Innovations in Systems and Software Engineering
https://doi.org/10.1007/s11334-023-00534-1

ORIGINAL ARTICLE

A certified access control policy language: TEpla

Amir Eaman[1,2] · **Amy Felty**[2]

Received: 19 July 2023 / Accepted: 20 July 2023

Abstract
Access control is an information security process which guards protected resources against unauthorized access, as specified by restrictions in security policies. A variety of policy languages have been designed to specify security policies of systems. In this paper, we introduce a certified policy language, called TEpla, with formal semantics and simple language constructs, which we have leveraged to express and formally verify properties about complex security goals. In developing TEpla, we focus on security in operating systems and exploit *security contexts* used in the *Type Enforcement* mechanism of the SELinux security module. TEpla is certified in the sense that we have encoded the formal semantics and machine-checked the proofs of its properties using the Coq Proof Assistant. In order to express the desired properties, we first analyze the behavior of the language by defining different ordering relations on policies, queries, and decisions. These ordering relations enable us to evaluate how algorithms for deciding whether or not requests are granted by policies will react to changes in policies and queries. The machine-checked mathematical proofs guarantee that TEpla behaves as prescribed by the semantics. TEpla is a crucial step toward developing certifiably correct policy-related tools for Type Enforcement policies.

Keywords Access control · Policy languages · Formal methods

1 Introduction

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. Access control policy languages have an essential role in expressing the intended access authorization to regulate requests to resources. Security policy languages used to develop security policies significantly affect this process, mainly because the policy developers' understanding of the semantics of the languages has a direct effect on the way they write policies. Formal semantics can tremendously improve the use of a language by constructing a precise reference for the underlying language. Semantic-related tools which analyze or reason about specifications written in the language require formal semantics to process the language correctly. Moreover, the implementation of such tools can be verified, which is another important consequence of formal semantics.

We propose a small and certifiably correct policy language, TEpla. The design of TEpla is motivated, on the one hand by the necessity of supporting features of real policy languages (like those of SELinux) and on another hand by the necessity of staying generic, thus amenable to capture various types of security policies and relations between them. TEpla can provide ease of use, analysis, and verification of its properties. By *certified* policy language, we mean a policy language with formal semantics and formally verified mathematical proofs of important properties, which reflects the concept of certification in formal methods communities and programming languages [1]. One of our goals is to avoid language-introduced errors (i.e., errors that are introduced to IT systems due to multiple contradictory interpretations of policies). Ease of reasoning and analysis of policies is facilitated by a clear specification of TEpla's behavior and

Amy Felty has contributed equally to this work.

✉ Amir Eaman
amir.eaman@acadiau.ca

Amy Felty
afelty@uottawa.ca

[1] Jodrey School of Computer Science, Acadia University, 15 University Ave, Wolfville B4P 2P7, NS, Canada

[2] School of Electrical Engineering and Computer Science, University of Ottawa, 800 King Edward Ave, Ottawa K1N 6N5, ON, Canada

Published online: 28 August 2023

semantics as it satisfies important formal properties designed for this purpose [2]. In addition to these properties, TEpla is flexible enough for defining complex security constraints through introducing user-defined predicates. This enables security administrators to define various security goals in security policies. We analyze the language's behavior by defining different ordering relations on policies, queries, and decisions. These ordering relations enable us to evaluate how language decisions react to changes in policies and queries. See, for example, the *non-decreasing* property of TEpla policies discussed in Sect. 4.3.

In order to keep the core of the language simple, we focus on developing a new certified policy language for the *Type Enforcement* mechanism, which is a subset of the SELinux security module [3] implemented in Linux distributions. Type Enforcement exploits the *security context* of resources to regulate accesses. The security context is a set of allowable values for particular attributes assigned to system resources.

SELinux is a linux security module (LSM) that enables security developers to define security policies. It implements the mandatory access control (MAC) [4] strategy, which allows policy writers to express whether a *subject* can perform an operation on an *object*, e.g., whether an SELinux process can perform a read or write on a file or socket.

