason why composition of policies is important to evaluate in TEpla. These properties facilitate reasoning and verification about TEpla policies, which we believe gives crucial insight into how TEpla policies behave. In other words, we think these formal properties are necessary since the properties let policy developers as well as policy-related tools know upfront about the behaviors of policies or access requests in TEpla, moreover, they provide mathematical guarantees of the behavior of TEpla's main structures (i.e., policies, access queries, and decisions).

As a consequence, these sets of properties allow us to reason about the behavior of policies with regard to analyzing changes to policies (integration or decomposition of policies), queries (adding or removing information to/from certain parts of queries). As mentioned earlier, under certain circumstances (refer to ordering relations on policies, decisions, and queries, which are defined in sections 3.4.1 and 3.4.2), decisions can evolve in the particular order, preventing erratic behavior of policies after granting or denying requests. Developing more properties and enhancing the current properties are left as our future work.

We follow the style of [Tschantz and Krishnamurthi, 2006] to define this set of formal properties for TEpla. They define *Safety*, *Monotonicity*, *Independent Composition*, *Determinism* properties for policy languages, as discussed in Section 2.7 for SELinux. In case of any difference with our definitions, we compare our encoding of the properties with the counterpart properties in [Tschantz and Krishnamurthi, 2006] and give the reasons why we think our definitions are better for TEpla or any other policy language. Despite the similarities in our definitions of properties, we provide different names for the *safety* and *monotonicity* properties. These properties do not aim to evaluate whether or not a language can establish a safe system, or if a language is totally non-increasing or non-decreasing; so we rename them to avoid giving an incorrect impression of the purpose of the properties.

Specifically, for the *safety* property, we define a particular operator to compare queries, rather than generally state that this property uses more information to compare queries as it is defined in [Tschantz and Krishnamurthi, 2006]. This is because the way *safety* is defined is open to different interpretations as it does not clearly specify enough information to define an ordering on more versus less information. In their examples [Tschantz and Krishnamurthi, 2006], for instance, they evaluate the *Core XACML* (CXACML) and *FOL* policy languages [Tschantz and Krishnamurthi, 2006] to determine whether or not they satisfy the *safety* property, but different parts of queries are used for comparing queries in these two policy languages (see subsection 3.4.4 for more details). In particular, *safety* in CXACML involves a comparison of certain components of queries, while *safety* in FOL involves different ones. Our definition of *safety* involves only the source and destination of queries. Subsection 3.4.4 provides more details and the way we define this property.

Recall that we showed that SELinux is not *safe* in Section 2.7. Perhaps our original analysis of this language involved choosing the wrong information to compare, leading to our conclusion (and proof) that SELinux is not *safe*. We do not pursue further analysis of SELinux. Instead, we focus on a definition of *safety* for TEpla that encompasses information relevant to the notion of *safety* for this language.

The next subsections present TEpla's formal properties in forms of mathematical theorems. The theorems formally state the specific conditions by which we evaluate TEpla's behavior. For two of the theorems, some related lemmas are presented to show the overall blueprint of the way we will use to prove these properties in Chapter 4. It is clear that the lemmas we introduce in this chapter help us to prove theorems but are not sufficient to prove the theorems. In the next chapter, we present more details for these theorems and use the Coq proof assistant to prove the properties, thus presenting machine-checked proofs for lemmas and theorems.

TEpla's properties are based on the ordering relations we define on policies, queries, and decisions. Thus, before describing the properties, in subsections 3.4.1 and 3.4.2, we introduce three operators on decisions, policies, and queries through defining ordering relations on these sets. In the following subsections " $\Longrightarrow$ " and "$\wedge$" signify *implication* and logical *And*, respectively.

3.4.1 Ordering Relation on Decisions and Policies

As mentioned earlier, decisions in TEpla are a three-valued set which we denote here by (D). We define the binary relation "<::" on this three-element set as $NotPermitted$ <:: $Permitted$ <:: $UnKnown$ to define the *Partial Ordered Set* [Huth and Ryan, 2004] (D,<::). The *hasse diagram* for this partial ordered set (poset) is shown in Fig. 3.2. Note that "$\leq$" in Section 2.7 for SELinux is defined as the same ordering, except that SELinux does not have an *UnKnown* decision (*Denied* and *Granted* decisions are the same as *NotPermitted* and *Permitted* respectively).

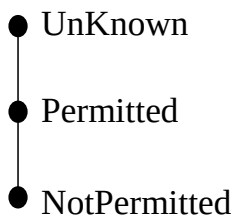

Figure 3.2: Hasse diagram for poset (D,<::)

The hasse diagram in Fig. 3.2 is aligned with the interpretations of decisions we expressed in subsection 3.2.1. Taking into account the *closed-world assumption* in TEpla's decisions, the lowest decision in this hasse diagram is thus *NotPermitted*. That is, all accesses are denied by default. To permit an access query, a relevant access rule in the first part of TEpla policies (*TEpolicy*) must authorize the access. If the query is not granted through *TErule* part of polices, TEpla denies the access, which means that the ultimate access decision is *NotPermitted*. In the case that the query is granted by at least one *TErule* (i.e., one rule authorizes the query with *Permitted*), TEpla proceeds to check whether or not the query satisfies the *TEconstraint* part of policies. The decision for the query continues to be *Permitted* as long as it satisfies the constraints; if not, that is the query fails to satisfy some constraints, the decision eventually changes to *UnKnown*.

Under specific conditions of queries or policies, which we introduce in TEpla formal properties, the decisions can only evolve in the direction of *NotPermitted* to *UnKnown* shown in Fig. 3.2. Changing *UnKnown* or *Permitted* decisions to *NotPermitted* decisions, in languages that allow this, could be the consequence of adding sub-policies, for example. However, in many contexts, going from *UnKnown* or *Permitted* decisions to *NotPermitted* decisions is considered risky. This is dangerous because once access is permitted, the permit receiver can see the information and there is no way to revoke the seen information. In TEpla, we allow composition of policies in which decisions never go from *UnKnown* or *Permitted* to *NotPermitted* when TEpla checks the sub-policies of the composed policy (see subsection 3.4.5 for more details about this property).

Recall that in Section 3.3, for a given query, each rule in a policy resulted in a decision, and we applied *MaximalElement* to this set of decisions to make the final decision about this query. We use the hasse diagram in Fig. 3.2 to define *MaximalElement* of a sequence of decisions $(d_1, \ldots, d_n)$, denoted by d_{max}, as the following definition:

Definition 5

$$(\forall\ d_i \in d_{1..n},\ d_{max} <:: d_i \implies (d_{max} = d_i)))$$

We define the binary relation $\lesssim$ on two policies p_1, p_2, where $p_1 \lesssim p_2$ whenever p_2 has more information than p_1. More formally:

Definition 6

$$p_1 \lesssim p_2 \implies length(p_1) \leqslant length(p_2) \wedge p_1 \subseteq p_2$$

Specifically, the binary relation $p_1 \lesssim p_2$ holds iff p_2 has more *authorization rules* and it contains all the authorization rules in p_1. By authorization rules here we mean *TErule* or *TEconstraint* elements of TEpla policies (denoted by P). This gives the poset $(P, \lesssim)$ which we use in defining TEpla's theorems.

3.4.2 Ordering Relation on Queries

We use Algorithm (7) to define the binary relation $\trianglelefteq$ between queries in TEpla. Note that the notation $fx_1, x_2, ..., x_n$ means function f is applied to n arguments $x_1, x_2, ..., x_n$. In TEpla, two queries $q_1 = (Sourcetype\_q1, Destinationtype\_q1, cls, perm)$ and $q_2 = (Sourcetype\_q2, Destinationtype\_q2, cls, perm)$ have relation $q_1 \trianglelefteq q_2$ if $isSubset\ Sourcetype\_q2\ Sourcetype\_q1$ and $isSubset\ Destinationtype\_q2\ Destinationtype\_q1$ (see the definition of *isSubset* in section 3.3.1); in other words, if they have equal *cls*, *perm* components and the $Sourcetype\_q2$, $Destinationtype\_q2$ of q_2 are a subset of the $Sourcetype\_q1$, $Destinationtype\_q1$ of q_1, respectively.

Algorithm 7 Algorithm for the Binary Relation $\trianglelefteq$ for TEpla Queries

```
1: procedure RELATION ⊴ IN TEPLA (querya ⊴ queryb)
2:     sourceQuerya ← first component of querya
3:     sourceQueryb ← first component of queryb
4:     destinationQuerya ← second components of querya
5:     destinationQueryb ← second components of queryb
6:     objectClass_Querya ← third components of querya
7:     objectClass_Queryb ← third components of queryb
8:     perm_Querya ← last component of querya
9:     perm_Queryb ← last component of queryb
10:    if sourceQueryb isSubset sourceQuerya then
11:        if destinationQueryb isSubset destinationQuerya then
12:            if objectClass_Queryb isEqual objectClass_Querya then
13:                if perm_Queryb isEqual perm_Querya then    return true
14:                else    return false
15:            else    return false
16:        else    return false
17:    else    return false
```

Using the binary relation $\trianglelefteq$ on queries (Q), we define the poset $(Q, \trianglelefteq)$ which we use in TEpla's theorems.

3.4.3 Semantics of TEpla as a Homomorphism

In the next few subsections, we express the formal properties of TEpla. We evaluate the behavior of TEpla through studying how TEpla semantics acts as a mapping between the posets we defined in previous subsections. In other words, defining posets $(P, \lesssim)$, $(Q, \trianglelefteq)$ and $(D, <::)$ allows us to evaluate the behavior of TEpla semantics.

Algorithm (6) is the core mapping function of the TEpla semantics as it includes other translation functions in its logic. Algorithm (6) is a function that maps a policy and a query to a decision (i.e., Algorithm (6): $(P, Q) \rightarrow D$). The behavior of Algorithm (6), as the core translation function of the semantics, specifies the overall semantics of TEpla. TEpla has in fact been designed so that its semantics as designed by Algorithm (6) is *order-preserving* for the relation $\lesssim$ on policies, $\trianglelefteq$ on queries, and $<::$ on decisions, as we will see. This means that Algorithm (6) acts as a *homomorphism* [Huth and Ryan, 2004, Harzheim, 2005] on the posets we defined on P, Q and D.

As mentioned, in the following subsections, we introduce a set of properties of TEpla, which are based on ordering relations we defined in subsections 3.4.1 and 3.4.2.

3.4.4 Order Preservation of TEpla Queries

Of particular importance is the preservation of the order of decisions with respect to queries. In particular, if two queries $q_1 \trianglelefteq q_2$, then the resulting decisions after applying Algorithm (6) to these queries, d_1 and d_2 respectively, are in the relation $d_1 <:: d_2$. That is, Algorithm (6) preserves the relations of posets $(Q, \trianglelefteq)$ and $(D, <::)$ in the mapping between the elements of Q to D. Our definition of ordering for queries involves only source and destination, which are elements that queries in any language must have. We do so in order to have as general a definition as possible. When policies are large, verifying policies often involves testing a number of queries against the policy. In a policy language with this property, undue access, i.e., bypassing security checks or unauthorized access to internal data [Parrend and Frénot, 2008], is impossible for queries that have incomplete information [Tschantz and Krishnamurthi, 2006], which is helpful for policy reasoning. In addition, having an unambiguous ordering of queries facilitates sorting, filtering, and optimizing query evaluations. As we defined how we compare queries in Algorithm (7), we believe that any operation that compares queries in policy languages must use source and destination components of queries in its logic. This is because source and destination components are the inherent components, thus may not be omitted, of queries in policy languages such as TEpla.

This property can be compared to the *safety* property defined in [Tschantz and Krishnamurthi, 2006]. The authors define queries as (s, r, a, e) which denote *Subject*, *Resource* (i.e., *objects*), *Action*, and *Environment* (i.e., other relevant information, or the previous actions of the *subjects*) elements respectively. They define the partial ordering "$\sqsubseteq$" to compare two queries $q_1 \sqsubseteq q_2$ if q_2 has more information than q_1. In the examples of [Tschantz and Krishnamurthi, 2006], *CXACML*, for instance, uses all parts of the queries (not just source and destination) for the comparison operation. According to this definition *CXACML* is not *safe*. On the other hand, the comparison for *FOL* involves only the environment components of queries, and *FOL* does satisfy *safety*. This fact makes the behavior of this operator different from a function, thus implementing this operator as a function is demanding. In addition, considering the consequences of adding information to granted queries is one of the main factors in defining "$\sqsubseteq$", however, after granting queries, it is unlikely that the

same queries will be modified to contain more information because this usually happens through adding security regulations in policies, not queries. Our interpretation of adding information is to add other authorization rules to policies, which we discuss in the next subsection. In general, operations have to act as functions with specific inputs and logic rather than depending them on different interpretations and scenarios that are rather unlikely. We believe our relation to compare queries "$\trianglelefteq$" is more applicable for analyzing queries in TEpla as well as general enough to apply to other policy languages.

- Order Preservation of queries of Policy Rules

  Here we introduce the first lemma which helps us to prove order preservation of TEpla queries. As mentioned earlier, TEpla policies (P) consist of two parts *TErules* and *TEconstraints* (denoted as r, c respectively). Thus, it is logical to prove this property for the elements of these components: a *TErule* and a *TEconstraint*. The first stepping stone is to prove this property for a *TErule*. Lemma 1 shows order preservation of queries for a *TErule*. In particular, if two queries q, q' have the relation $\trianglelefteq$, then the decisions of applying Algorithm (1) and (2) with same *TErule*, have the relation $<::$, as expressed in Lemma 1. We prove this lemma for TEpla in Chapter 4.

  Lemma 1

  $$\forall\ (q, q' \in Q), (r \in TErule),\\ q \trianglelefteq q' \implies Algorithm\ 1,2\ (r, q) <:: Algorithm\ 1,2\ (r, q').$$

- Order Preservation of queries of Constraints

  After expressing order preservation of queries for a *TErule*, now it is time to consider the corresponding property for a *TEconstraint*. Lemma 2 expresses the order preservation of queries for a *TEconstraint*. In particular, if two queries q, q' have the relation $\trianglelefteq$, then the decisions of applying Algorithm (4) with the same list of *TErules* and *TEconstraint*, have the relation $<::$, as expressed in Lemma 2. Note that for proving Lemma 2, and thus Theorem 1 for TEpla, in addition to the premise $q \trianglelefteq q'$, we need another premise to express that constraints contain only the predicates that satisfy a set of conditions that we introduce in the next chapter. These conditions specify the behavior of predicates in regard to particular changes in policies and queries, the details of which are presented in Section 4.3.

  Lemma 2

  $$\forall\ (q, q' \in Q), (c \in TEconstraint), (l \in list\ TErule),\\ q \trianglelefteq q' \implies Algorithm\ 4\ (c, q, l) <:: Algorithm\ 4\ (c, q', l).$$

- Order Preservation of queries for Policies

  We now express the order preservation of queries as a theorem for TEpla policies. Proving this theorem requires lemmas 1 and 2.

  Theorem 1

$$\forall\ (p \in P), (q, q' \in Q),$$
$$q \trianglelefteq q' \implies Algorithm\ 6\ (p, q) <:: Algorithm\ 6\ (p, q').$$

Theorem 1 expresses the order preservation of TEpla queries. As the theorem states, for the same policies, queries with the relation $\trianglelefteq$ render to decisions with the relation $<::$. By presenting this theorem, it is now time to introduce the second formal property of TEpla which presents another feature of TEpla semantics.

3.4.5 Non-Decreasing Property of TEpla Policies

It is of crucial importance to re-evaluate the behavior of polices after adding new policy statements, which try to accommodate recent regulations of the system. In this view, combining a different sub-policy with policies has the same results as adding new policy statements to the policy. In TEpla, by adding new policy statements in the form of access rules to a TEpla policy, previously rendered decisions in TEpla can just evolve to higher decisions in the order we defined in Fig. 3.2 and thus there is no oscillation in this order for decisions. The consequence of this property is interesting for policy developers as they know ahead the behavior of policies after adding new access rules or combining them with a sub-policy.

Therefore, this property states that TEpla policies do not change their decisions in the reverse direction of Fig. 3.2. Changing decisions of policies, for example, from *Permitted* to *NotPermitted* is impossible (assuming that policies grow by adding new access rules). This is highly beneficial particularity in the case of revoking access for granted requests. As we mentioned, revoking access from already granted requests is problematic because once the information has been revealed, there is no way to reverse the effect of revealing this information. This cannot happen in TEpla, based on this property. The property is aligned with *monotonicity* defined in [Tschantz and Krishnamurthi, 2006].

- Non-Decreasing property for Policy Rules

  Similar to the lemmas we used to prove order preservation of queries in TEpla policies, we introduce selected sample lemmas which are related to different parts of TEpla policies to prove non-decreasing properties of TEpla policies. As for the previous property, the first stepping stone is to prove this property for *TErules*. However, for this property, we use a list of *TErules*. In Lemma 3, we state that a non-decreasing

property holds for any two sequences of *TErules* which have similar relations as the operator "$\lesssim$" for policies defined in subsection 3.4.1. In other words, if there are two sequences of *TErules* l, l' such that l' contains the same *TErule*s as l and possibly more, then the decisions of Algorithm (3) for these sequences, d and d' respectively, have the relation $d <:: d'$.