We carried out a study [5] on policy languages, which proposes solutions for dealing with the many gaps for using policy languages with informal semantics, mainly focusing on the SELinux policy language in particular, and gaps in developing verified security policies in general. TEpla is an important step in closing these gaps. We believe that the same development paradigm used for TEpla can be adopted to develop other verified policy languages, such as one for AppArmor [6] or one for full SELinux, thus providing higher-trust policy languages for Linux.

As mentioned earlier, TEpla also provides additional language constructs that allow security administrators to encode different security goals in policies as user-defined predicates. Using this mechanism, administrators can express a variety of conditions, thus significantly increasing the flexibility over the language's built-in conditions. However, there are some conditions that policy writers need to verify about their predicate definitions in order to ensure that their defined predicates are compatible with TEpla properties. Note that our proof development uses no axioms; we require all conditions to be proved.

We use the Coq Proof Assistant [7, 8] (version 8.16) to develop machine-checked mathematical proofs for TEpla's properties. The Coq development of TEpla contains approximately 4700 lines of script and is available in an online appendix [9]. This appendix also contains a mapping from names used in this paper to names used in the Coq code.

In Sect. 2, we describe the TEpla language structures and their meaning, including rules, decisions, queries, constraints, and policies. A constraint can be considered as an additional form of a policy rule, which takes a user-defined predicate as an argument. This section also defines ordering relations on TEpla decisions, policies, and queries. In Sect. 3, we present the Coq encoding of the syntax and semantics of TEpla. In Sect. 4, we discuss the main properties that we have proved about TEpla, and Sect. 5 concludes the paper. Appendix 1 contains the full BNF grammar of TEpla.

This paper is an extended version of [10]. The work presented here also appears in the Ph.D. thesis of the first author [11], and the reader can find further details there.

2 Infrastructure of TEpla

The main element in a system is a *resource*, which can be either a subject or an object, as described in the previous section. In fact, a resource can act as a subject in some contexts and an object in others. In many policy languages, including TEpla, resources have attributes. As mentioned, the values of these attributes form the security context of the resources. The main building block of TEpla is *type* which is the core language concept in TEpla's syntax and semantics. This section outlines the key language concepts of TEpla. In the next section, we go a bit deeper and go through the overall mechanized formalization of infrastructure of TEpla using the Coq Proof Assistant.

2.1 Overview

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. In TEpla, the security context is the values of an attribute called *basic type*. Each resource is assigned one basic type, providing it with an identity in the same way as done in SELinux. For example, consider two resources of a system called *file_web* and *port_protocol*. We can assign, for instance, the values of the basic type attribute to be `mail_t` and `http_t`, respectively.

TEpla allows policy developers to group basic types of resources together to form a *group type*, providing a single identifier for a group of resources. We group together basic types when there exists a conceptual relationship among them. For example, we can group together

the basic types mentioned above to form the group type {mail_t, http_t}, here represented as a set. Basic and group types together form the notion of a *type*. Thus mail_t, http_t, and {mail_t, http_t} are all examples of types.

SELinux uses the terminology *source* and *destination* to mean subjects and objects, and *domain* and *type* to classify their types, respectively. Here, we continue to use *subject* and *object* and we use *type* to classify both.

Two other central data types in TEpla include *object class* and *permitted action*. Object classes specify possible instances of all resources of a certain kind, such as files, sockets, and directories. In particular, primarily, this grouping is used to define a set of *permitted action*s for each group (i.e., object class). Permitted actions specify the actions that subjects are authorized to perform on objects. Permitted actions can range from being as simple as reading data, sharing data, or executing a file [12].

2.2 Language

As mentioned above, the primitive attributes include *type*, *Object Class*, *Permitted Action*. Note that the concept of type is different from *Object Class* 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 to which each element belongs. 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 set of *permitted action*s for each *object class*) as future work. These data types provide a straightforward syntax for policy writers. The BNF grammar of TEpla is defined in Appendix 1 in Fig. 3. In the following sections, we denote the terms which are used in the BNF grammar inside parentheses after the name of the language constructs.

2.3 Access 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 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*). That is, TEpla policies (*TEpolicy*) consist of a pair of a *TErules* sequence and a *TEcontraints* sequence (see Sects. 2.4 and 2.6). Constraints add conditions on the rules that may replace a decision obtained from considering only the rules alone.

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. Policies thus can be formed by any sequences of *TErules* and *TEconstraints*.

Figure 1 represents architecture and decision-making process of TEpla. Access requests are first evaluated against allow rules of policies. A query which is not granted by the allow rules will cause a *NotPermitted* decision without considering the decision for constraints. If a query is authorized by the allow rules, it will be checked against constraints to check if it violates any security goals. Two possible decisions of checking a query against constraints of the policies are *Permitted* and *Unknown*, which means that the query satisfies the security goals, or violates a security goal, respectively. If none of the constraints are applicable to the query, the decision of the allow rules (i.e., *Permitted*) will be the final access decision for the query (see Sect. 2.9)

Type Enforcement exploits the security context of resources to regulate accesses. As mentioned above, every system resource is labeled with a value from the *basictype* syntax

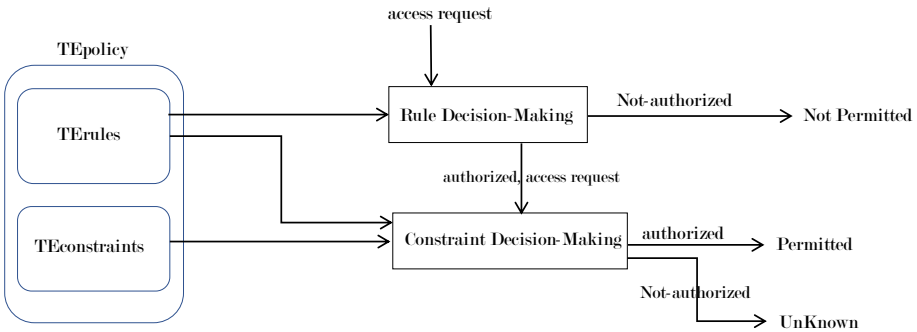

Fig. 1 The overall architecture and decision-making flow in TEpla

class. This makes TEpla a fine-grained policy language as administrators can apply different access regulations on different entities of a system.

2.4 Policy rules

Allow rules (*TErule*s) 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 *Boolean Condition* (*cond_bool*). *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 *Boolean Condition* 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. As an example, suppose `File` is an *Object Class* and `Read` is a *Permitted Action*. Then, ({`mail_t`, `http_t`}, `mail_t`, `File`, `Read`, `true`) is an example *Allow* rule. We note that we do not use the last argument in this paper, so the value will always be `true`; we discuss it further in future work.

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 set of at least size two *basictype*s, is left for future work.

2.5 Access requests

Access requests or *queries* consist of four components *Source Type*, *Destination Type*, *Object Class* and *Requested Action*. Access requests are inquiries into the policy to check the possibility that *Source Type* is allowed to perform the action *Requested Action* on the object *Destination Type* of the object class *Object Class* (see the BNF grammar of TEpla in Fig. 3). 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. Continuing our example, the following is a sample query: (`mail_t`, `http_t`, `File`, `Write`). In this request, a subject of type `mail_t` is requesting to write to an object whose class is `File` and whose type is `http_t`.

2.6 Constraints

As discussed earlier, rules alone cannot always 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 with 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, (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 constraint is only applicable when the values of these components match. The next three arguments are *type*, *type*, and {*type*} (i.e., a set of *type*s) whose values are provided by the policy writers; they can provide important information when a constraint is evaluated.

*TEconstraint*s in TEpla include a function that returns a Boolean as their last argument, expressing when the constraint is satisfied.