Lemma 3

$$\begin{aligned}&\forall\ (l, l' \in list\ TErules), (q \in Q), (d, d' \in D),\\ &\quad length(l) \leq length(l') \wedge\ l \subseteq l'\ \wedge\\ &(Algorithm\ 3\ (l, q) = d \wedge Algorithm\ 3\ (l', q) = d')\ \Longrightarrow\ d <:: d'.\end{aligned}$$

- Non-Decreasing property for Constraints

  TEconstraints components of policies need to have the same behavior as the non-decreasing property for *TErule*s. Lemma 4 expresses the non-decreasing property for sequences of *TEconstraint*s as well as *TErule*s. As Lemma 4 states, if we have two sequences of *TErule*s l, l' such that l' contains the same list of *TErule*s as l and possibly more, and similarly if we have two sequences of *TEconstraints* c, c' such that c' has the same list of *TEconstraints* of c and possibly more, then the decisions of Algorithm (5) on these sequences (i.e., d, d') have the relation $d <:: d'$. Note that, similar to Lemma 2 and Theorem 1, for proving Lemma 4, and thus Theorem 2 for TEpla, we need another premise about the predicates of constraints, the details of which are presented in Section 4.3.

  Lemma 4

$$\begin{aligned}&\forall\ (l, l' \in list\ TErules), (c, c' \in list\ TEconstraint)(q \in Q), (d, d' \in D),\\ &\quad length(l) \leq length(l')\ \wedge length(c) \leq length(c')\ \wedge\ \ l \subseteq l'\ \wedge\ c \subseteq c'\ \wedge\\ &(Algorithm\ 5\ (c, q, l) = d \wedge Algorithm\ 5\ (c', q, l') = d')\ \Longrightarrow\ d <:: d'.\end{aligned}$$

- Non-Decreasing property for Policies

  We now express the non-decreasing property for policies as a theorem for TEpla policies. Proving this theorem requires lemmas 3 and 4. We prove this lemma for TEpla in Chapter 4.

  Theorem 2

$$\begin{aligned}&\forall\ (p, p' \in P), (q \in Q), (d, d' \in D),\\ &\quad p \lesssim p'\ \wedge\ (Algorithm\ 6\ (p, q) = d\ ,\ Algorithm\ 6\ (p', q) = d')\ \Longrightarrow\ d <:: d'.\end{aligned}$$

Theorem 2 expresses the non-decreasing property for TEpla policies. As the theorem states, the decisions for policies p, p' with $\lesssim$ relation (d, d' respectively) are always non-decreasing as they have the relation $d <:: d'$.

3.4.6 Independent Composition of TEpla Policies

In the domain of access control policies, it is crucial to be able to analyze the behavior of policies based on the components of the policies (i.e., sub-policies) since it will be beneficial to reason about the whole policy by evaluating sub-policies. As we mentioned, this demand can happen regularly in computer systems. Therefore, knowing how combining different sub-policies can affect the final decision of the whole policy is another appropriate property to evaluate in TEpla. This helps in analysis of policies (i.e., combined policies) which are made up of other policies (i.e., sub-policies), as the decisions for the combined policies can be determined from the decisions of included policies. The TEpla policy combinator "$\oplus$" is a logical operator which connects sub-policies to constitute one single TEpla policy by putting *And* logic between sub-policies. Similar to *independent composition* in [Tschantz and Krishnamurthi, 2006], we codify this property in TEpla as the following theorem.

Theorem 3

$$\begin{aligned}&\forall\ (p_1,\ldots,p_n \in P),(q \in Q),(d_1,\ldots d_n,d^* \in D),\\ &\quad((Algorithm\ 6\ (p_1,q) = d_1 \wedge \quad \cdots \quad \wedge\ Algorithm\ 6\ (p_n,q) = d_n\ \wedge\\ &\qquad Algorithm\ 6\ (\oplus(p_1,\ldots,p_n),q) = d^*)\ \wedge\\ &\qquad (\forall\ d_i \in d_{1..n},\ d_\oplus <:: d_i\ \Longrightarrow\ (d_\oplus = d_i)))\ \Longrightarrow\ d_\oplus <:: d^*.\end{aligned}$$

Theorem 3 expresses the independent composition of TEpla policies. The theorem states that decisions of policies can be specified by the included composed sub-policies. In particular, if we have a set of policies $p_1, p_2, ..., p_n$ which result in $d_1, d_2, ..., d_n$ decisions respectively, then the decision after combining these policies by the TEpla policy combinator $\oplus(p_1,\ldots,p_n)$, d^*, is in the relation $d_\oplus <:: d^*$ in which $d_\oplus$ is equal to the *MaximalElement* of the sequence of decisions $d_1, d_2, ..., d_n$. This theorem extends Theorem 2 to policy compositions.

3.4.7 Determinism of TEpla

TEpla is a deterministic language which means TEpla always produces the same decision for the same policies and queries. Theorem 4 states that TEpla renders the same decision for all policies and queries that are not changed.

Theorem 4

$$\begin{aligned}&\forall\ (p \in P),(q \in Q),(d,d' \in D),\\ &\quad(Algorithm\ 6\ (p,q) = d \wedge Algorithm\ 6\ (p,q) = d')\ \Longrightarrow\ d = d'.\end{aligned}$$

3.5 Summary

We presented the building blocks of TEpla; the syntax of TEpla was given in the BNF form, and the semantics of TEpla was introduced through mapping functions from policies and access requests to decisions. In addition, we explained the major formal properties of TEpla through different lemmas and theorems. In the next chapter, we provide the details of how we implement TEpla and prove the formal properties of its behavior by using the Coq proof assistant.

Chapter 4

The Coq Development

In this chapter, we describe the Coq implementation of the infrastructure of TEpla (i.e., the elements of syntax and semantics of TEpla described in Chapter 3) as well as proofs of properties related to the certification of TEpla as a TE policy language, which are the properties presented in Section 3.4. The whole Coq development of TEpla contains approximately 4700 lines of Coq (including lemmas, comments, and blank lines) and is available on the Web [Eaman, 2019]. All the proofs of lemmas and theorems of TEpla are available in the Coq source of TEpla. We use version 8.10 of Coq.

4.1 Overview

We use the Coq proof assistant to write machine-checked mathematical proofs for TEpla's properties. The benefits of Coq for certified programming [Chlipala, 2019] including features such as *dependent types*, *proof by reflection*, *programmable proof automation*, *support for induction* and *polymorphism*, make Coq a suitable tool for developing TEpla.

The first step of formalizing TEpla is to use the logic of Coq to define the basic elements of TEpla (i.e., syntax and semantics). Coq has two important features that are especially useful for our task, the ability to define functions and inductive types [Chlipala, 2019]. We use these features for specifying TEpla in Coq. In addition, lemmas and theorems related to the properties of TEpla are encoded in Coq's logic.

Expressing these properties and developing proof scripts for the lemmas and theorems is the next step for mechanizing TEpla in Coq. Proof scripts are developed as sequences of tactics, which have specific mathematical meaning in the context of manipulating logical goals in Coq. As mentioned in Chapter 2, Coq's tactics are commands which implement proof steps and proof strategies.

For facilitating the reuse of proof scripts across a variety of TEpla's proofs, we selectively break down the proof of properties into lemmas. This allows us not only to have reusable logical results but also to focus the development of proof scripts on smaller goals.

4.2 Defining TEpla in Coq

In this section, we begin by encoding the syntax and semantics of TEpla in Coq's logic. In addition, we present the conditions on predicates that all definitions of TEpla predicates (elements of `TEpredicate_ID` in the grammar in listing 3.1) must satisfy.

4.2.1 Syntax in Coq

The first step to implement TEpla in Coq is defining the datatypes. The datatypes *basictype*, *cls*, *prm* (in the BNF grammar of TEpla described in listing 3.1) are defined as type `nat` (depicted in listing 4.1) which is the datatype of natural numbers from the Coq library `Coq.Init.Datatypes` [The Coq Development Team, 2019]. According to listing 4.1, we define the types `class`, `permi`, and `basictype` to encode *cls*, *prm*, and *basictype* respectively. In listing 4.2, we define some possible instances of these types. Here, `mail_t`, `http_t`, `print_exec_t` and `indexhtml_t` are instances of type `basictype`, `Net_socket`, `File` and `Process` are instances of type `class`, and `Read`, `Signal` and `Entrypoint` are instances of type `permi`.

```
Definition class := nat.
Definition permi := nat.
Definition basictype := nat.
```

Listing 4.1: Defining datatypes by using `nat`

```
Definition Net_socket : class := 101.
Definition File : class := 102.
Definition Process : class := 103.

Definition mail_t : basictype := 200.
Definition http_t : basictype := 201.
Definition print_exec_t : basictype := 202.
Definition indexhtml_t :basictype := 203.
Definition readme_t : basictype := 204.

Definition Read : permi := 500.
Definition Signal : permi := 501.
Definition Entrypoint : permi := 502.
```

Listing 4.2: Some instances of the datatypes defined in listing 4.1

Recall from Section 3.2 that *types* in TEpla consist of *basictype*, which defines the security context of entities and *attribute*, which conceptually group a number of *basictypes* to provide a single identifier for the group.

We encode *attribute* by the definition `attribute`, in listing 4.3.

```
Definition attribute : Set :=
      list (basictype).
```

Listing 4.3: Defining `attribute` as a list of `basictype`

Two examples of `attribute` are given in listing 4.4. We define `attr_prg_type` of type `attribute` by making a list of length two with two instances of basictype as its elements. By using the notations mentioned in Section 2.9, `(mail_t) :: (print_exec_t) :: []`, in listing 4.4, represents a list with two elements.

```
Definition attr_prg_type : attribute :=
                           (mail_t) :: (print_exec_t) :: [].
Definition attr_inter_type : attribute :=
                           (http_t) :: (indexhtml_t) :: [].
```

Listing 4.4: Defining the *attribute*s `attr_prg_type` and `attr_inter_type`

By encoding the two syntax classes *basictype* and *attribute*, we now can define *type*. As in listing 4.5, we define the inductive datatype `type` with two constructors `singletype` and `grouptype`. These constructors take arguments of type `basictype` and `attribute` respectively to produce a term belonging to `type`.

```
Inductive type : Type :=
  | singletype : basictype → type
  | grouptype : attribute → type.
```

Listing 4.5: Defining the inductive datatype `type`

Listing 4.6 defines two instances of `type`. The first example is `prg_Type` which is a type produced by applying the constructor `singletype` to the basictype `mail_t`. Similarly, the second example is `prg_attr_Type` which is the result of applying the constructor `grouptype` to the attribute `attr_prg_type`.

```
Definition prg_Type : type := singletype mail_t.
Definition prg_attr_Type : type := grouptype attr_prg_type.
```

Listing 4.6: Defining two instances of `type`

We encode *dcs* by the inductively defined type `answer`, defined in listing 2.5 of Section 2.9 when we introduced inductively defined types in Coq. There are three possible decisions for queries in TEpla, `Permitted`, `NotPermitted`, and `UnKnown` and thus there are three constructors of type `answer`.

The inductively defined type `TErule`, in listing 4.7, encodes *TErule*. As TEpla has two kind of rules for the first component of policies (see Section 3.2.2), `Allow` and `Type_Transition` are two constructors of type `TErule`. `Allow` rules are implemented as a tuple using the *pair* structured datatype from the Coq standard library `Coq.Init.Datatypes` which can be used to define tuples. The argument to `Allow` is a 5-tuple with five components consisting of *type*, *type*, *cls*, *prm*, and *cond_bool*. Similar to this, `Type_Transition` rules are defined as a tuple with three components *type*, *type*, and *cls*.

```
Inductive TErule : Set :=
  | Allow : type * type * class * permi * bool → TErule
  | Type_Transition : type * type * class → TErule.
```

Listing 4.7: Defining the inductive type `TErule`

The first definition in listing 4.8 constructs an instance of the type `bool`, `condition_bool`, which we use as the fifth component of the example `Allow` rules. The next four definitions in listing 4.8 demonstrate four instances of type `TErule` including `TEruleA`, `TEruleB`, `TEruleC`, and `TEruleD`.

```
Definition condition_bool : bool := true.

Definition TEruleA : TErule :=
   Allow (singletype http_t, singletype mail_t, File, Read, condition_bool).

Definition TEruleB : TErule :=
   Allow (grouptype attr_inter_type, singletype print_exec_t,
          Sockets, Open, condition_bool).

Definition TEruleC : TErule :=
   Allow (singletype readme_t, grouptype attr_prg_type, File,
            Signal, condition_bool).

Definition TEruleD : TErule :=
  Type_Transition (singletype http_t, singletype print_exec_t,
                   Process).
```

Listing 4.8: Examples of `TErule`

The inductively defined type `TEconstraint`, in listing 4.9, encodes *TEconstraint*. `TEconstraint` rules are implemented as a 6-tuple consisting of *cls*, *prm*, *type*, *type*, list of *type*, and the sixth component is a predicate function with eight arguments (see subsection 3.2.7 for more details). The arguments of the input predicate function are list of *TErule*, list of *type*, *cls*, *prm*, and four *type*s. The predicate function returns `bool` which is shown as its return type.

```
Inductive TEconstraint : Set :=
  | Constraint : class * permi * type * type * list type *
           (list TErule→  list type → class→  permi→
             type→  type→  type→  type→  bool) → TEconstraint.
```

Listing 4.9: Defining the inductive type `TEconstraint`

Here we define an example predicate, which can be used as the sixth component of `TEconstraint`. The predicate function `Operation_predicate_TEpla`, in listing 4.10, is a predicate function with eight inputs, whose return type is boolean. In this predicate the source and destination *type*s of queries, provided by arguments `querySourcetype`, and `queryDestinationType` respectively, are compared against two input variables of `type`, provided by the arguments `not_authorized_types_source` and `not_authorized_types_destination`, which are regarded as untrustworthy `type`s.

The comparison checks whether or not the source and destination *type*s of queries are subsets of the unauthorized *type*s. For checking the subset of two *types*, we define function `typeSubset` expressed in listing 4.15 and 4.16, which will be explained later. The predicate returns false if the source and destination *type*s of the queries are a `typeSubset` of the input variables `not_authorized_types_source` and `not_authorized_types_destination` respectively. Note that this particular example predicate doesn't use its other inputs.

```
Definition Operation_predicate_TEpla
 (list_TErules:list TErule) (list_types:list type)(sClass:class)
 (sPermis:permi)
 (querySourceType queryDestinationType not_authorized_types_source
    not_authorized_types_destination :type) : bool :=

    if(typeSubset querySourceType not_authorized_types_source &&
 typeSubset queryDestinationType not_authorized_types_destination)
                    then  false
                      else true.
```

Listing 4.10: The predicate `Operation_predicate_TEpla`

By defining the predicate `Operation_predicate_TEpla`, we can use it in constraints such as `Constraint_Example` (depicted in listing 4.11).

```
Definition Constraint_Example : TEconstraint :=
   Constraint(File, Read, grouptype attr_prg_Type,
          grouptype attr_inter_Type, [], Operation_predicate_TEpla).
```

Listing 4.11: An example of `TEconstraint`

The constraint `Constraint_Example` is applicable to all queries whose *cls* and *prm* are `File` and `Read` respectively. Since the definition of predicate `Operation_predicate_TEpla` in listing 4.10 does not use the fifth argument of constraints (provided by the input variable `list_types`), we use an empty list [ ] as the fifth argument of the constraint `Constraint_Example`.

By inductively defining `TErule` and `TEconstaint`, which are the necessary components of TEpla policies, we therefore now can define policies in Coq as another datatype. The encoding of *TEpolicy*, `TEpolicy`, is shown in the definition of listing 4.12. We define *TEpolicy* in the form of a tuple with a pair of two components consisting of a list of elements of *TErule* and a list of elements of *TEconstraint*s.

```
Inductive TEpolicy : Set :=
      |TEPolicy : list TErule * list TEconstraint → TEpolicy.
```

Listing 4.12: The datatype `TEpolicy`

The policy example `TEpla_policy_ex` in listing 4.13 shows an instance of `TEpolicy`. We define `list_TErule` and `list_TEconstraint`, which are a list of `TErule` and a list of `TEconstraint` respectively, as the first and second components of `TEpla_policy_ex`.

```
Definition list_TErule : list TErule :=
                  TEruleA :: TEruleB :: TEruleC :: TEruleD :: nil.

Definition list_TEconstriants : list TEconstraint :=
                      Constraint_Example :: nil.

Definition TEpla_policy_ex : TEpolicy :=
               TEPolicy (list_TErule, list_TEconstriants).
```

Listing 4.13: `TEpla_policy_ex`, an instance of `TEpolicy`

In the first definition of listing 4.14, similar to the type `TErule`, a TEpla access request *qr* is defined by type `query` which is implemented as a tuple using the pair datatype with four components including *type*, *type*, *cls*, and *prm*. An instance of type `query` named `TEquery1` is given in the second definition of listing 4.14.

```
 (* TEpla Query *)
Definition query : Set :=
    type * type * class * permi.

Definition TEquery1 : query :=
   (singletype mail_t, singletype print_exec_t, File, Signal).
```

Listing 4.14: Defining `query` and the example `TEquery1`

Now, it is time to define the basic definitions we need to use in the semantics of TEpla. One of the most important relations that we need is to check whether or not one `type` is a subset of another one. We encode *isSubset* defined in Section 3.3.1 as `typeSubset` in listing 4.15 and 4.16.

```
(* checking the membership of basictype in an attribute *)
Fixpoint basictype_member_attribute
  (basic_type: basictype) (sattrib: attribute) : bool :=
 match sattrib with
  |(p_one):: bodyattrib ⇒ Nat.eqb basic_type (p_one) ||
           basictype_member_attribute basic_type bodyattrib
  |nil ⇒ false
end.



(* checking the equality (membership) of two attributes*)
Fixpoint attributeSubset
          (first_attribute second_attribute: attribute) : bool :=
 match first_attribute with
   |(x1)::body_attrib ⇒
       (basictype_member_attribute x1 second_attribute) &&
        attributeSubset body_attrib second_attribute
   |nil ⇒ true
  end.

Definition attributeEqual (first_attribute second_attribute:attribute) :=
    attributeSubset first_attribute second_attribute &&
        attributeSubset second_attribute first_attribute.


(* checking if an attribute's members are equal to a basictype *)
Fixpoint attribute_member_basictype
  (fattrib: attribute) (s_basic_type: basictype) : bool :=
 match fattrib with
  |(p_one)::bodyattrib ⇒
         Nat.eqb s_basic_type (p_one) &&
         attribute_member_basictype bodyattrib s_basic_type
  |nil ⇒ true
end.
```

Listing 4.15: The function `attributeEqual` and some other functions

To define `typeSubset`, we need to define the concepts of subset and equality on `basictype` and `attribute`. The functions in listing 4.15 including `basictype_member_attribute`, `attribute_member_basictype`, `attributeSubset`, and `attributeEqual` state the Coq encoding of the concepts of subset and equality on `basictype` and `attribute` datatypes. We use `Fixpoint`, which is the Coq nota-

tion for recursive functions, for defining these three functions. Accordingly, these functions are recursive functions. In each recursive step, we extract one element from the input parameters of type `attribute` and use it to obtain the result of the functions. Recall that we implement an `attribute` as a list of elements of `basictype`. The function `basictype_member_attribute` expresses that a `basictype` is a member of an `attribute` if it is equal to one of the elements of the `attribute`. The function `attributeSubset` states that an `attribute` is a subset of another if every `basictype` in the first `attribute` is a member of the second `attribute`. The function `attributeEqual` states that two `attributes` are equal if each one is a subset of the other. The function `attribute_member_basictype` states that an `attribute` is equal to a `basictype` if all of its members are the same and equal to `basictype`.