To illustrate constraints and predicates, we use a "separation of duty" running example defined as follows, which uses the predicate `Prd_SoD`, defined in the next subsection: (`File`, `Read`, {`mail_t`, `http_t`}, `networkManager_ssh_t`, [], `Prd_SoD`). This constraint only allows subjects whose types are elements of {`mail_t`, `http_t`} to perform the action `Read` on objects whose basic type is `networkManager_ssh_t` and whose object class is `File` as long as the additional requirement is met that objects of types {`mail_t`, `http_t`} and `networkManager_ssh_t` are never permitted to be acted upon by subjects of the same type. `Prd_SoD` will formally express what is meant by this additional requirement. Informally, whenever two *Allow* rules permit subjects of the same type to perform actions, if the object in one of the rules has a type in {`mail_t`, `http_t`}, then the object in the other rule cannot have type `networkManager_ssh_t`. Similarly, if the object in one rule has type `networkManager_ssh_t`, then the object in the other rule cannot have a type that is a subset of {`mail_t`, `http_t`}. This constraint is applicable

to all queries whose *Object Class* and *Permitted Action* are `File` and `Read`, respectively.

2.7 Predicates

Predicates (*TEpredicates_ID*), i.e., a function that returns a Boolean, are the last input argument of a *TEconstraint*, which makes these functions act as *predicates* [13]. 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. *TEpredicates_ID* has eight arguments, which supply a comprehensive set of values by which policy developers can define the required security criteria. We will see in Sect. 2.9 how a query is evaluated against a constraint. Here, we present the definition of `Prd_SoD`.

Algorithm 1 Defining the predicate Prd_SoD

Function Prd_SoD (*rules*: list *TErule*, *types*:list *type*, *sClass*:*cls*, *perm*:*prm*, (*qSrcT qDestT predSrcT predDestT*: *type*))
if $(qSrcT \subseteq predSrcT) \wedge (qDestT \subseteq predDestT)$ **then**
  list1 ← search *rules* to find subject *types* that access *predSrcT*
  list2 ← search *rules* to find subject *types* that access *predDestT*
  return $(list1 \cap list2) = \emptyset$
else
  return *true*

In the above procedure, $T_1 \subseteq T_2$ means that all the basic types in T_1 are also included in T_2. The parameter *rules* 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 body of this predicate function searches the first input argument, i.e., *rules*, 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`. As we will see in Sect. 2.9, 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).

2.8 Ordering relation on decisions, queries, and policies

We define a *Partially Ordered Set (poset)* [13] called $(\mathbb{DCS}, <::)$ on TEpla's three-valued set of decisions as $NotPermitted <:: Permitted <:: UnKnown$. The lowest decision in this ordering is *NotPermitted*, which means that all accesses are first denied by default. To permit an access query, a relevant rule in the first component of policies must authorize the access. If the query is not granted at this stage, TEpla denies the access, which means that the ultimate access decision is *NotPermitted*. In the case that the query is granted (with decision *Permitted*), TEpla proceeds to check whether or not the query satisfies the constraint component 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 changes to *UnKnown* (see Fig. 1). 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 Sect. 4.3 for more details about this property).

Additionally, we define a relation on queries $(\mathbb{Q}, <<=)$. Two queries $\mathbb{Q}_1 = (srcTQ_1, dstTQ_1, clsQ_1, prmActQ_1)$ and $\mathbb{Q}_2 = (srcTQ_2, dstTQ_2, clsQ_2, prmActQ_1)$ are in relation $\mathbb{Q}_1 <<= \mathbb{Q}_2$ if and only if $srcTQ_2 \subseteq srcTQ_1$ and $dstTQ_2 \subseteq dstTQ_1$ hold.

Finally, we define the binary relation $(\mathbb{TEPLCY}, \lesssim)$ on policies, where $p_1 \lesssim p_2$ whenever p_2 has more information that p_1. More formally:

$$\forall(p_1, p_2 \in \mathbb{TEPLCY}), p_1 \lesssim p_2 \textit{ iff } length(p_1) \leq length(p_2) \wedge p_1 \subseteq p_2.$$

In this definition, *length* means the sum of the lengths of the rule component and the constraint component of a policy. We call the combined list *authorization rules*. Here, we overload the $\subseteq$ operator; $p_1 \subseteq p_2$ means that p_2 has more authorization rules and it contains all the authorization rules in p_1.[1]

Policy writers should make sure the predicates that they develop satisfy certain conditions. Note that this checking is performed statically for predicates, by proving the required properties before using them in policies, and there is no dynamic checking needed. These conditions express that given two queries related by "<<=" or two policies related by "$\lesssim$," the evaluation of predicates by the function that evaluates a query against a constraint (Algorithm (3) in Sect. 2.9) preserves the defined order on decisions "<::."

2.9 Semantics

We define the semantics of TEpla as a mapping from policies and access requests to decisions, in the form of five translation functions, which together act as the decision-making chain that evaluates a query against a policy, taking into account all the various parts of the policy. The first function, shown in Algorithm 2, evaluates a query against a single rule leading to a decision of either *Permitted* or *NotPermitted*.

[1] In the Coq implementation, we do not have a separate definition for $\lesssim$. Instead, we express it directly when needed using list operators.

Algorithm 2 Evaluating a Query against an Allow rule

Function Translation Function 1 (*allowRule, query*)

(*srcType,dstType,clsRule,prmActions,boolCond*) ← *allowRule*
(*srcTQuery,dstTQuery,clsQuery,prmActQry*) ← *query*

if *((boolCond)*∧*(srcTQuery* ⊆ *srcType)*∧*(dstTQuery* ⊆ *dstType)*
∧ *(clsQuery = clsRule)* ∧*(prmActQry = prmActions))* **then**
  return *Permitted*
else
  return *NotPermitted*

For *Allow* rules, the authorization conditions include checking to see whether a rule applies to the query, the Boolean condition is true, types of the subject and object in the query are a subset of the corresponding types in the rule, and the object class and permitted action are the same. If all conditions are satisfied, the result is *Permitted*; otherwise, it is *NotPermitted*. The function for *Type Transition* rules is similar (details omitted), but only the types of the subject and object need to be checked in order to determine that the rule applies.

The second translation function, shown in Algorithm 3, evaluates a query against a constraint. It takes a single constraint, a query, and a list of rules (all the rules in the rule component of a policy) as arguments. The rules argument can be used to extract access information required for expressing security goals encoded in predicates. In order to check whether or not the constraint is applicable to the query, the object class and permitted action components are compared and must be the same. If applicable, the constraint predicate is checked. Note that the arguments passed to *Prdct* include the list of rules as well as all the other components of the constraint and query, except the two that are used to check the applicability of the constraint. If the evaluation of the predicate returns *true*, then the decision is *Permitted*. Otherwise, the decision is *UnKnown*. Note that if the constraint does not apply to the query, the default value *NotPermitted* is returned, which means the constraint has no role in changing the access decision for the query, and other parts of the policy will make the final decision.

Algorithm 3 Evaluating a Query against a Constraint

Function Translation Function 2 (*cstrt,query,policyRules*)

(*clsCst,prmCst,tArg1,tArg2,typeList,Prdct*) ← *cstrt*
(*srcQry,dstQry,clsQry,prmQry*) ← *query*

if *(clsCst=clsQry)* ∧ *(prmCst=prmQry)* **then**
  if *(Prdct(policyRules,typeList,clsCst,prmCst,srcQry,dstQry,tArg1,tArg2))* **then**
    return *Permitted*
  else
    return *UnKown*
else
  return *NotPermitted*

A query must be evaluated against all the rules and constraints in a policy. We omit the other translation functions that are defined to complete this task. They include two functions for processing every element in a list against the query, one for rules and one for constraints, plus a function for putting everything together to evaluate a query against a policy. The Coq implementation of the main translation function that combines the decisions of different parts of policies, taking into account the ordering relations we defined on decisions in Sect. 2.8, is called `TEPLCY_EvalTE`; it takes a policy and query as its two input arguments.

3 Coq development

In this section, we describe the Coq implementation of the infrastructure of TEpla (i.e., the elements of syntax and semantics of TEpla described in Chapter 2). As mentioned above, the whole Coq development of TEpla is available in [9]. All the proofs of lemmas and theorems of TEpla are available in the Coq source of TEpla.