By defining these functions, we can implement `typeSubset` as a function in listing 4.16, whose definition is by cases and implements exactly the *isSubset* relation in Section 3.3.1.

```
(* checking the equality (membership) of two labels *)
Fixpoint typeSubset
  (first_type:type) (second_type:type) : bool :=

 match first_type with
  |singletype sgl_type⇒
     match second_type with
         |singletype sgl_t2 ⇒ (Nat.eqb (sgl_type)(sgl_t2))
         |grouptype grp_t2⇒  basictype_member_attribute sgl_type grp_t2
     end
 |grouptype grp_type ⇒
    match second_type with
       |singletype sgl_t2 ⇒ attribute_member_basictype grp_type sgl_t2
       |grouptype grp_t2⇒  attributeSubset grp_type grp_t2
    end
 end.
```

Listing 4.16: The function `typeSubset` to check subset of `types`

It is important to analyze the properties of `typeSubset` because many lemmas and theorems use them to express certain definitions that depend on these properties. Some properties of `typeSubset` and `attributeSubset`, including their transitivity in certain cases, are shown in listing 4.17.

```
Lemma transitivity_attribute_subset
  (a_attr b_attr c_attr: attribute):

   attributeSubset a_attr b_attr = true →
       attributeSubset b_attr c_attr = true →
           attributeSubset a_attr c_attr = true.

Lemma transitivity_type_subset
  (t1_type t2_type t3_type: type):

    typeSubset t1_type t2_type = true →
         typeSubset t2_type t3_type = true →
                  typeSubset t1_type t3_type = true.

Lemma typeSubset_transexcept
  (t1_type t2_type t3_type: type):

    typeSubset t1_type t2_type = true →
         typeSubset t1_type t3_type = false →
                  typeSubset t2_type t3_type = false.
```

Listing 4.17: Some properties of `typeSubset` and `attributeSubset`

To encode the binary relations for comparing decisions, and queries, i.e., $<::$, and $\trianglelefteq$, we implement two functions. We do not have a separate definition for $\lesssim$ as defined in Definition (6) of Section 3.4.1. Instead, we express it directly when needed using list operators. For example, if (*rules1*, *constraints1*) and (*rules2*, *constraints2*) are two policies, then we know that (*rules1*, *constraints1*) $\lesssim$ (*rules1* ++ *rules2*, *constraints1*++*constraints2*).

The encoding of $\trianglelefteq$ to compare queries, which is defined in Algorithm (7) (see section 3.4.2), is defined by `compare_queries` in listing 4.18. The infix symbol "<<=" is declared for denoting the function `compare_queries`. Testing the equality of instances of type `nat` is encoded by the polymorphic function `Entity_eq` which takes the type parameter `nat` as the first parameter to specify the type of the input arguments.

```
Definition compare_queries (qry qry' : query) :=
  match qry,qry' with
    |( sta, dta, cla, pera), (stb, dtb,clb ,perb) ⇒
                  ((typeSubset(stb)(sta)))=true ∧
                   (typeSubset(dtb)(dta))=true ∧
              ((Entity_eq (nat)(cla)(clb))= true) ∧
                  ((Entity_eq (nat)(pera)(perb))=true)

  end.

Notation "q <<= qPrime" := (compare_queries q qPrime) (at level 70).
```

Listing 4.18: Defining function `compare_queries` and the infix notation "<<="

For encoding "<::", which was used in Section 3.4.1 to compare decisions of TEpla, we defined the function `compare_decisions` in listing 2.6. We declare the infix notation "<::" for denoting the function `compare_decisions`.

```
Notation "d <:: dPrime" := (compare_decisions d dPrime) (at level 70).
```

4.2.2 Semantics in Coq

Referring to the six algorithms that we designed in Section 3.3, the semantics of TEpla, which is encoded in Coq in terms of translation functions, includes `TErule_EvalTE` in listing 4.19 for encoding Algorithms (1) and (2), `listTErule_EvalTE` in listings 4.20 and 4.21 for encoding Algorithm (3), `TEconstraint_EvalTE` in listing 4.22 for encoding Algorithm (4), `listTEconstraint_EvalTE` in listing 4.23 for encoding Algorithm (5), and `TEpolicy_EvalTE` in listings 4.24 and 4.25 for encoding Algorithm (6).

The translation function `TErule_EvalTE` (listing 4.19) translates a query into a decision indicating whether it is granted or denied depending on the given rule and the specifics of the query.

The function `TErule_EvalTE` decomposes the input policy rule and query into their components. Using these components to compare the policy rule and query, we can decide whether or not the policy rule is applicable to the query. As mentioned in Section 3.3.1, if the *cond_bool* of allow rules (denoted as `alw_boolCondition` in the function) is false, the allow rule is not applicable to queries.

As mentioned above, this translation function encodes both the Algorithms (1) and (2) because it includes both `Allow` and `Type_Transition` rules, which are constructors of

TErule. If the input rule is applicable to the query, the output decision is Permitted. Otherwise, the decision will be NotPermitted.

```
Definition TErule_EvalTE
  (TErule_policy:TErule) (q:query) : answer :=
 match TErule_policy with
  |Allow (alw_srcType,alw_dstType,alw_class,alw_permi,alw_boolCondition)
     ⇒
          match q with
            |( qsrct, qdst, qcl, qper)⇒
                   if ((typeSubset(qsrct)(alw_srcType)) &&
                       (typeSubset(qdst)(alw_dstType)) &&
                       (Entity_eq (nat)(qcl)(alw_class)) &&
                       (Entity_eq(nat)(qper)(alw_permi)) &&
                       (Bool.eqb alw_boolCondition true))
                   then
                        Permitted

                   else (*not matching with the current rule*)
                        NotPermitted
          end
  |Type_Transition (trn_sType, trn_dType, trn_class)⇒
          match q with
            |( qsrct, qdst, qcl, qper)⇒
                  if ((typeSubset(qsrct)(trn_sType)) &&
                      (typeSubset(qdst)(trn_dType)) &&
                      (Entity_eq (nat)(qcl)(trn_class)))
                  then
                        Permitted

                   else (*not matching with the current rule*)
                       NotPermitted
          end
end.
```

Listing 4.19: Evaluation of a TErule

As mentioned in Section 3.2.8, policies have lists of elements of TErule and TEconstraint respectively. We thus need a translation from a list of decisions, resulting from translations of a query and each rule in the lists, to the final decision. The encoding of *MaximalElement* (defined in definition 5 of Section 3.4.1) is defined in listing 4.20. The binary function maximalDcs takes two input decisions and returns the greater decision. If needed, we apply this function recursively to a list of decisions to find the final decision.

The criteria for finding the greater decisions are according to the partial order of decisions defined in Section 3.4.1.

```
Definition maximalDcs
 (ansFirst ansSecond : answer) : answer :=
 match ansFirst,ansSecond with
   | NotPermitted, _ ⇒ ansSecond
   | Permitted, Permitted ⇒ ansFirst
   | _, UnKnown ⇒ ansSecond
   | Permitted, NotPermitted ⇒ ansFirst
   | UnKnown,_ ⇒ ansFirst
  end.
```

Listing 4.20: Defining the function `maximalDcs`

The function `listTErule_EvalTE` (listing 4.21), the encoding of Algorithm (3), takes two arguments, a list of elements of `TErule` and a `query`. The function outputs an answer according to the contained rules in the input list of rules as well as the input query. That is the input query is evaluated against all the rules in the input list of rules, which are represented respectively by `qry` and `listTErule` in listing 4.21.

Note that unlike Algorithm (3), the code in listing 4.21 does not return a list. Instead, it processes the data directly without constructing the list and returns a single answer. By applying function `maximalDcs` to two decisions at each step, the function `listTErule_EvalTE` outputs the final decision. This is possible because we define the function `listTErule_EvalTE` as a recursive function. Note that the base case returns `NotPermitted` because the default decision for a query is to deny it.

```
(** _Evaluating_ a policy and a query produces an answer *)
Fixpoint listTErule_EvalTE
  (listTErule: list TErule) (qry: query) : answer :=
  match listTErule with
      |rule_h::rule_body ⇒
              maximalDcs (TErule_EvalTE rule_h qry)
                         (listTErule_EvalTE rule_body qry)
       |[]   ⇒ NotPermitted
 end.
```

Listing 4.21: Evaluating a list of `TErule` and a `query`

The function `TEconstraint_EvalTE` implementing Algorithm (4) takes three arguments of type `TEconstraint`, `query`, and list of `TErule`. As mentioned in Section

3.3.2, the argument of type list of TErule, expressed by the input argument listTErule, can be used to extract access information required for expressing security goals encoded in predicates. We use pattern-matching to obtain the components of the input constraint and query. In order to check whether or not the constraint is applicable to the query, their cls and prm components are compared. In case of equality of these components, the constraint is applicable.

```
Definition TEconstraint_EvalTE
  (constrain_rule:TEconstraint) (query_to_constr:query)
                          (listTErule:list TErule) : answer :=

 match constrain_rule with
  |Constraint (cstrn_class, cstrn_prm,cstrn_type_arg1,cstrn_type_arg2,
              cstrn_listType_arg,predicate_fn) ⇒
     match query_to_constr with
      |( query_src, query_dst, query_cls, query_prm)⇒

     if (Nat.eqb query_cls cstrn_class &&
         Nat.eqb query_prm cstrn_prm) then
         match (predicate_fn listTErule cstrn_listType_arg
                       cstrn_class cstrn_prm query_src query_dst
                       cstrn_type_arg1 cstrn_type_arg2) with
             |true ⇒ Permitted
             |false ⇒ UnKnown
         end
    else
           NotPermitted
     end

 end.
```

Listing 4.22: Evaluation of a TEconstraint

As it is depicted in the listing 4.22, the decision of the constraint depends on the result of the contained predicate. If the evaluation of the predicates returns true, the constraint's decision is Permitted. Otherwise, the decision is UnKnown. We here use Nat.eqb from the Coq library Coq.Arith.PeanoNat to check the equality of natural numbers. Recall that both the class and permi datatypes are defined as nat. Note that if the constraint is not applicable, by default, NotPermitted is returned.

Recall the example constraint in listing 4.11 which uses the example predicate in listing 4.10. When this constraint is passed as input to TEconstraint_EvalTE, this constraint provides the arguments grouptype attr_prg_type and grouptype attr_-

iner_type for the input variables not_authorized_types_source and not_authorized_types_destination of the predicate Operation_predicate_TEpla.

The function listTEconstraint_EvalTE, which implements Algorithm (5) (listing 4.23), translates a list of TEconstraint and a query into a single decision. Similar to the recursion on the input list argument in function listTErule_Eval, each recursive step considers one constraint from the input list of constraints (represented by listConstraint) and checks its answer for the input query (represented by qry) by sending them both as arguments to the function TEconstraint_EvalTE. The function maximalDcs (defined in listing 4.20) combines the decisions of constraints by applying its logic to two decisions at each step.

By defining this function, consequently, we have the translation functions for a list of elements of TErule (i.e., function listTErule_EvalTE) and a list of elements of TEconstraint (i.e., function listTEconstraint_EvalTE). Therefore, we can proceed with defining the translation function for a TEpla policy.

```
Fixpoint listTEconstraint_EvalTE
     (listConstraint: list TEconstraint) (q: query)
     (TEruleLst:list TErule) : answer :=

  match listConstraint with
    | rule_h::rule_body ⇒
        maximalDcs (TEconstraint_EvalTE rule_h q TEruleLst)
                   (listTEconstraint_EvalTE rule_body q TEruleLst)
    | [] ⇒ NotPermitted
   end.
```

Listing 4.23: Evaluating a list of TEconstraint and a query

For encoding Algorithm (6), we need to define the helper function maximalpolicy_Order defined in listing 4.24. This function applies the logic of combining the decisions of the two components of policies expressed in Algorithm (6) and Section 3.4.1.

The function maximalpolicy_Order compares the decision returned by the evaluation of the rules (the first component of policies) provided by the input argument CompOne with the decision returned by the evaluation of the constraints (the second component of policies) provided by the argument CompTwo. Before doing so, it checks whether or not CompOne is equal to NotPermitted because in this case NotPermitted must be returned. The evaluation of constraints does not change this decision. This check is represented by (Permitted <:: comOne) because the only decision that fails this condition is NotPermitted. On the other hand, if the condition is true which means the decision of the first component is greater than or equal to Permitted, i.e., the query is granted by at least one rule in the first component of policies, we must consider the decisions of the

second component of policies. In this case, the final decisions of policies are the result of the application of maximalDcs to the decisions of two components of policies.

```
Definition maximalpolicy_Order (compOne compTwo : answer) : answer :=
  if(Permitted <:: compOne) then
       maximalDcs compOne compTwo
  else
      NotPermitted.
```

Listing 4.24: Finding the decisions of policies

By defining maximalpolicy_Order, now we can define the translation function TEpoliy_EvalTE for a TEpla policy and a query. The function TEpoliy_EvalTE performs pattern-matching on the input parameter policy of type TEpolicy, resulting in having the two components of the policy as compOne_TEruleLst and CompTwo_TEconstraintsLst. These two lists are evaluated against the query by their relevant translation functions.

```
Fixpoint TEpolicy_EvalTE
 (policy: TEpolicy) (q: query) : answer :=

 match policy with
  | TEPolicy (CompOne_TErulesLst, CompTwo_TEconstraintsLst) ⇒
      maximalpolicy_Order
         (listTErule_EvalTE
                         CompOne_TErulesLst q)
         (listTEconstraint_EvalTE
                  CompTwo_TEconstraintsLst q CompOne_TErulesLst)
 end.
```

Listing 4.25: Evaluating a policy TEpolicy_EvalTE

In function TEpolicy_EvalTE, the first and second components of policies are evaluated respectively by the function listTErule_EvalTE and listTEconstraint_EvalTE. The function maximalpolicy_Order combines the results of these evaluations as described above.

4.3 Conditions on Predicates

As mentioned earlier in Section 3.2.7, policy writers should make sure the predicates that they develop (as the last argument of a `TEconstraint`) satisfy certain conditions. Note that this checking is performed statically for predicates, by proving the required properties in Coq before using them in policies, and there is no dynamic checking needed. Accordingly, policy writers have to verify that their definitions of predicates satisfy the three conditions `predicate_query_condition` (in listing 4.27), `Predicate_plc_cdn`, and `Predicate_plc_cdn_Transition` (in listing 4.31), which require some definitions which we explain first. These three conditions express that given two queries related by "<<=" or two policies related by "$\lesssim$", the evaluation of predicates by the function `TEconstraint_EvalTE` preserves the defined order on decisions "<::".

TEpla policies and queries affect the evaluation of predicates in `TEconstraint_EvalTE` since predicates receive a list of elements of type `TErule`, which comes from the first component of policies (`TEpolicy`), and two input arguments of datatype `type`, which come from the source and destination `types` of queries (see listing 4.22). A predicate must be defined so that whatever the policy is, the results of evaluating the predicate in the definition of `TEconstraint_EvalTE` on two policies and queries that satisfy the order relations "$\lesssim$" and "<<=", respectively, are two decisions that respect the ordering on decisions "<::" defined in listing 3.2. TEpla predicates can produce `false` and `true` decisions, which are mapped to `UnKnown` and `Permitted` respectively by `TEconstraint_EvalTE`. Taking into account this mapping as well as the ordering relation on the decisions defined in Section 3.4.1, in listing 4.26 we define a binary relation as the boolean characterization of the relation between decisions of predicates (i.e., `true` and `false`).

The boolean relation `boolChrs_transition_dcs` takes two decisions returned by a predicate and verifies that they respect the ordering on decisions. The second case of pattern-matching in `boolChrs_transition_dcs`, for example, happens when the first result of applying the predicate is `false` and the second is `true`. According to the function `TEconstraint_EvalTE` in which predicates are evaluated, these results correspond to `UnKnown` and `Permitted` decisions, which does not satisfy the relation satisfy the relation on decisions (i.e., "<::"), making the output of the `boolChrs_transition_dcs` for this case equal to `false`.

```
Definition boolChrs_transition_dcs (boolF boolS: bool) : bool :=
 match boolF,boolS with
  | _, false⇒  true
  |false, true ⇒ false
  | true,true ⇒ true
  end.

Notation "res1 <transition_Verify_Decision> res2" :=
          (boolChrs_transition_dcs res1 res2) (at level 70).
```

Listing 4.26: The boolean relation `boolChrs_transition_dcs`

We first consider applying `TEconstraint_EvalTE` with two queries `qry` and `qry'` such that (`qry <<= qry'`), and express the required condition for this case (See `predicate_query_condition` and its use of `boolChrs_transition_dcs` in listing 4.27). The source of these two queries are `qry_source_type` and `qry'_source'_type`, and as appears in the code, it must be the case that (`typeSubset qry'_source'_type qry_source_type`), and similarly for the destination components. No other arguments are compared in this particular predicate.

Therefore, we can reflect the relation between the source and destination `type` of queries as the premise of `predicate_query_condition`. By having the proper premise, `predicate_query_condition` states that the changes in the boolean results of predicates for queries of the relation "<<=" must be according to the order of boolean values defined in `<transition_verify_Decision>`. As a result, `predicate_query_condition` is the first condition that predicates must satisfy.

```
Definition predicate_query_condition
(predicate_from_constraint: list TErule → list type →
    class → permi → type → type → type → type → bool ):=

∀ (list_TErule: list TErule) (listT:list type)(qry'_source'_type
qry'_destination'_type qry_source_type qry_destination_type
typeFst typeSnd : type)(cla:class)(perm:permi),

(typeSubset qry'_source'_type qry_source_type) &&
(typeSubset qry'_destination'_type qry_destination_type)= true →

 ((predicate_from_constraint list_TErule listT cla perm qry_source_type
    qry_destination_type typeFst typeSnd)
                          <transition_Verify_Decision>
 (predicate_from_constraint list_TErule listT cla perm
   qry'_source'_type qry'_destination'_type typeFst typeSnd)) = true.
```

Listing 4.27: Defining the condition of predicates for queries

In listing 4.28, in the lemma `predicate_query_condition_implication`, we prove that constraints whose predicates satisfy `predicate_query_condition`, result in decisions with the relation "<::".

```
Lemma predicate_query_condition_implication :
∀ (qry  qry':query )(cnstnt: TEconstraint) (IT: list TErule),
    predicate_query_condition (predicate_in_the_constraint cnstnt) →
    qry <<= qry' →

((TEconstraint_EvalTE cnstnt qry IT) <::
                (TEconstraint_EvalTE cnstnt qry' IT)) = true.
```

Listing 4.28: Lemma `Predicate_query_condition_implication`

In listing 4.28, we use the function `predicate_in_the_constraint` defined in listing 4.29 to extract the last component of a constraint, which is the predicate.

```
Definition predicate_in_the_constraint (ctrt : TEconstraint) :=
 match ctrt with
  |Constraint (c_class,c_permi,c_type1,c_type2,
                listType,c_predicatFunc) ⇒ c_predicatFunc
 end.
```

Listing 4.29: Function `predicate_in_the_constraint`

We use the lemma `predicate_query_condition_implication` in the proofs of many other lemmas and theorems. To simplify using this lemma in proofs, we define `constraints_implication_prd` by rephrasing the statement of the lemma as a definition in listing 4.30. The function `const_imp_prd_List`, the second function in listing 4.30, recursively checks `constraints_implication_prd` on all the constraints in the input list of `TEconstraint`. That is, by applying this function to a list of constraints, we can ensure that all constraints of the list include predicates that all satisfy the condition expressed as `predicate_query_condition`.

```
Definition constraints_implication_prd
(TEconstr: TEconstraint) :=
∀ (qry  qry':query ) (IT: list TErule),

     (qry <<= qry' →
        (TEconstraint_EvalTE TEconstr qry IT) <::
               (TEconstraint_EvalTE TEconstr qry' IT) = true).

Fixpoint const_imp_prd_List
  (lstConstraint : list TEconstraint) : Prop :=
   match lstConstraint with
       |[|] ⇒  True
      |hd::tl⇒ constraints_implication_prd hd ∧ const_imp_prd_List tl
    end.
```

Listing 4.30: Defining `constraints_implication_prd` and `const_imp_prd_List`

As mentioned earlier, in addition to queries, the first components of policies affect the result of the evaluation of predicates through their first input argument of type list of `TErule` (see listing 4.22).

We next consider applying `TEconstraint_EvalTE` with two different sets of rules in two policies. We define `Predicate_plc_cdn` and `Predicate_plc_cdn_transition`

in listing 4.31 as the conditions on predicates about these two rule sets. This is an example where we express the $\lesssim$ relation implicitly. For example, note that (`list_TErule1` $\lesssim$ `list_TErule2 ++ list_TErule1`)

The condition `Predicate_plc_cdn` states that the result of a predicate with the concatenation of two lists of `TErules` as its first input argument is equal to the result of the predicate when we change the order of the two lists, i.e., changing the order of lists does not change the result of the predicate.

Accordingly, the condition `Predicate_plc_cdn_Transition` states that when the second argument is the concatenation of a list of `TErules` to the first input argument to a predicate (i.e., initial list of `TErule` named `list_TErule1` in the listing 4.31), the results of evaluating the predicate are according to `<boolChrs_transition_dcs>`.

```
Definition Predicate_plc_cdn
  (predicate_from_constraint: list TErule → list type →
     class → permi → type → type → type → type → bool ):=
  ∀ (listT:list type)
  (list_TErule1 list_TErule2: list TErule)(typeA typeB typeC typeD : type) (
      cla : class)(perm : permi),

 ((predicate_from_constraint (list_TErule1 ++list_TErule2) listT cla perm
    typeA typeB typeC typeD) =
 (predicate_from_constraint (list_TErule2 ++list_TErule1) listT cla perm
    typeA typeB typeC typeD)).


Definition Predicate_plc_cdn_Transition
  (predicate_from_constraint: list TErule → list type →
     class → permi → type → type → type → type → bool ):=
  ∀ (listT:list type)
  (list_TErule1 list_TErule2: list TErule)(typeA typeB typeC typeD : type)(
      cla : class)(perm : permi),

 (((predicate_from_constraint) list_TErule1 listT cla perm typeA typeB
    typeC typeD)
                      <transition_Verify_Decision>
 (predicate_from_constraint (list_TErule2 ++list_TErule1) listT cla perm
    typeA typeB typeC typeD)) = true.
```

Listing 4.31: Defining the conditions of predicates for policies

Regarding the two conditions in listing 4.31, `Predicate_plc_cdn_Transition` is defined using the `<transition_Verify_Decision>` relation, which we introduced in

listing 4.26.

The definition cnsrt_prd_plcyRules, the first definition in listing 4.32, is the conjunction "∧" of the conditions listed in listing 4.31. The function cnsrt_prd_plcy-RulesList recursively applies the definition cnsrt_prd_plcyRules to all the constraints in the input argument listConsrt which is a list of TEconstraint. In other words, by applying cnsrt_prd_plcyRulesList to a list of constraints, we state that all the predicates of the included constraints in the list, satisfy the conditions of listing 4.31.

```
Definition cnsrt_prd_plcyRules (predicate_from_constraint: list TErule
   → list type →
    class → permi → type → type → type → type → bool ) :=

     Predicate_plc_cdn predicate_from_constraint ∧
     Predicate_plc_cdn_Transition predicate_from_constraint.

Fixpoint cnsrt_prd_plcyRulesList (listConsrt : list TEconstraint) : Prop :=
  match listConsrt with
     || ⇒ True
     |h::t ⇒ cnsrt_prd_plcyRules (predicate_in_the_constraint h) ∧
           cnsrt_prd_plcyRulesList t
   end.
```

Listing 4.32: Defining cnsrt_prd_plcyRules and cnsrt_prd_plcyRulesList

Here we present an implication of having the conditions stated by cnsrt_prd_-plcyRules. By assuming that predicates satisfy the conditions expressed in cnsrt_prd-_plcyRules, we can prove the lemma constraintEvalPropSnd in listing 4.33. The lemma expresses that if all the constraints of a list of constraints satisfy the conditions of cnsrt_prd_plcyRules, then the result of the evaluation of the list of constraints with a particular list of TErule has the relation <:: with the result of the evaluation of the constraint by adding more TErules (represented by the argument listRuleB) to the list of the initial list of TErules which is represented by the argument listRuleA.

```
Lemma constraintEvalPropSnd:
  ∀ (listRuleA listRuleB: list TErule)
  (listCnstrt: list TEconstraint) (q:query),
   cnsrt_prd_plcyRulesList (listCnstrt) →

    (listTEconstraint_EvalTE listCnstrt q listRuleA <::
     listTEconstraint_EvalTE listCnstrt q (listRuleA++listRuleB)) = true.
```

Listing 4.33: Lemma `constraintEvalPropSnd`

To sum up the conditions on predicates, policy writers have to verify that their defined predicates are compatible with TEpla properties. Therefore, they have to make sure predicates satisfy the three conditions defined in listings 4.27 and 4.31. In the next section, we prove the formal properties of TEpla. Moreover, we prove that the predicate `Operation_predicate_TEpla` defined in listing 4.10, satisfies the three conditions on predicates, i.e., `predicate_query_condition` (in listing 4.27), `Predicate_plc_cdn`, and `Predicate_plc_cdn_Transition` (in listing 4.31). By performing this verification, constraints can thus utilize this predicate without violating the properties of TEpla.

4.3.1 The Expressive Power of Predicates

The expressive power of predicates is limited by the conditions that they have to satisfy, which were discussed in this section. Alternatively, however, we propose two methods to extend the expressive power of predicates. Accordingly, the expressive power of constraints is the same as the expressive power of predicates because they exploit predicates to encode security goals.

The first method is related to the fact that predicates can just change the decisions for queries from `Permitted` to `UnKnown`. There are cases where we may want to allow a change from `UnKnown` to `Permitted`. Currently, predicates cannot change `Permitted` or `UnKnown` decisions to `NotPermitted` decisions. This means that by not verifying predicates, the only possible decision order that is not pursuant to the decision order defined in Section 3.4.1 is `UnKnown` to `Permitted` decisions. Consequently, accommodating this fact enables policy developers to have the same expressive power for encoding security goals in predicates as the studies that use sets to express security goals, such as [Jaeger and Tidswell, 2001], which empirically illustrates that practical binary constraints can be expressed by comparisons of two sets.

The second method considers applying a structural restriction on policies. For instance, assume that all policies of an organization have the same rule component but different constraint components. Such a situation is applicable when each department has

different security goals but has the same set of rules defined by the security administrator. With this change, we will no longer have to check that the conditions on predicates hold. The properties *non-decreasing*, *independent composition*, and *determinism* will still hold in the policies. Therefore, by applying this restriction, we can use predicates without verifying them; however, the property *order preservation* will not hold in this case. Accordingly, applying this limitation depends on the intended properties of policies, which are determined by security administrators of organizations. Note that we do not prove the application of this restriction on formal properties, however, as an example, we provide the proof of *independent composition* after applying this change on policies and removing the conditions from predicates, which is available in the TEpla Coq source code [Eaman, 2019] as Theorem `Independent_Composition_limited`. The expression and proof of this theorem are similar to Theorem `Independent_Composition` in Section 4.4.4, except that we apply the restriction on policies and remove the premise about the conditions of predicates.

4.4 Proving the Main Properties

In this section, we prove the formal properties of TEpla and verify the required conditions of the predicate `Operation_predicate_TEpla`. We start by proving that the predicate satisfies the conditions stated in Section 4.3 (Figs. 4.27 and 4.31). We continue by proving the decidability of decisions. This section then continues by proving the set of properties which are explained in Section 3.4, which require some definitions which we explain first.

4.4.1 Verifying the `Operation_predicate_TEpla`

Predicate definitions have to be verified to ensure that they satisfy the three conditions that were discussed in the previous section. Here, we verify that the predicate `Operation_predicate_TEpla` defined in listing 4.10 satisfies all these conditions. In the first lemma of listing 4.34, `qry_condition_Operatonprd`, we prove that the condition stated in 4.27 is held by this predicate. In addition, the next two lemmas of the listing, i.e., `plc_conditionF_Operatonprd` and `plc_conditionS_Operatonprd`, prove that the predicate satisfies the two conditions stated in listing 4.31, `Predicate_plc_cdn` and `Predicate_plc_cdn_Transition` respectively. Consequently, constraints in policies can use this predicate.

```
Lemma qry_condition_Operatonprd:

          predicate_query_condition Operation_predicate_TEpla.


Lemma plc_conditionF_Operatonprd:

          Predicate_plc_cdn Operation_predicate_TEpla.


Lemma plc_conditionS_Operatonprd:

          Predicate_plc_cdn_Transition Operation_predicate_TEpla.
```

Listing 4.34: Verifying the predicate `Operation_predicate_TEpla`

4.4.2 Decidability of Decisions

In this section, we use proof by reflection to prove the decidability of decisions in TEpla. We defined the boolean relation `compare_decisions` (in listing 2.6) for comparing decisions by considering the partial order of decisions expressed in Section 3.4.1. For exploiting the proof by reflection approach, we need to define a propositional representation of the order of decisions, which is expressed as `ans_compr_Prop` in listing 4.35. As depicted in listing 4.35, we need four constructors to cover all the possible relations of decisions. The constructor `Prp_refl` for constructing the reflexive relation of each decisions, the constructor `Prp_trans` for building the transitivity relation of decisions, and the two constructors `Prp_dnp_dp` and `Prp_dp_duk` for expressing the basic relation between `NotPermitted`, `Permitted` decisions, and `Permitted`, `UnKnown` decisions.

```
Inductive ans_compr_Prop : answer → answer → Prop :=
  | Prp_refl : ∀ d,  ans_compr_Prop d d
  | Prp_trans : ∀ dnp dp duk, ans_compr_Prop dnp p →
                  ans_compr_Prop dp duk → ans_compr_Prop dnp duk
  | Prp_dnp_dp : ans_compr_Prop NotPermitted Permitted
  | Prp_dp_duk : ans_compr_Prop Permitted UnKnown.
```

Listing 4.35: The inductively defined relation `ans_compr_Prop`

We prove the decidability of decisions in TEpla through the boolean reflection approach for partial orders presented by the author of [Azevedo de Amorim, 2016]. The

proof begins by proving that the boolean and propositional representation of the order of decisions are equivalent. That is we now prove that a propositional representation, defined as ans_compr_Prop, of the boolean relation compare_decisions, are logically equivalent (stated as a theorem in listing 4.36).

```
Theorem aPrp_Boolreflection d1 d2 :
    reflect (ans_compr_Prop d1 d2) (compare_decisions d1 d2).
```

Listing 4.36: The reflection relation between ans_compr_Prop and compare_decision

After proving the equivalence of between the two characterizations of the order of decisions, we then define the decidability of decisions according to the theorem decidability_of_decisions in listing 4.37.

```
Theorem decidability_of_decisions dcsF dcsS :
        {ans_compr_Prop dcsF dcsS} + {  ans_compr_Prop dcsF dcsS}.
```

Listing 4.37: Decidability of TEpla decisions

The theorem in listing 4.37 indicates that the relation "<::" on decisions is decidable. That is when we compare decisions by the relation "<::", there is always an answer for this comparison.

4.4.3 Proof of Order Preservation

We discuss the proof of *Order Preservation* in some detail in this section. Our encoding of the order preservation property (Theorem (1) in Section 3.4.4) into Coq is through theorem Order_Preservation_TEpla (see listing 4.38). const_imp_prd_List is a predicate to state that all the predicates of the input list of constraints listCnstrt have the condition stated in listing 4.27.

```
Theorem Order_Preservation_TEpla :
  ∀ (listRule:list TErule) (listCnstrt:list TEconstraint)
  (q q' : query),

     (q <<= q') ∧ const_imp_prd_List listCnstrt →

    ((TEpolicy_EvalTE (TEPolicy (listRule, listCnstrt)) q) <::
    (TEpolicy_EvalTE (TEPolicy (listRule, listCnstrt)) q')) = true.
```

Listing 4.38: Proving *Order Preservation*

The proof of theorem `Order_Preservation_TEpla` begins by induction on `listCnstrt`. This induction breaks the goal into two subgoals, subgoal `I`, and subgoal `II` depicted in listing 4.39. (See the complete proof script of this lemma in listing 4.54.)

```
(* subgoal I*)
∀ q q' : query,
q <<= q' ∧ const_imp_prd_List [ ] →

(TEpolicy_EvalTE (TEPolicy (listRule, [ ])) q <::
 TEpolicy_EvalTE (TEPolicy (listRule, [ ])) q') = true

(* subgoal II*)
∀ q q' : query,
q <<= q' ∧ const_imp_prd_List (a :: listCnstrt) →

(TEpolicy_EvalTE
   (TEPolicy (listRule, a :: listCnstrt)) q <::
 TEpolicy_EvalTE
   (TEPolicy (listRule, a :: listCnstrt)) q') = true
```

Listing 4.39: The base and inductive case of induction on `listCnstrt`

The first subgoal (subgoal `I`) is the *base case* of the induction. In the base case, we must show that the theorem holds for `listCnstrt` when it is equal to the empty list [ ], i.e., `listCnstrt` is replaced by the empty list. By applying the tactics `intros` and `simpl`, subgoal `I` changes to the following goal:

```
(maximalpolicy_Order
     (listTErule_EvalTE listRule q) NotPermitted <::
 maximalpolicy_Order
     (listTErule_EvalTE listRule q') NotPermitted) = true
```

To continue the proof, we define the lemma `maximalpolicy_Prop` as defined in listing 4.40.

```
Lemma maximalpolicy_Prop:
  ∀ (a b c d: answer),
      (a <:: c)=true ∧ (b <:: d)=true →
         (maximalpolicy_Order a b <:: maximalpolicy_Order c d)=true.
```

Listing 4.40: Lemma `maximalpolicy_Prop`

We use the tactic `apply maximalpolicy_Prop` in solving this new goal, which modifies the goal to the following goal:

```
 (listTErule_EvalTE listRule q <::
                     listTErule_EvalTE listRule q') = true
            ∧
 (NotPermitted <:: NotPermitted) = true
```

The proof of this goal is completed by applying some tactics and the lemma `max_prop2` defined in listing 4.41.

```
Lemma max_prop2:
 ∀ (lt :list TErule)(q q' :query),
   q <<= q' →
   (listTErule_EvalTE lt q) <:: (listTErule_EvalTE lt q')=true.
```

Listing 4.41: Lemma `max_prop2`

The other subgoal (i.e., subgoal `II` in listing 4.39) is the *inductive case* of the induction. In the inductive case, we have to show that, assuming the theorem holds for `listCnstrt`, it holds for `a::listCnstrt` where `a` is a single constraint. The assumption is added to the context as the induction hypothesis, shown in listing 4.42.

```
∀ q q': query,
 q <<= q' ∧ const_imp_prd_List listCnstrt →

     (TEpolicy_EvalTE (TEPolicy (listRule, listCnstrt)) q <::
        TEpolicy_EvalTE (TEPolicy (listRule, listCnstrt)) q') = true
```

Listing 4.42: The induction hypothesis of induction on `listCnstrt`

We continue the proof of the inductive case by applying different tactics and using some lemmas, such as lemma `OrdPreserv_cstrt` in listing 4.43.

```
Lemma OrdPreserv_cstrt:
 ∀ (lstRule : list TErule)(cnstrt: TEconstraint)(q q' :query),
  q<<=q' ∧ constraints_implication_prd (cnstrt) →
   (TEconstraint_EvalTE (cnstrt) q lstRule <::
           TEconstraint_EvalTE (cnstrt) q' lstRule) = true.
```

Listing 4.43: Lemma `OrdPreserv_cstrt`

Lemma `OrdPreserv_cstrt` states that, by assuming that two queries `q` and `q'` are related by `<<=` and the constraint `cnstrt` satisfies the condition expressed by `constraints_implication_prd`, the relation `<::` holds between the results of the evaluation of `TEconstraint_EvalTE` on the two queries `q` and `q'` with the same input arguments `cnstrt` and `lstRule`.