3.1 Syntax in Coq

We start by defining the basic data types. Here, `C`, `P`, `basicT`, which represent object classes, permitted actions, and basic types, respectively, are all defined as `nat` (ℕ), which is the datatype of natural numbers in Coq [8]. These definitions plus the implementation of some examples discussed earlier are below.

```
Definition C := N. Definition P := N. Definition
    basic T := N.
Definition File : C := 600.
Definition mail_t : basic T := 300. Definition http_t
    : basic T := 301.
Definition networkManager_ssh_t : basic T := 302.
Definition Read : P := 702.   Definition Write : P :=
    703.
```

We encode a group type as a list of basic types, i.e., we represent them using Coq's built-in datatype for lists. For example, the code below introduces `G` to define group types and `program_G`, which represents the example set {`mail_t`, `http_t`}. A group type should contain at least 2 elements.

```
Definition G : Set := list basic T .
Definition program_ G : G := [mail_t;http_t].
```

We can now encode our principle entity, the *type* structure; we define the inductive datatype `T` with two constructors `singleT` and `groupT`. These constructors take arguments of type `basicT` and `G`, respectively, to produce a term belonging to `T`.

```
Inductive T : Type:=
  | single T : basic T → T
```

```
| group 𝕋 : 𝔾 → 𝕋.
```

Continuing our example, consider two subjects whose security contexts are represented by the values `http_t` and `mail_t`, and a third subject that is allowed to access objects of both types. These are represented by (`single𝕋 http_t`), (`single𝕋 mail_t`), and (`group𝕋 program_𝔾`), respectively.

Rules, which include both *Allow* and *Type Transition*, are encoded as the inductive type ℝ. This definition is followed by an encoding of the example *Allow* rule given in Sect. 2.4.

```
Inductive ℝ : Set :=
 | Allow : 𝕋 * 𝕋 * ℂ * ℙ * 𝔹 → ℝ
 | Type_Transition : 𝕋 * 𝕋 * ℂ → ℝ.
Definition ℝ_A : ℝ :=
 Allow (group 𝕋 program_𝔾, single 𝕋 mail_t, File,
     Read, true).
```

The * operator represents tuple types in Coq; *Allow* rules are represented using a 5-tuple and *Type Transition* rules are represented using a triple.

Decisions are defined by the inductive type 𝔻ℂ𝕊 and access requests or queries (ℚ) by a 4-tuple. These definitions along with the example query from Sect. 2.5 are below:

```
Inductive 𝔻ℂ𝕊 : Set := Permitted | NotPermitted |
    UnKnown.
Definition ℚ : Set := 𝕋 * 𝕋 * ℂ * ℙ.
Definition sample ℚ : ℚ := (single 𝕋 mail_t, single
    𝕋 http_t, File, Write).
```

Constraints are defined below as the type ℂ𝕊𝕋𝔼. This definition is followed by the example constraint from Sect. 2.6.

```
Inductive ℂ𝕊𝕋𝔼 : Set :=
 | Constraint : ℂ * ℙ * 𝕋 * 𝕋 * list 𝕋 *
    (list ℝ → list 𝕋 → ℂ → ℙ → 𝕋 → 𝕋 → 𝕋 → 𝕋
        → 𝔹) → ℂ𝕊𝕋𝔼.
Definition ℂ𝕊𝕋𝔼_SoD : ℂ𝕊𝕋𝔼 := Constraint(File,
    Read, group 𝕋 program_𝔾,
                    single 𝕋
                        networkManager_ssh_t, [],
                        Prd_SoD).
```

Note that constraints are encoded using a 6-tuple, where the last element of the tuple is a function that returns a Coq Boolean. The above example includes the function `Prd_SoD`, defined later in Sect. 3.4.

Policies are defined below as the type 𝕋ℙℒℂ𝕐 using a tuple consisting of a list of rules and a list of constraints. We also define an example policy below using the example rule and example constraint above; in this example both the rule component and the constraint component are lists of length 1.

```
Inductive 𝕋ℙℒℂ𝕐 : Set := TEPolicy : list ℝ * list
    ℂ𝕊𝕋𝔼 → 𝕋ℙℒℂ𝕐.
Definition 𝕋ℙℒℂ𝕐_example : 𝕋ℙℒℂ𝕐 := TEPolicy ([
    ℝ_A], [ ℂ𝕊𝕋𝔼_SoD]).
```