4.4.4 Proofs of Remaining Properties

Proving *Independent Composition* (expressed in the Theorem 3 of Section 3.4.6) is more elaborate than the proof of *Order Preservation* because we need two sequences of finite size, both defined as inductive datatypes: a list of policies and a list of decisions. Suppose that p_i is the element at index i in the list of policies and d_i is the element at index i in the list of decisions. Then these two lists are pairwise related by the following relation (we call the tuple (p_i, d_i) an *evaluation pair*):

$$(\texttt{TEpolicy\_EvalTE}\ (p_i)\ ) = d_i$$

We encode *Independent Composition* using lists of evaluation pairs. Theorem `Independent_Composition` (in listing 4.44) states the *Independent Composition* of TEpla. In this theorem, `poList_answList` is the list of evaluation pairs.

```
Theorem Independent_Composition:
   ∀ (poList_answList : list (TEpolicy * answer)) (q : query) (dstar : answer),

 Foreach q
      (List.map (@fst TEpolicy answer) poList_answList)
      (List.map (@snd TEpolicy answer) poList_answList)
                                        ∧
        (TEpolicy_EvalTE
          (poList_Serialization
           (List.map
           (@fst TEpolicy answer) poList_answList)) q) = dstar →
              (maximum
                 (List.map(@snd TEpolicy answer) poList_answList) <:: dstar)
                    = true.
```

Listing 4.44: Theorem `Independent_Composition`

The definition of `Independent_Composition` uses the function `maximum` defined in listing 4.45. The function `maximum` is a recursive function for finding the maximum of a list of elements of decisions by applying two decisions at each step to `maximalDcs`, in listing 4.20.

```
Fixpoint maximum lstAns : answer :=
   match lstAns with
    |[]  ⇒ NotPermitted
    |( x :: xs) ⇒ maximalDcs x (maximum xs)
   end.
```

Listing 4.45: The function `maximum`

The projection functions: `@fst` and `@snd`, from the Coq library `Coq.Init.list`, extract two separate elements from a pair. The higher-order function `map`, which is defined (listing 4.46) in the Coq library `Coq.Init.list`, receives a function, and applies the function to each element of a list and returns a new list with the results of the function. The use of curly brackets in this definition indicates to Coq that the first two arguments to `map` are types which can be omitted because Coq can infer them automatically. Thus, in the definition of theorem `Independent_Composition`, we use the function `map` to apply the `@fst` and `@snd` functions to all the elements of the input list of evaluation pairs, `poList_answList`, in order to obtain two separate lists of policies and decisions.

```
Fixpoint map {A B: Type} (f: A → B) (l: list A): (list B) :=
    match l with
      | [ ] ⇒ [ ]
      | h :: t ⇒ (f h) :: (map f t)
    end.
```

Listing 4.46: The higher-order function `map`

The universal predicate `Foreach`, in listing 4.47, extracts evaluation pairs from these lists. In addition to expressing the evaluation pair relation, `Foreach` states that all the constraints in the second components of polices (i.e., constraint component) satisfy the conditions stated in listing 4.32. In particular, the lists must be the same length and the elements at each position in the lists form an evaluation pair. The base case of `Foreach` is the constructor `foreach_single` which applies to lists of length one (i.e., lists forming one evaluation pair). The other constructor of the predicate `Foreach`, `foreach_cons`, is the inductive case for lists with more than one element.

```
Inductive Foreach:
  query → list TEpolicy → list answer → Prop :=
 |foreach_single : ∀ (ply:TEpolicy) (q:query) (ansr:answer),
         (TEpolicy_EvalTE (ply) q) = ansr ∧ validConstrt (ply) →
         Foreach q [ply] [ansr]


 |foreach_cons : ∀ (ply:TEpolicy) (q:query) (ansr:answer)
           (lstTEPolicy:list TEpolicy) (listAnsw:list answer),

            (TEpolicy_EvalTE (ply) q) = ansr ∧ validConstrt (ply) →
            Foreach q lstTEPolicy listAnsw →
                Foreach q (ply::lstTEPolicy) (ansr::listAnsw).
```

Listing 4.47: The universal predicate `Foreach` for expressing evaluation pairs in lists

The definition of `Foreach` uses `validConstrt` to state that all the constraints in the second components of polices satisfy the condition of `cnsrt_prd_plcyRulesList` (see the listing 4.32).

```
Definition validConstrt (policy: TEpolicy) :=
 match x with
    |TEPolicy (r,cstrt) ⇒ cnsrt_prd_plcyRulesList cstrt
 end.
```

Listing 4.48: Defining `validConstrt`

In theorem `Independent_Composition`, because `TEpolicy_EvalTE` takes a policy (a list of rules and constraints), we need to combine all the policies in the list of policies into a single policy by extracting all the rules and combining them into one list, and similarly for constraints. Therefore, we define the recursive function `poList_Serialization`, in listing 4.49.

```
(* converting list of TEpolicies to a single TEpolicy *)
Fixpoint poList_Serialization (pl : list TEpolicy) : TEpolicy :=
 match pl with
  | [] ⇒ TEPolicy ([],[])
  | (TEPolicy(TEr,TEc)):: tl ⇒ let '(TEPolicy (lsd,rhs)) :=
     poList_Serialization tl in (TEPolicy (TEr++lsd, TEc++rhs))
 end.
```

Listing 4.49: The function `poList_Serialization`

In each recursive step, we concatenate one policy of the input list of policies to the current list of rules and constraints using the infix operator "++" which appends two lists. One of the trivial properties of `poList_Serialization` is that for any policy of type `TEpolicy` we have:

```
poList_Serialization ([policy]) = policy.
```

The proof of the theorem `Independent_Composition` begins by induction on `poList_answList`. We continue the proof by performing case analysis on `answer` terms, through the tactic `destruct`, and solving the subgoals by applying some other tactics or lemmas, such as `Foreach_propmax`, listing 4.50. The lemma `Foreach_propmax` expresses that the evaluation of the serialization of the policies in a list of evaluation pairs is greater than (in terms of the order relation of decisions <::) the result of the application of function `maximum` to the decisions of the list of evaluation pairs.

```
Lemma Foreach_propmax:
 ∀ (t:TEpolicy) (lstpairs: list (TEpolicy * answer)) (q:query) ,

 Foreach q (map fst lstpairs) (map snd lstpairs) →
 (maximum (map snd lstpairs) <::
    (TEpolicy_EvalTE (poList_Serialization (map fst lstpairs)) q)) = true.
```

Listing 4.50: Lemma `Foreach_propmax`

We state and prove the *Non-Decreasing* property of TEpla expressed as Theorem 2 of Section 3.4.5. Listing 4.51 shows the theorem `Non_Decreasing_TEpla`. In consequence of this theorem, TEpla is *Non-Decreasing*.

The theorem `Non_Decreasing_TEpla` states that adding a policy, such as `Single_pol`, to any list of policies, such as `Pol_list`, can change the decisions according to the order relation of decisions we defined in Section 3.4.1. Theorem `Non_Decreasing_TEpla` uses the function `validCnstrtListPolicy` in listing 4.52 to recursively apply `validConstrt` to all elements of the input parameter `Pol_list` which is a list of `TEpolicy`. Note that in this theorem, (`poList_serialization (Pol_list)` $\lesssim$ `poList_serialization (Single_pol::Pol_list)`).

```
Theorem Non_Decreasing_TEpla:
 (∀ (Pol_list: list TEpolicy) (Single_pol:TEpolicy) (q:query)
  (d dprime: answer),
 validCnstrtListPolicy Pol_list ∧ validConstrt Single_pol →

    (TEpolicy_EvalTE
           (poList_Serialization (Pol_list)) q) = d →
    (TEpolicy_EvalTE
       (poList_Serialization (Single_pol::Pol_list))q) = dprime →
                          (d <:: dprime)=true)
```

Listing 4.51: Theorem `Non_Decreasing_TEpla`

```
Fixpoint validCnstrtListPolicy (listPolicy : list TEpolicy) : Prop :=
  match listPolicy with
     |[]⇒  True
     |h::t⇒  validConstrt h ∧ validCnstrtListPolicy t
   end.
```

Listing 4.52: The function `validCnstrtListPolicy`

Determinism, expressed in the Theorem 4 of Section 3.4.7, is the last property we prove for TEpla. TEpla is *deterministic* because the translation functions that make up the semantics of TEpla, such as `TErule_EvalTE`, `TEconstraint_EvalTE`, and `TEpolicy_EvalTE`, always return the same decisions for the same policies and queries. This is because they are defined as functions in Coq. Theorem `Deterministic` in listing 4.53 states the *determinism* of TEpla.

```
Theorem Deterministic:
 ∀ (p: TEpolicy) (q: query)(d dprime : answer),
            (TEpolicy_EvalTE p q) = d →
                (TEpolicy_EvalTE p q) = dprime →
                                d = dprime.
```

Listing 4.53: Theorem `Deterministic`

4.5 Proof Script

As an example of a proof script of a theorem, the proof script for the theorem `Order_Preservation_TEpla`, explained in Section 4.4.3, is shown in listing 4.54. In Coq, we can use symbols including "−", "+", and "⋆" to mark the cases of proofs that correspond to each subgoal on different levels. These symbols are named *bullet*s and are optional, making a proof more readable. The part of the proof that comes after a bullet is the sub-proof for a subgoal. Note that this proof uses many previously proved lemmas. The lemmas `maximalpolicy_Prop`, `max_prop2`, and `OrdPreserv_cstrt`, respectively, are in Figs. 4.40, 4.41, and 4.43. The other lemmas used in this proof including `validconst_prop`, `maximalDcs_prop`, and `max_appendAns4` are shown in Figs. 4.55, 4.56 and 4.57, respectively.

```
Lemma Order_Preservation_TEpla :
 ∀ (listRule : list TErule)(listCnstrt:list TEconstraint)
 (q q' :query),
  q <<= q' ∧ const_imp_prd_List (listCnstrt) →
     ((TEpolicy_EvalTE (TEPolicy (listRule, listCnstrt)) q) <::
         (TEpolicy_EvalTE (TEPolicy (listRule, listCnstrt)) q')) = true.
Proof.
induction listCnstrt.
- intros. simpl.          (* proving subgoal I *)
  elim H. intros.
  apply maximalpolicy_Prop.
  split.    + apply max_prop2. assumption.
            + simpl. reflexivity.
- intros. rewrite max_appendAns4.      (* proving subgoal II *)
 assert(TEpolicy_EvalTE (TEPolicy (listRule, a :: listCnstrt)) q' =
        (maximalDcs (TEpolicy_EvalTE (TEPolicy (listRule, [a])) q')
        (TEpolicy_EvalTE (TEPolicy (listRule, listCnstrt)) q') )).
 rewrite max_appendAns4. reflexivity.
 rewrite H0.
 apply maximalDcs_prop.
 split. + simpl. apply maximalpolicy_Prop. split.
 apply max_prop2. elim H. intros. assumption.
 apply maximalDcs_prop. split.
 apply OrdPreserv_cstrt.
 split. elim H. intros. assumption.
 elim H. intros. apply validconst_prop in H2.
 elim H2. intros. assumption.
 simpl. reflexivity.
      + apply IHlistCnstrt. split. elim H. intros. assumption.
        elim H. intros. apply validconst_prop in H2.
       elim H2. intros. assumption.
Qed.
```

Listing 4.54: The proof script of Theorem `Order_Preservation_TEpla`

```
Lemma validconst_prop:
 ∀ (c : TEconstraint)(l:list TEconstraint),
  const_imp_prd_List (c::l) →
       constraints_implication_prd c ∧ const_imp_prd_List l.
```

Listing 4.55: Lemma `validconst_prop`

```
Lemma maximalDcs_prop:
 ∀ (d1 d2 d3 d4: answer),
    (d1<::d3)=true ∧ (d2<::d4)=true →
         (maximalDcs d1 d2 <:: maximalDcs d3 d4)=true.
```

Listing 4.56: Lemma `maximalDcs_prop`

```
Lemma max_appendAns4:
 ∀ (a:list TErule) (h:TEconstraint)(l:list TEconstraint)
 (q:query),
 (TEpolicy_EvalTE (TEPolicy (a, h :: l)) q) =
      maximalDcs
           (TEpolicy_EvalTE (TEPolicy (a, [h])) q)
           (TEpolicy_EvalTE (TEPolicy (a, l)) q).
```

Listing 4.57: Lemma `max_appendAns4`

4.6 Summary

This chapter described the use of the Coq proof assistant to implement TEpla and carry out all proofs. We started by introducing the main elements of TEpla, i.e., syntax and semantics. The requirements of predicates were presented by expressing three conditions. In addition, a sample predicate was developed and verified, showing that it satisfies the three conditions. Finally, in the last section of the chapter, we proved the formal properties of TEpla. In the next chapter, we develop two more example predicates and use them in writing a sample TEpla security policy.

Chapter 5

Case Study: Developing a Security Policy

In this chapter, we develop an example policy. To this end, we develop two predicates and use them for encoding desired security goals as part of the constraints of the example policy. Accordingly, because predicates can exploit the access information of the first component of policies (the rules), we define functions to extract the access information of the entities of the system based on specific criteria. In addition, we represent unary and binary operators, which policy developers can utilize in defining predicates.

5.1 Overview

One of the strengths of TEpla is its capability for defining various predicates, provided that they satisfy the three conditions defined in Section 4.3. For developing an example policy, we use this capability to encode two security goals.

Recall that predicates receive the first component of policies as their first argument, which is in the form of a list of `TErule`, specifying all the access information of the entities of the system. As mentioned before, we have a set interpretation of policies, which helps to provide an easy way to express conditions. Consequently, by forming various sets of entities according to their access specification, we can apply basic set comparison operators on these sets to encode security goals.

Similar to the approach of [Jaeger and Tidswell, 2001] for expressing security goals, we introduce *Selector function*s, which retrieve various access information out of the first component of policies, and *Operator function*s, which apply certain operations on the results of selector functions or any given arguments of the appropriate types. These functions can be used in defining predicates that use particular access information of entities in their encoding of security goals.

In the next section, we develop a number of selector and operator functions. Using these functions, we define two predicates in Section 5.3. In Section 5.4, we develop an example policy with some constraints that apply the security goals encoded in the predicates.

5.2 Functions for Processing Access Information

In this section, in addition to defining some selector and operator functions, we present some formal properties of these functions.

5.2.1 Selector Functions

In this section we introduce three selector functions `authoschemeSearch_findsubject`, `authoschemeSearch_finddestination`, and `authoschemeSearch_findsubjectGeneral`, defined in listings 5.1, 5.3, and 5.4, respectively. All of these selector functions receive an input argument of type list of elements of `TErule` because primarily they act on the first component of policies. The return type of the selector functions is a list of elements of `type` which can be considered as a set. As mentioned above, we can apply operator functions on the returned set of selector functions.

The selector function `authoschemeSearch_findsubject`, defined in listing 5.1, searches all the `Allow` rules of the input argument list of elements of type `TErule`, which is represented by the argument `auscheme`, to find all the source `types` which can access the input argument `dType` to perform the action expressed by the input argument `sdaction`. In other words, the output of `authoschemeSearch_findsubject` is a list of elements of `type` that are authorized to access `dType` to perform the action `sdaction`.

```
Fixpoint authoschemeSearch_findsubject
 (auscheme:list TErule) (dType:type) (sdaction:permi): list type :=

 match auscheme with
 |Allow (src,dst,cs,pr,bl) :: bodyauscheme ⇒
   if (type_Equal dst dType && Nat.eqb pr sdaction) then
      src :: authoschemeSearch_findsubject bodyauscheme dType sdaction
             else
            authoschemeSearch_findsubject bodyauscheme dType sdaction

 |Type_Transition (src,dst,pr)::bodyauscheme ⇒
         authoschemeSearch_findsubject bodyauscheme dType sdaction
  |[]  ⇒ []
end.
```

Listing 5.1: The selector function `authoschemeSearch_findsubject`

The selector function `authoschemeSearch_findsubject` uses the function `type_Equal` to check whether two `types` are equal. The function `type_Equal` is defined in listing 5.2 and checks that the two input arguments of `type` are equal by verifying that they are subsets of each other, i.e., related by `typeSubset` (expressed in listing 4.16).

```
Definition type_Equal (fType sType:type) :=
    typeSubset fType sType && typeSubset sType fType.
```

Listing 5.2: The function `type_Equal`

The second selector function is `authoschemeSearch_finddestination` defined in listing 5.3 and is the same as the previous selector except that it searches for a source `type` instead of a destination `type`. It finds all the destination `types` which are accessed by the input argument `sType` for performing the action expressed by the input argument `sdaction`.

```
Fixpoint authoschemeSearch_finddestination
    (auscheme:list TErule) (sType:type) (sdaction:permi): list type :=

match auscheme with
 |Allow (sc,ds,cs,pm,bl)::bodyauscheme ⇒
     if (type_Equal sc sType && Nat.eqb pm sdaction) then
      ds :: authoschemeSearch_finddestination bodyauscheme sType sdaction
                    else
       authoschemeSearch_finddestination bodyauscheme sType sdaction

 |Type_Transition (sc,ds,pm)::bodyauscheme ⇒
        authoschemeSearch_finddestination bodyauscheme sType sdaction
 |[] ⇒  []
 end.
```

Listing 5.3: The selector function `authoschemeSearch_finddestination`

The third selector function `authoschemeSearch_findsubjectGeneral`, defined in listing 5.4, searches all the `Allow` rules of `auscheme` to find all the source `types` which can access the input argument `dType` to perform any kind of action. In other words, the output of `authoschemeSearch_findsubjectGeneral` is a list of elements of `type` that are authorized to access `dType` no matter what action they perform.

```
Fixpoint authoschemeSearch_findsubjectGeneral
       (auscheme:list TErule) (dType:type): list type :=
match auscheme with
|Allow (sc,ds,cl,pm,bl)::bodyauscheme ⇒
     if (type_Equal ds dType) then
       sc :: authoschemeSearch_findsubjectGeneral bodyauscheme dType
       else
           authoschemeSearch_findsubjectGeneral bodyauscheme dType

|Type_Transition (sc,ds,pm)::bodyauscheme ⇒
            authoschemeSearch_findsubjectGeneral bodyauscheme dType
|[]  ⇒ []
end.
```

Listing 5.4: The selector function `authoschemeSearch_findsubjectGeneral`

Listing 5.5 expresses the distributive property of the selector functions defined in this section. In particular, the results of applying selector functions to an input that is the

concatenation of two lists of elements of TErule are the same as the concatenation of the results of the application of the selector functions to each sub-list.

```
Lemma dstrb_destination (l1 l2:list TErule) (ta:type) (p1:permi):
 authoschemeSearch_finddestination (l2 ++l1) ta p1 =
       authoschemeSearch_finddestination (l2) ta p1 ++
        authoschemeSearch_finddestination (l1) ta p1.


Lemma dstrb_sbj (l1 l2:list TErule) (ta:type) (p1:permi):
 authoschemeSearch_findsubject (l2 ++l1) ta p1 =
        authoschemeSearch_findsubject (l2) ta p1 ++
        authoschemeSearch_findsubject (l1) ta p1.


Lemma dstrb_sbjGnrl (l1 l2:list TErule) (ta:type):
 authoschemeSearch_findsubjectGeneral (l2 ++l1) ta =
        authoschemeSearch_findsubjectGeneral (l2) ta ++
        authoschemeSearch_findsubjectGeneral (l1) ta.
```

Listing 5.5: The distributive property of the selector functions

5.2.2 Operator Functions

The operator functions we introduce include IntersectionList in listing 5.6, which returns the set of common elements of two lists of elements of type, is_emptylisttype in listing 5.8, which checks whether or not a list of elements of type is empty, and list_subSet in listing 5.10, which checks whether or not one list of elements of type is a subset of another one.

The first operator function we develop is IntersectionList. This operator function takes two arguments of list of elements of type represented by lstFirst, and lstSecond. The operator function IntersectionList returns all the common elements in the two input lists lstFirst, and lstSecond.

```
Fixpoint IntersectionList (lstFirst lstSecond: list type) : list type:=
 match lstFirst with
     |[]  ⇒ []
     | (a1) :: l' ⇒
           match lstSecond with
             |[]  ⇒ []
             | l ⇒ if (∃ b (type_Equal a1) lstSecond)
                        then (a1) :: IntersectionList l' lstSecond
                        else IntersectionList l' lstSecond
              end
 end.
```

Listing 5.6: The operator function `IntersectionList`

The operator function `IntersectionList` uses `existsb`, defined in the Coq library `Coq.Lists.List` (depicted in listing 5.7), to check whether or not an element of the first input list `lstF` is `type_Equal` to any elements of the input list `lstS`.

```
Fixpoint ∃ b (l:list A) : bool :=
     match l with
       | nil ⇒ false
       | a::l ⇒ f a || ∃ b l
     end.
```

Listing 5.7: The function `existsb`

The operator function `is_emptylisttype` defined in listing 5.8 takes one input argument of list of elements of `type` and returns `bool` to indicates whether the input list is empty or not.

```
Definition is_emptylisttype (l : list type) : bool :=
   match l with
     |[]  ⇒ true
     | _ ⇒ false
  end.
```

Listing 5.8: The operator function `is_emptylisttype`

The operator function `list_subSet` receives two input arguments of lists of elements of `type` represented by `lstFirst` and `lstSecond` and indicates whether or not

lstFirst is a subset of lstSecond. The function list_subSet exploits pattern matching and existsb to check if all the elements of the first input argument lstFirst have the relation typePredicate_listSubset, defined in listing 5.10, to any elements of the input argument lstSecond.

```
Fixpoint list_subSet (lstFirst lstSecond: list type) : bool:=
 match lstFirst with
       |[]  ⇒ true
       | a1 :: l' ⇒ if (∃ b (typePredicate_listSubset a1) lstSecond)
                       then list_subSet l' lstSecond
                     else false
 end.
```

Listing 5.9: The operator function list_subSet

The function typePredicate_listSubset returns true if the two input arguments of type have the same constructor and equal arguments. Equality of arguments of constructors in the function typePredicate_listSubset is defined such that for basictype arguments (i.e., arguments for the constructor singletype) their nat values are equal and for attribute arguments (i.e., arguments for the constructor grouptype) the elements included in their lists are equal. Recall that we encoded attribute as a list of elements of basictype. Accordingly, checking for equality of two attributes is implemented by "==", which is defined in the Coq library mathcomp.ssreflect.eqtype for checking the equality of two lists. The equality operator "==" checks whether or not the elements in the same place in each list are identical, and returns a boolean.

```
Fixpoint typePredicate_listSubset
                (fL:type) (sL:type) : bool :=
 match fL with
 |singletype st ⇒ match sL with
                    |singletype st2 ⇒ (Nat.eqb (st)(st2))
                    |grouptype gt2 ⇒ false
                  end
 |grouptype gt ⇒ match sL with
                   |singletype st2 ⇒ false
                   |grouptype gt2 ⇒ (gt == gt2)
                  end
 end.
```

Listing 5.10: The function typePredicate_listSubset

Listing 5.11 depicts seven basic properties of the operator functions that we introduced in this section. The lemma `notNil_intrsec`, for example, expresses that if the result of the application of `IntersectionList` to two lists of elements of `type` is not empty then by adding another element to these lists, the result of the intersection is still not empty. As another example, the lemma `sub_inter_trivial_true` states that the intersection of two lists of elements of `type` is a subset (using operator `list_subSet`) of the intersection of the same two lists after concatenating some elements to each of the lists.

```
Lemma intserc_distrib (l1 l2 ta: list type):
     IntersectionList (l2 ++l1) ta =
          IntersectionList l2 ta ++IntersectionList l1 ta.


Lemma notNil_intrsec (l1 l2 l3 l4: list type):
       IntersectionList (l1) (l2) <> [] →
              IntersectionList (l3++l1) (l4++l2) <> [].


Lemma isempty_intrsct (lst1 lst2 lst3 lst4:list type) :
      is_emptylisttype(IntersectionList (lst1++lst2)(lst3++lst4))=
      is_emptylisttype(IntersectionList (lst2++lst1)(lst4++lst3)).


Lemma isempty_trivial (l1 l2:list type):
        is_emptylisttype (l2 ++l1) =
        (is_emptylisttype l2) && (is_emptylisttype l1).


Lemma listsub_add (l1 l2 l3 :list type):
          (list_subSet l1 l2) = true →
                    (list_subSet l1 (l3++l2)) = true.


Lemma listsub_dist (a b c: list type):
      list_subSet (a++b) c = list_subSet a c && list_subSet b c.


Lemma sub_inter_trivial_true (l1 l2 l3 l4: list type):
     (list_subSet (IntersectionList l1 l2)
                 (IntersectionList (l3 ++l1)(l4++l2)) = true).
```

Listing 5.11: Some properties of the operator functions

5.3 Defining the Predicates

In the previous section, we defined some selector and operator functions. Now we can use these functions to define predicates. This section presents two predicates `Prd_SeparationOfDuty` and `Prd_TrustDomain`, which are respectively defined in listings 5.12 and 5.14.

The predicate `Prd_SeparationofDuty` (defined in listing 5.12) encodes our interpretation of SoD security goals. The predicate states that only distinct subjects can access two particular `types`, which are represented by the input arguments `sourceType`, and `destinType`. The function returns `true` if the results of applying `authoschemeSearch_findsubjectGeneral` on the input arguments `sourceType` and `destinType` with the same list of `TErule`, represented by `list_TErules`, have no common elements. In other words, the function returns `true` if no subject can access both the input arguments `sourceType` and `destinType`.

```
Fixpoint Prd_SeparationOfDuty
    (list_TErules: list TErule)(Listtype:list type)(sClass:class)
    (sPermis:permi) (querySourType:type) (queryDesType:type) (sourceType:
       type) (destinType:type) : bool :=

  if (typeSubset querySourType sourceType &&
      typeSubset queryDesType destinType)
          then

 is_emptylisttype
         (IntersectionList
             (authoschemeSearch_findsubjectGeneral
                              list_TErules sourceType)
             (authoschemeSearch_findsubjectGeneral
                              list_TErules destinType)
   else true.
```

Listing 5.12: The predicate `Prd_SeparationOfDuty`

For checking whether or not the predicate is applicable to a query, the predicate verifies that the source and destination `type` of the query (provided by the arguments `querySourType` and `queryDesType`) are subsets of the input arguments `sourceType`, and `destinType` respectively, i.e., related by `typeSubset`. The predicate returns `true` if this condition is false.

Recall that a predicate must take eight arguments as discussed in Section 3.2.7, but there is, of course, no requirement that the predicate uses them all. In the above predicate,

note that the second, third, and fourth arguments are not relevant for expressing Separation of Duty.

As mentioned in Section 4.3, predicates have to satisfy the three conditions `predicate_query_condition` (in listing 4.27), `Predicate_plc_cdn`, and `Predicate_plc_cdn_Transition` (in listing 4.31). Listing 5.13 expresses that the predicate `Prd_SeparationOfDuty` satisfies these three conditions. Note that the proofs of these lemmas are available in the source code of TEpla [Eaman, 2019].

```
Lemma qry_condition_SoDpredicate:
         predicate_query_condition Prd_SeparationOfDuty.

Lemma plc_conditionS_SoDpredicate:
        Predicate_plc_cdn Prd_SeparationOfDuty.

Lemma plc_conditionF_SoDpredicate:
          Predicate_plc_cdn_Transition Prd_SeparationOfDuty.
```

Listing 5.13: Verifying the predicate `Prd_SeparationOfDuty`

Listing 5.14 defines the predicate `Prd_TrustDomain`. In particular, this predicate considers all of the entities such that both the input argument `sourceType` can read them (perform the action `Read`) and the input argument `destinType` can write to them (perform the action `Write`). This set of entities must be a subset of `whitelist`. Again, the check is done only if the source and destination of the rule match the source and destination of the query.

```
Fixpoint Prd_TrustDomain
    (list_TErules: list TErule) (whitelist: list type)
    (sClass:class) (sPermis:permi) (querySourType:type)
    (queryDesType: type) (sourceType:type) (destinType:type) : bool :=
  if (typeSubset querySourType sourceType &&
      typeSubset queryDesType destinType)
                      then
 list_subSet (IntersectionList
             (authoschemeSearch_finddestination
                       list_TErules sourceType Read)
             (authoschemeSearch_findsubject
                       list_TErules destinType Write)) (whitelist)

   else true.
```

Listing 5.14: The predicate `Prd_TrustDomain`

Listing 5.15 expresses that the predicate `Prd_TrustDomain` satisfies the three conditions of predicates. The proofs of these lemmas are available in the Coq source of TEpla [Eaman, 2019].

```
Lemma qry_condition_EpTpredicate:
             predicate_query_condition Prd_TrustDomain.

Lemma plc_conditionS_EpTpredicate:
             Predicate_plc_cdn Prd_TrustDomain.

Lemma plc_conditionF_EpTpredicate:
              Predicate_plc_cdn_Transition Prd_TrustDomain.
```

Listing 5.15: Verifying the predicate `Prd_TrustDomain`

5.4 An Example Security Policy

Now it is time to write an example security policy. We begin by defining the primary datatypes in listing 5.16. Most of the names in listing 5.16 are taken from a SELinux policy in Fedora 29 [Fedora Linux, 2019]. Note that the access authorized in the rule component of the example policy and the security goals that are encoded in the constraint component of the example policy are related to a sample secure system and thus are not related to a real

system and scenario. That is, we develop sample access requirements without considering a real secure system. The listing includes examples of the syntax classes *cond_bool*, *cls*, *basictype*, *prm*, and *attribute* from Fig. 3.1, which extend the definitions in listings 4.1-4.6.

```
Definition condition_bool : bool :=true.

Definition db_schema :class := 600.
Definition kernel_service: class := 601.
Definition system : class := 602.
Definition x_event : class := 603.
Definition Process : class := 604.

Definition NetworkManager_t : basictype := 300.
Definition chrome_sandbox_tmp_t : basictype := 301.
Definition http_client_packet_t : basictype := 302.
Definition inetd_t :basictype := 303.
Definition iptables_t : basictype := 304.
Definition oracleasmfs_t : basictype := 305.
Definition sysadm_passwd_t : basictype := 306.
Definition firewallguit_t : basictype := 307.
Definition eventlogd_t : basictype := 308.

Definition append : permi := 700.
Definition map_create : permi := 701.
Definition Read : permi := 702.
Definition Write : permi := 703.
Definition module_request : permi := 704.
Definition get_value : permi := 705.
Definition update : permi := 706.
Definition Transition: permi := 708.

Definition proc_type : attribute :=
       (chrome_sandbox_tmp_t) :: (oracleasmfs_t) ::
       (iptables_t):: (sysadm_passwd_t):: [].
Definition sysctl_type : attribute :=
              (NetworkManager_t) :: (iptables_t) :: [].
Definition filesystem_type : attribute :=
 (chrome_sandbox_tmp_t) :: (iptables_t) :: [].
Definition netlabel_peer_type : attribute :=
                   (inetd_t) :: (oracleasmfs_t) ::[].
Definition whitelist : list type :=
      (grouptype proc_type):: (singletype firewallguit_t) :: [].
```

Listing 5.16: Defining the datatypes for the example policy

After defining the primary datatypes of the example policy, we define the TErules of the first component of the example policy as defined in listings 5.17 and 5.18.

```
Definition TErule1 : TErule :=
  Allow (grouptype filesystem_type, grouptype sysctl_type, db_schema,
      append, condition_bool).

Definition TErule2 : TErule :=
  Allow (singletype sysadm_passwd_t, singletype firewallguit_t, db_schema,
      get_value, condition_bool).

Definition TErule3 : TErule :=
  Allow (singletype chrome_sandbox_tmp_t, singletype http_client_packet_t
      , system, Write, condition_bool).

Definition TErule4 : TErule :=
  Allow (singletype chrome_sandbox_tmp_t, singletype iptables_t, system,
      Write, condition_bool).

Definition TErule5 : TErule :=
  Allow (singletype iptables_t, singletype http_client_packet_t, system,
      update, condition_bool).

Definition TErule6 : TErule :=
  Allow (singletype chrome_sandbox_tmp_t, singletype inetd_t, db_schema,
      get_value, condition_bool).

Definition TErule7 : TErule :=
  Allow (singletype oracleasmfs_t, singletype sysadm_passwd_t, system,
      module_request, condition_bool).

Definition TErule8 : TErule :=
  Allow (singletype sysadm_passwd_t, singletype iptables_t, kernel_service
      , append, condition_bool).

Definition TErule9 : TErule :=
  Allow (singletype oracleasmfs_t, singletype NetworkManager_t, system,
      get_value, condition_bool).
```

Listing 5.17: Defining the TErules for the example policy - Part I

```
Definition TErule10 : TErule :=
  Allow (singletype iptables_t, singletype sysadm_passwd_t, db_schema,
      get_value, condition_bool).

Definition TErule11 : TErule :=
  Allow (singletype http_client_packet_t, singletype firewallguit_t,
      db_schema, update, condition_bool).

Definition TErule12 : TErule :=
  Allow (singletype chrome_sandbox_tmp_t, singletype firewallguit_t,
      db_schema, append, condition_bool).

Definition TErule13 : TErule :=
  Allow (singletype inetd_t, singletype firewallguit_t, system, Read,
      condition_bool).

Definition TErule14 : TErule :=
  Allow (singletype firewallguit_t, grouptype proc_type, system, get_value,
      condition_bool).

Definition TErule15: TErule :=
  Type_Transition (singletype chrome_sandbox_tmp_t, singletype inetd_t,
      Process).

Definition TErule16 : TErule :=
  Type_Transition (singletype eventlogd_t, singletype
      chrome_sandbox_tmp_t, Process).

Definition TErule17 : TErule :=
  Allow (singletype inetd_t, singletype iptables_t, db_schema, Read,
      condition_bool).

Definition TErule18 : TErule :=
  Allow (singletype firewallguit_t, singletype http_client_packet_t,
      system, Write, condition_bool).

Definition TErule19 : TErule :=
  Allow (grouptype netlabel_peer_type, singletype NetworkManager_t, system
      , module_request, condition_bool).

 Definition TErule20: TErule :=
  Allow (singletype inetd_t, singletype http_client_packet_t, system,
      update, condition_bool).
```

Listing 5.18: Defining the TErules for the example policy - Part II

Note that for evaluating a particular query, all the `Allow` rules in listings 5.17, and 5.18 are considered because their *cond_bool* components (refer to Section 4.2.2) are equal to `condition_bool` whose value is `true`. That is `true` is the only condition that we use in this example. We could include more complicated conditions that sometimes evaluate to true and sometimes evaluate to false depending on information in the system (the environment). Considering a set of attributes as extra conditions to in authorizing access is a future work (See Chapter 6).

We also define the constraints `TEconstraint` for the second component of the example policy. Listing 5.19 defines five constraints.

```
Definition Constraint_SoDA: TEconstraint :=
    Constraint(kernel_service, append,grouptype proc_type, singletype
       iptables_t,[] ,Prd_SeparationOfDuty).

Definition Constraint_SoDB: TEconstraint :=
    Constraint(db_schema, get_value,singletype sysadm_passwd_t,
       singletype firewallguit_t,[],Prd_SeparationOfDuty).

Definition Constraint_SoDC: TEconstraint :=
    Constraint(db_schema, module_request,singletype sysadm_passwd_t,
       singletype firewallguit_t,[],Prd_SeparationOfDuty).

Definition Constraint_IntegrityTrust : TEconstraint :=
    Constraint(system, update,singletype inetd_t, singletype
       http_client_packet_t, whitelist,Prd_TrustDomain).

Definition Constraint_Operational : TEconstraint :=
    Constraint(system, module_request,grouptype proc_type, grouptype
       netlabel_peer_type, [],Operation_predicate_TEpla).
```

Listing 5.19: Defining the `TEconstraints` for the example policy

The constraints `Constraint_SoDA`, `Constraint_SoDB`, and `Constraint_SoDC` use the predicate `Prd_SeparationOfDuty`, defined in listing 5.12. The constraint `Constraint_SoDA` ensures that the attribute `proc_type` can access the basictype `iptables_t` of the class `kernel_service` to perform the action `append` if they are not accesses by common entities. Similarly, the constraints `Constraint_SoDB`, and `Constraint_SoDC` have the same security goal but on the access of the basictype `sysadm_passwd_t` to the basictype `firewallguit_t` of class `db_schema` for performing the actions `get_value`, and `module_request`, respectively.

The constraint `Constraint_IntegrityTrust` exploits the predicate `Prd_Trust-Domain` defined in listing 5.14. By using this predicates, `Constraint_IntegrityTrust`

ensures that the basictype intd_t can access the basictype http_client_packet_t of the class system to perform the action update if those entities that the basictype inetd_t can perform the action Write and the basictype http_client_packet_t can perform the action Read are a member of the list whitelist.

The constraint Constraint_Operational uses the predicate Operation_predicate_TEpla defined in listing 4.10. The constraint ensures that the source and destination type of accesses to the class system for performing the action module_request are not subsets of, i.e., related by typesubset, the attributes proc_type, and netlabel_peer_type, respectively.

We defined the TErules and TEconstraints that we use in the example policy. As the rule and constraint components of policies are lists of elements, we define lists of elements of TErule and TEconstraint in listing 5.20, represented by listpl_TErule and listpl_TEconstraint, respectively.

```
Definition listpl_TErule : list TErule :=
        TErule1 :: TErule2 :: TErule3 :: TErule4:: TErule5:: TErule6 :: TErule7::
           TErule8 ::
            TErule9 :: TErule10 :: TErule11:: TErule12::TErule13 :: TErule14 ::
                 TErule15:: TErule16::TErule17:: TErule18::TErule19::
                    TErule20:: [].

Definition listpl_TEconstraint : list TEconstraint :=
                     Constraint_SoDA::Constraint_SoDB::Constraint_SoDC::
                         Constraint_IntegrityTrust::
                            Constraint_Operational::[].


Definition TEpla_policy : TEpolicy :=
               TEPolicy (listpl_TErule, listpl_TEconstraint).
```

Listing 5.20: Defining the example policy TEpla_policy and the components

Listing 5.20 defines the example policy TEpla_policy, which has listpl_TErule as its rule component, and listpl_TEconstraint as its constraint component.

In listing 5.21, ten queries are defined. The evaluation of these queries is demonstrated in listing 5.22. In listing 5.21, the results of evaluating the queries by TEpolicy_EvalTE (defined in listing 4.25) are depicted as comments in front of each evaluation.

```
Definition TEquery1 : query := (singletype sysadm_passwd_t, singletype
    iptables_t, kernel_service, update).

Definition TEquery2 : query :=(singletype chrome_sandbox_tmp_t, singletype
     iptables_t, db_schema, module_request).

Definition TEquery3 : query := (singletype inetd_t, singletype
    http_client_packet_t, system, update).

Definition TEquery4 : query := (singletype iptables_t, singletype
    firewallguit_t, db_schema, append).

Definition TEquery5 : query := (singletype chrome_sandbox_tmp_t,
    singletype firewallguit_t, db_schema, append).

Definition TEquery6 : query :=(singletype sysadm_passwd_t, singletype
    firewallguit_t, db_schema, get_value).

Definition TEquery7 : query :=(singletype chrome_sandbox_tmp_t, singletype
     inetd_t, Process, Transition).

Definition TEquery8 : query := (singletype inetd_t, singletype
    chrome_sandbox_tmp_t, system, append).

Definition TEquery9 : query := (singletype oracleasmfs_t, singletype
    sysadm_passwd_t, system, module_request).

Definition TEquery10 : query :=(singletype inetd_t, singletype iptables_t,
    db_schema, Read).
```

Listing 5.21: Sample queries on the example policy `TEpla_policy`

```
Eval compute in
    (TEpolicy_EvalTE TEpla_policy TEquery1).(*NotPermitted*)
Eval compute in
    (TEpolicy_EvalTE TEpla_policy TEquery2).(*NotPermitted*)
Eval compute in
    (TEpolicy_EvalTE TEpla_policy TEquery3).(*Permitted*)
Eval compute in
    (TEpolicy_EvalTE TEpla_policy TEquery4).(*NotPermitted*)
Eval compute in
     (TEpolicy_EvalTE TEpla_policy TEquery5).(*Permitted*)
Eval compute in
     (TEpolicy_EvalTE TEpla_policy TEquery6).(*UnKnown*)
Eval compute in
     (TEpolicy_EvalTE TEpla_policy TEquery7).(*Permitted*)
Eval compute in
    (TEpolicy_EvalTE TEpla_policy TEquery8).(*NotPermitted*)
Eval compute in
     (TEpolicy_EvalTE TEpla_policy TEquery9).(*UnKown*)
Eval compute in
     (TEpolicy_EvalTE TEpla_policy TEquery10).(*Permitted*)
```

Listing 5.22: Evaluating the queries of Listing 5.21

Here we explain the evaluation of the queries `TEquery3` and `TEquery6`.

The query `TEquery3` is evaluated against `TEpla_policy` by the Coq command `Eval` in listing 5.22 (the third `Eval` in the figure) that calls `TEpolicy_EvalTE`, defined in listing 4.25, and returns `Permitted` because it is granted by `TErule20` and successfully checked against `Constraint_IntegrityTrust`, which are the only rule and constraint that apply to this query. In particular, first `listTErule_EvalTE`, defined in listing 4.21, is called to evaluate the query with respect to the rules. Every rule in the rule component of the example policy `TEpla_policy` is evaluated by `TEpolicy_EvalTE`, defined in listing 4.19, to check whether or not it is applicable to the query `TEquery3`. Because the source and destination `type` of the query, i.e., the basictypes `inetd_t`, and `http_client_packet_t`, respectively, are equal to the source and destination type of the `TErule20` (i.e., their comparison by the subset relation `typeSubset` is true), as well as the fact that they have the same *cls*, and *prm* parts (i.e., `system`, `update` respectively), this rule is applicable to the query.

Going back to `TEpolicy_EvalTE`, the call to `listTErule_EvalTE` has returned `Permitted`. Next, `listTEconstraint_EvalTE`, defined in listing 4.23, is called to evaluate the query against the constraints of the policy `TEpla_policy`. Every constraint along with the query `TEquery3` is sent to the function `TEconstraint_EvalTE`, defined in listing 4.22, to check whether the constraint is applicable to the query. Because the

class system and the action update of the query TEquery3 are equal to the class and action arguments of the constraint Constraint_IntegrityTrust, this constraint is applicable to the query.

This constraint uses the predicate Prd_TrustDomain defined in listing 5.14. In particular, by considering this predicate and the provided arguments by the constraint Constraint_IntegrityTrust, the predicate evaluates the following condition:

```
list_subSet(
IntersectionList
   (authoschemeSearch_finddestination listpl_TErule (singletype inetd_t)
      Read)
   (authoschemeSearch_findsubject listpl_TErule (singletype
      http_client_packet_t) Write)) whitelist.
```

The evaluation of this condition is true, which leads to the Permitted decision of the second component of TEpla_policy. The condition is correct because in this example the function authoschemeSearch_finddestination returns the list [singletype firewallguit_t; singletype iptables_t], and the function authoscheme-Search_findsubject returns the list [singletype chrome_sandbox_tmp_t;-singletype firewallguit_t]. The intersection (IntersectionList) of these two lists is the list [singletype firewallguit_t] which is a subset (list_subSet) of the list whitelist.

Consequently, evaluating the query TEquery3 by the rule component of the example policy TEpla_policy is Permitted and by the constraint component of the policy is Permitted. As a result, the evaluation of the query TEquery3 by the example policy TEpla_policy is Permitted. This is because calling the function TEplapolicy_Ev-alTE with the arguments TEpla_policy and TEquery3 leads to maximalpolicy_-Order Permitted Permitted which is equal to Permitted. The whole evaluation process of the query TEquery3 is shown in Fig. 5.23.

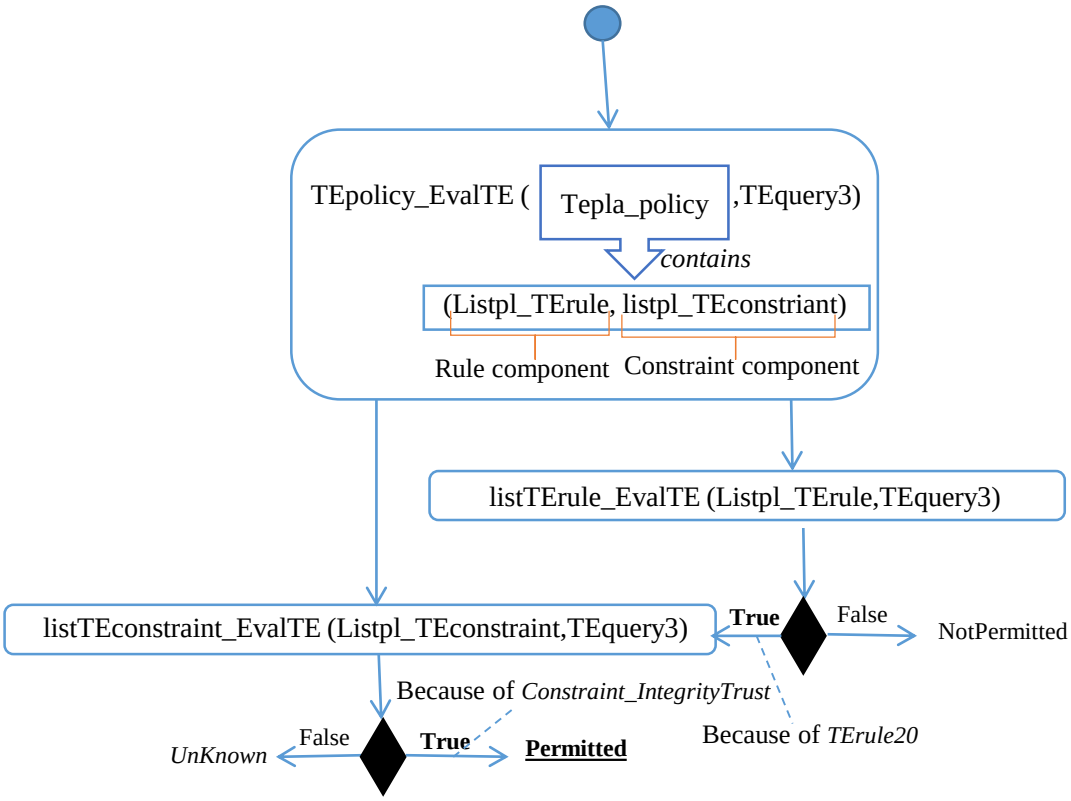

Figure 5.23: The overall evaluation process for the query `TEquery3`

We now consider the query `TEquery6`, which is granted by the rule `TErule2`, making the decision of the rule component of the policy `TEpla_policy`, `Permitted`. As a result, the evaluation of the query `TEquery6` continues by evaluating the constraints of the policy `TEpla_policy`. The constraint `Constraint_SoDB` is applicable to this query. As this constraint exploits the predicate `Prd_SeparationOfDuty` (defined in listing 5.12), the following condition is checked:

```
is_emptylisttype(
   IntersectionList
      (authoschemeSearch_findsubjectGeneral listpl_TErule
                              (singletype sysadm_passwd_t))
      (authoschemeSearch_findsubjectGeneral listpl_TErule
                              (singletype firewallguit_t)))
```

The result of this condition is `false`, which leads to the decision `UnKnown` of the constraint component of the policy `TEpla_policy`.

The condition is `false` because the first `authoschemeSearch_findsubject-General` returns the list `[singletype chrome_sandbox_tmp_t; singletype -iptables_t]` and the second `authoschemeSearch_findsubjectGeneral` returns the list `[singletype sysadm_passwd_t; singletype http_client_packet-_t; singletype chrome_sandbox_tmp_t; singletype inetd_t]`. The intersection (`IntersectionList`) of these lists is `[singletype chrome_sandbox_t-mp_t]`. Consequently, the intersection of these two lists is not empty, making the result of the constraint `UnKnown`. This makes evaluation of the query `TEquery6` by `TEpolicy_EvalTE` as `maximalpolicy_Order Permitted UnKnown` which is equal to `UnKnown`. As mentioned in Section 3.2.1, it is up to the discretion of security administrators to determine whether to grant or deny the queries with `UnKnown` decisions. This is because the query is authorized by the rule component of the policy; however, it does not satisfy the constraint. Accordingly, policy writers can use this information to debug or refine rules or constraints so that the new version returns `Permitted` or `NotPermitted`, instead of `UnKnown`. This kind of changes in policies is possible through a policy development process (as described in [Jaeger et al., 2003a] this process is similar to software development process) as new resources are added to the system or new access requirements are emerging.

Chapter 6

Conclusion and Future Work

We presented the infrastructure of TEpla, a certified Type Enforcement policy language, and the formal verification of this language. The infrastructure as well as the formal properties of TEpla are encoded in the Coq proof assistant. These machine-checked mathematical proofs guarantee that TEpla behaves as prescribed by the semantics. TEpla, with formal semantics and verified properties, is an essential step toward developing certifiably correct policy-related tools for Type Enforcement policies.

We used the Coq proof assistant for formal encoding and mechanized theorem proving of TEpla. It is true that we could use other tools for building machine-checked mathematical proofs of TEpla, such as Isabelle/HOL proof assistant [Nipkow et al., 2002]. Comparing Isabelle and Coq for our application is left for future work. The features of Coq that we found useful include Coq's higher-order functional programming language [Chlipala, 2019], because it allowed us to directly encode predicates used in TEpla constraints to express various security goals. Also, by encoding TEpla in Coq, we found that expressing datatypes and functions that are close to Coq's logic leads to straightforward proofs. In addition, proving properties formally helped us with recognizing and correcting some mismatch between our formal definitions of TEpla and the infrastructure of TEpla described in Chapter 3.

TEpla is certified in terms of formal semantics and machine-checked proofs of a particular set of properties. We analyzed the behavior of the language by defining different ordering relations on policies, queries, and decisions. These ordering relations enabled us to evaluate how language decisions react to changes in policies and queries. The behavior of the language according to the ordering relations of queries and policies was presented by a set of formal properties including *order preservation*, *independent composition*, *non-decreasing*, and *determinism*. This insight into language behavior provides a formal way to analyze and reason about language specifications, i.e., policies written in the language.

Moreover, we provide the language constructs for allowing security administrators to encode different security goals in policies. This makes the language flexible because policy

developers are not limited to built-in conditions to express their intended predicates. However, there are some limitations in the language, such as the limited number of arguments for constraints since we have provided just one language construct for constraints. We plan to enhance this aspect in the next versions of TEpla.

The certification of TEpla done so far will help with the development of the next versions of TEpla. First, after having developed the current version, it has become clear where some specific features and additional functionality will be useful. Second, as the proofs are updated for these changes, the proof development we have done so far will help to pinpoint exactly what parts of the proof need to be changed, and whenever problems with proofs are encountered, analyzing these problems will direct us toward changes needed to correct them. By combining the activities of adding to the language and certifying these changes, we will retain the property that the new language comes with a formal certification. Indeed, the current version of TEpla can already serve as a TE policy language; however, in comparison to policy languages that have been evolving for many years with many contributors and support of communities, it is limited by the issues that we list below, which are about how we plan to enhance the language constructs.

We can use the program extraction feature of Coq to generate a certified program from the algorithms used to express TEpla semantics, similar to what was done in [Capretta et al., 2007] for firewall policy evaluation. To obtain a more efficient version, one approach is to write such a version in Coq, prove it is equivalent to the one we have presented in this thesis, and then use program extraction on the efficient version. We can then continue our work in two possible directions to make further progress toward a practical implementation of TEpla. The first direction is to extend TEpla to include all the features of a language like SELinux, and then apply the techniques just mentioned to extract an efficient version that could replace SELinux in current operating systems. The second direction is to use the optimized extracted version of TEpla directly. To do so would require isolating the TE component of current practical implementations of SELinux and replace it with our implementation.

Developing a certified tool for analyzing and developing TE policies is one possible direction to continue this research. That is, owing to the formal behavior of TEpla, a future direction will be to develop a certified policy-related tool for TEpla. Automated policy generation from access requirements of a secure system and policy verification are two desirable features of such a certified policy-related tool. Moreover, another direction is to utilize the same approach that we used to develop TEpla, in developing a policy language for the semantic web that uses machine-readable data of different resources in its application. Therefore, we can exploit our approach in other domains such as the semantic web, in addition to its current application in operating systems. Further research on new formal properties that facilitate reasoning and analysis of policy languages used in the semantic web is another direction to continue this research.

Following the current version of TEpla, our future work can address these issues in the next versions of TEpla:

- Adding the capability to express queries for retrieving access information: Currently, queries are limited to inquire about access permissions of entities. We can enhance the language by providing information retrieval queries that inquire about retrieving access information of entities, such as formulating a query about retrieving all the entities that are accessed by a particular subject. By accommodating constructs similar to selector functions in the syntax of queries, we can develop a language construct to write queries for access information retrieval based on particular criteria.

- Extending TE rules: Adding support to cover more kinds of access rules is another future direction for TEpla. For example, over the years, SELinux has enabled policy developers to define access rules related to features such as entrypoint, declaring type aliases, and relabeling types, which are suitable to be covered in the next versions of TEpla. The concept of entrypoint in SELinux relates to permitting an entity to change its security context to the security context of a process if the entity is authorized to run a specific executable object which is considered to be the entrypoint of the process. The current structure of the language facilitates the addition of this feature because most of the language constructs are defined as inductive data types. Inductive data types provide the facility to create new datatypes by adding extra constructors to the previous ones, without the need to define new datatypes from scratch. This way of adding new features will also facilitate updating proofs because completing such proofs means considering only the new cases for the new constructors and does not require changing the previously proved sub-cases.

- Constructing attributes of the environment: Including the capability to express information about the environment in policy languages provide mechanisms to refine further the kinds of accesses allowed. Likewise, we will include environments in TEpla to allow policy writers to express these attributes in their policies. The current version has some capability in this direction because of the inclusion of boolean conditions in access rules. Therefore, we will complete this feature of the language to enable policy writers to use attributes in expressing conditions on the environment.

- Providing language constructs for grouping object classes and actions: Currently, TEpla allows grouping of `basictypes` in logical groups through `attributes`. We will add a similar capability for object classes and actions. By adding this feature, policy developers can compare object classes and actions, such as checking membership to a particular set, when they define predicates. Accordingly, by adding this feature, the third and fourth arguments of predicates, which we do not use in our example predicates, will be useful for expressing a larger class of predicates. Furthermore, this feature will reduce the number of access rules in policies because to allow, for example, a particular access to perform two actions, we need to define two different rules, whereas by grouping the two actions in a single logical group, one rule that uses this logical group in its action part, can express the same regulation. Alternately, as part of a concrete syntax, we could allow one rule with macro expansion to more than one. In addition, we will restrict the set of authorized actions for each

object class to incorporate more of the functionality of SELinux. This restriction means that an object class has a particular set of actions, which are the only actions that subjects can perform on the entities of the object class.

- Re-engineering the structure of constraints: Although predicates are one of the strengths of TEpla, the number and built-in types of the arguments in their current definition limit them. We will address this issue by re-engineering the structure of constraints. This is possible as Coq supports polymorphism and inductive types, two concepts that can help to reach this goal. Polymorphism helps us to develop constraints that can infer the types of arguments implicitly. For instance, constraints that check membership of an element in a particular list can take advantage of polymorphism since the constraint can be used to apply the same logic for lists of different types. Using inductive types for constraints has the same advantages as discussed for TE rules.

- Providing a comprehensive list of selector and operator functions: In Chapter 5, we introduced a number of selector and operator functions. Adding to these functions can help with the development of constraints.

To sum up, TEpla is a certified solution for challenges and drawbacks that security administrators face in developing or analyzing security policies. It presents a structured way to address these issues, not only for policy developers but also for policy-related tools. As an appropriate candidate for a certified TE policy language, we plan to foster it by including more features and language constructs.

References

[Abbas et al., 2018] Abbas, M., Ben-Yelles, C., and Rioboo, R. (2018). Formalizing UML/OCL multiple inheritance with FoCaLiZe. In *2018 International Conference on Smart Communications in Network Technologies (SaCoNeT)*, pages 261–266. IEEE.

[Abelson and Sussman, 1996] Abelson, H. and Sussman, G. J. (1996). *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, MA, USA, 2nd edition.

[Amthor et al., 2011] Amthor, P., Kühnhauser, W. E., and Pölck, A. (2011). Model-based safety analysis of SELinux security policies. In *5th International Conference on Network and System Security (NSS)*, pages 208–215.

[Archer et al., 2003] Archer, M., Leonard, E. I., and Pradella, M. (2003). Modeling Security-Enhanced Linux policy specifications for analysis. In *3rd DARPA Information Survivability Conference and Exposition (DISCEX-III)*, pages 164–169 vol.2.

[Azevedo de Amorim, 2016] Azevedo de Amorim, A. (2016). Binding operators for Nominal Sets. *Electronic Notes in Theoretical Computer Science*, pages 3–27 vol. 325.

[Benzaken et al., 2014] Benzaken, V., Contejean, É., and Dumbrava, S. (2014). A Coq formalization of the relational data model. In *Programming Languages and Systems (ESOP)*, pages 189–208, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Bertot and Castéran, 2004] Bertot, Y. and Castéran, P. (2004). *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer Science & Business Media.

[Bishop, 2002] Bishop, M. A. (2002). *The Art and Science of Computer Security*. Addison-Wesley Longman Publishing Co., Inc.

[Capretta et al., 2007] Capretta, V., Stepien, B., Felty, A., and Matwin, S. (2007). Formal correctness of conflict detection for firewalls. In *ACM Workshop on Formal Methods in Security Engineering (FMSE)*, pages 22–30.

[Chen and Kao, 2006] Chen, Y. and Kao, Y. (2006). Information flow query and verification for security policy of Security-Enhanced Linux. In *1st International Workshop on Security (IWSEC)*, pages 389–404.

[Chlipala, 2019] Chlipala, A. (2019). *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press. https://mitpress.mit.edu/books/certified-programming-dependent-types.

[Clemente et al., 2012] Clemente, P., Kaba, B., Rouzaud-Cornabas, J., Alexandre, M., and Aujay, G. (2012). SPTrack: Visual analysis of information flows within SELinux policies and attack logs. In *International Conference on Active Media Technology (AMT)*, pages 596–605.

[Eaman, 2019] Eaman, A. (2019). The Coq Development of TEpla. http://www.site.uottawa.ca/~afelty/eaman-thesis/teplacoqsource.v.

[Eaman et al., 2017] Eaman, A., Sistany, B., and Felty, A. (2017). Review of existing analysis tools for selinux security policies: Challenges and a proposed solution. In *7th International Multidisciplinary Conference on e-Technologies (MCETECH)*, pages 116–135.

[Fedora Linux, 2019] Fedora Linux (2019). The Fedora Linux Official Website. https://getfedora.org/.

[Gonthier and Mahboubi, 2010] Gonthier, G. and Mahboubi, A. (2010). An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning, ASDD-AlmaDL*, 3(2):95–152.

[Grégoire and Tassi, 2016] Grégoire, B. and Tassi, E. (2016). Boolean reflection via type classes. In *Coq Workshop*, Nancy, France. HAL Id: hal-01410530.

[Guttman et al., 2005] Guttman, J. D., Herzog, A. L., Ramsdell, J. D., and Skorupka, C. W. (2005). Verifying information flow goals in Security-Enhanced Linux. *Journal of Computer Security*, 13(1):115–134.

[Harrison et al., 1976] Harrison, M. A., Ruzzo, W. L., and Ullman, J. D. (1976). Protection in operating systems. *Communications of the ACM*, 19(8):461–471.

[Harzheim, 2005] Harzheim, E. (2005). *Ordered Sets.* Springer-Verlag US. ISBN: 978-0-387-24219-40.

[Hausmann, 2005] Hausmann, J. H. (2005). *Dynamic Meta Modeling : a semantics description technique for visual modeling languages.* PhD thesis. University of Paderborn.

[Hurd et al., 2009] Hurd, J., Carlsson, M., Finne, S., Letner, B., Stanley, J., and White, P. (2009). Policy DSL: High-level specifications of information flows for security policies. *High Confidence Software and Systems (HCSS).*

[Huth and Ryan, 2004] Huth, M. and Ryan, M. (2004). *Logic in Computer Science: Modelling and reasoning about Systems.* Cambridge University Press, New York, NY, USA.

[Jaeger et al., 2003a] Jaeger, T., Sailer, R., and Zhang, X. (2003a). Analyzing integrity protection in the SELinux example policy. In *12th USENIX Security Symposium*, pages 59–74. USENIX Association.

[Jaeger and Tidswell, 2001] Jaeger, T. and Tidswell, J. (2001). Practical safety in flexible access control models. *ACM Transactions on Information and System Security (TISSEC)*, 4:158–190.

[Jaeger et al., 2003b] Jaeger, T., Zhang, X., and Edwards, A. (2003b). Policy management using access control spaces. *ACM Transactions on Information and System Security (TISSEC)*, 6(3):327–364.

[Kissinger and Hale, 2006] Kissinger, A. and Hale, J. C. (2006). Lopol: A deductive database approach to policy analysis and rewriting. In *Security-Enhanced Linux Symposium*, pages 388–393, Baltimore, Maryland, USA.

[Kuliniewicz, 2006] Kuliniewicz, P. (2006). Seng: An enhanced policy language for SELinux. Security Enhanced Linux Symposium. Maryland, USA.

[Leroy, 2012] Leroy, X. (2012). Mechanized semantics for compiler verification. In *Proceedings of the Second International Conference on Certified Programs and Proofs*, CPP'12, pages 4–6. Springer-Verlag.

[Loscocco and Smalley, 2001] Loscocco, P. and Smalley, S. D. (2001). Meeting critical security objectives with Security-Enhanced Linux. In *Ottawa Linux Symposium*, pages 115–134.

[Marouf and Shehab, 2011] Marouf, S. and Shehab, M. (2011). SEGrapher: Visualization-based SELinux policy analysis. In *4th Symposium on Configuration Analytics and Automation (SAFECONFIG)*, pages 1–8.

[Mayer et al., 2006] Mayer, F., Caplan, D., and MacMillan, K. (2006). *SELinux by Example: Using Security Enhance Linux*. Prentice Hall.

[Nakamura et al., 2009] Nakamura, Y., Sameshima, Y., and Tabata, T. (2009). SEEdit: SELinux security policy configuration system with higher level language. In *23rd Large Installation System Administration Conference*, pages 107–117.

[National Security Agency, 2019] National Security Agency (2019). Security-Enhanced Linux. `https://www.nsa.gov/what-we-do/research/selinux/`.

[Nipkow et al., 2002] Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag. Volume 2283 of Lecture Notes in Computer Science.

[Parrend and Frénot, 2008] Parrend, P. and Frénot, S. (2008). Classification of component vulnerabilities in java service oriented programming (sop) platforms. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering (CBSE)*, pages 80–96. Springer-Verlag.

[Pierce et al., 2019] Pierce, B. C., Azevedo de Amorim, A., Casinghino, C., Gaboardi, M., Greenberg, M., SjÃűberg, C. H. V., and Yorgey, B. (2019). *Software Foundations*. Online Book, University of Pennsylvania, `https://softwarefoundations.cis.upenn.edu/`.

[Pierce et al., 2008] Pierce, B. C., Sewell, P., Weirich, S., and Zdancewic, S. (2008). *It is Time to Mechanize Programming Language Metatheory*, pages 26–30. Springer Berlin Heidelberg, Zurich, Switzerland.

[Quigley, 2007] Quigley, D. P. (2007). PLEASE: Policy language for easy administration of SELinux. Master's thesis, Stony Brook University. Technical Report FSL-07-02, `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.131.1594&rep=rep1&type=pdf`.

[Radhika et al., 2018] Radhika, B. S., Kumar, N. V. N., and Shyamasundar, R. K. (2018). Flowconseal: Automatic flow consistency analysis of seandroid and selinux policies. In *Data and Applications Security and Privacy XXXII, IFIP Conference*, pages 219–231. Springer International Publishing AG.

[Reshetova et al., 2017] Reshetova, E., Bonazzi, F., and Asokan, N. (2017). SELint: an SEAndroid policy analysis tool. *3rd International Conference on Information Systems Security and Privacy*, abs/1608.02339:47–58.

[Reshetova et al., 2015] Reshetova, E., Bonazzi, F., Nyman, T., Borgaonkar, R., and Asokan, N. (2015). Characterizing SEAndroid policies in the wild. *eprint arXiv:1510.05497*, abs/1510.05497.

[Sellers et al., 2006] Sellers, C., Athey, J., Shimko, S., Mayer, F., Wilson, A., and MacMillan, K. (2006). Experiences implementing a higher-level policy language for SELinux. Security Enhanced Linux Symposium. `http://selinuxsymposium.org/2006/papers/08-higher-level-experience.pdf`.

[Sengupta and Bhattacharya, 2008] Sengupta, S. and Bhattacharya, S. (2008). Formalization of UML diagrams and their consistency verification: A z notation based approach. In *Proceedings of the 1st India Software Engineering Conference*, ISEC '08, pages 151–152. ACM.

[Singh et al., 2007] Singh, A., Ramakrishnan, C. R., Ramakrishnan, I. V., Stoller, S. D., and Warren, D. S. (2007). Security policy analysis using deductive spreadsheets. In *ACM workshop on Formal methods in Security Engineering (FMSE)*, pages 42–50.

[Sistany, 2016] Sistany, B. (2016). *A Certified Core Policy Language*. PhD thesis, University of Ottawa. https://ruor.uottawa.ca/handle/10393/34865.

[Sistany and Felty, 2017] Sistany, B. and Felty, A. (2017). A certified core policy language. In *15th Annual International Conference on Privacy, Security, and Trust (PST)*, pages 391–393.

[St-Marting, 2012] St-Marting, M. (2012). A verified algorithm for detecting conflicts in XACML access control rules. Master's thesis, University of Ottawa. https://ruor.uottawa.ca/handle/10393/20539.

[Stallings and Brown, 2018] Stallings, W. and Brown, L. (2018). *Computer Security, Principles and Practices*. Pearson Education. ISBN: 978-0-13-479410-5.

[Stepien and Felty, 2016] Stepien, B. and Felty, A. (2016). Using expert systems to statically detect dynamic conflicts in XACML. In *11th International Conference on Availability, Reliability and Security (ARES)*, pages 127–136.

[The Coq Development Team, 2019] The Coq Development Team (2019). *The Coq Proof Assistant Reference Manual, version 8.9.0*. INRIA. https://coq.inria.fr/refman/.

[Tresys Technology, 2019] Tresys Technology (2019). APOL. SETools: Policy analysis tools for SELinux https://github.com/SELinuxProject/setools.

[Tschantz, 2005] Tschantz, M. C. (2005). *The Clarity of Languages for Access-Control Policies*. PhD thesis, Brown University.

[Tschantz and Krishnamurthi, 2006] Tschantz, M. C. and Krishnamurthi, S. (2006). Towards reasonability properties for access-control policy languages. In *11th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 160–169.

[Walick, 2016] Walick, M. (2016). *Introduction to Mathematical Logic:Extended Edition*. World Scientific Publishing, Singapore, extended edition.

[Wang et al., 2015] Wang, R., Enck, W., Reeves, D. S., Zhang, X., Ning, P., Xu, D., Zhou, W., and Azab, A. M. (2015). EASEAndroid: Automatic policy analysis and refinement for Security-Enhanced Android via large-scale semi-supervised learning. In *24th USENIX Security Symposium*.

[Xu et al., 2013] Xu, W., Shehab, M., and Ahn, G. (2013). Visualization-based policy analysis for SELinux: framework and user study. *International Journal of Information Security*, 12(3):155–171.

[Xu et al., 2009] Xu, W., Zhang, X., and Ahn, G. (2009). Towards system integrity protection with graph-based policy analysis. In *Data and Applications Security XXIII, 23rd Annual International Federation for Information Processing (IFIP)*, pages 65–80.

[Zanin and Mancini, 2004] Zanin, G. and Mancini, L. V. (2004). Towards a formal model for security policies specification and validation in the selinux system. In *9th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 136–145. ACM Press.

[Zhai et al., 2014] Zhai, G., Guo, T., and Huang, J. (2014). SCIATool: A tool for analyzing SELinux policies based on access control spaces, information flows and CPNs. In *Trusted Systems—6th International Conference (INTRUST)*, pages 294–309. Springer International Publishing.

[Zook, 2016] Zook, J. T. (2016). High-level formal modeling and verification of mandatory access control policies across multiple SELinux devices. Master's thesis, University of Idaho. `https://digital.lib.uidaho.edu/digital/collection/etd/id/1206/`.