3.2 Semantics in Coq

The Coq definitions of Algorithms 2 and 3 are shown in Listings 1 and 2, respectively.

```
Definition ℝ_EvalTE (ℝ_policy: ℝ) (q: ℚ) : 𝔻ℂ𝕊 :=
 match ℝ_policy with
 |Allow (alw_src 𝕋, alw_dst 𝕋, alw_ℂ, alw_ℙ, alw_𝔹)
     ⇒
   match q with
     |(qsrc 𝕋, qds 𝕋, q ℂ, q ℙ) ⇒
       if ((𝕋 Subset qsrc 𝕋 alw_src 𝕋) && (𝕋
           Subset qds 𝕋 alw_dst 𝕋) &&
          (Nat.eqb q ℂ alw_ℂ) && (Nat.eqb q ℙ alw_
              ℙ) && (alw_𝔹)
       then Permitted else NotPermitted
   end
 |Type_Transition (trn_src 𝕋, trn_dst 𝕋, trn_ℂ) ⇒
     ...
 end.
```

Listing 1 Evaluation of a rule and a query

In Listing 1, the Coq function 𝕋Subset does the subset check, making sure that the types of the subject and object in the query are a subset of the corresponding types in the rule. Note that in both listings, the built-in Coq function `Nat.eqb` is used for the equality checks.

The function ℂ𝕊𝕋𝔼_EvalTE implemented in Listing 2 evaluates a query against a constraint. As mentioned in Sect. 2, the rules argument can be used to extract access information required for expressing security goals encoded in predicates.

```
Definition ℂ𝕊𝕋𝔼_EvalTE
  (constraint_rule: ℂ𝕊𝕋𝔼) (ℚ_to_constr: ℚ) (list ℝ
      :list ℝ) : 𝔻ℂ𝕊 :=
 match constraint_rule with
  |Constraint (cstrn_ℂ, cstrn_ℙ, cstrn_𝕋_arg1,
      cstrn_𝕋_arg2,
          cstrn_list 𝕋, cstrn_ℙℝ𝔻𝕋) ⇒
   match ℚ_to_constr with
    |(ℚ_src 𝕋, ℚ_dst 𝕋, ℚ_ℂ, ℚ_ℙ) ⇒
     if (Nat.eqb ℚ_ℂ cstrn_ℂ && Nat.eqb ℚ_ℙ
         cstrn_ℙ) then
     match (cstrn_ℙℝ𝔻𝕋 list ℝ cstrn_list 𝕋
         cstrn_ℂ cstrn_ℙ
          ℚ_src 𝕋 ℚ_dst 𝕋 cstrn_𝕋_arg1 cstrn_
              𝕋_arg2) with
      |true ⇒ Permitted
      |false ⇒ UnKnown
     end
     else NotPermitted
   end
 end.
```

Listing 2 Evaluation of a constraint

3.3 Processing access information

It is often useful to view various kinds of information in the list of rules as *sets* of values, and so we provide several general operators that support this view, such as *intersection*,

union, as well as *set comparison* operators such as *subset* and *set equality*. TSubset from the previous section was an example of this. Here, we follow the general approach in [14], where it is shown that such operators form a suitable formalism for expressing security conditions and goals formulated as constraints. *Selector functions* process access information in policies by retrieving various kinds of information from a list of rules. In contrast, *operator functions* apply certain operations on the results of selector functions along with other arguments of the predicate. The selector function called Search_subject is defined in Listing 3. The Search_subject function receives a list of rules and an object type as inputs. It searches all the Allow rules of input rules to find all types of subjects that are allowed to access (i.e., perform any kind of action on) objects of the type specified by the object type argument. In particular, the output of Search_subject is a list of elements of type that are authorized to access the object type dscType to perform the action sdaction. The result is a list containing these subject types.

```
Fixpoint Search_subject
(listrule:list ℝ) (dscType: 𝕋) (sdaction: ℙ): list
    𝕋 :=
match listrule with
|Allow (alw_src 𝕋 ,alw_dst 𝕋 ,alw_ ℂ ,alw_ ℙ ,alw_ 𝔹 ) ::
    reclistrule ⇒
  if (𝕋 _Equal alw_ds 𝕋 dscType && Nat.eqb alw_ ℙ
      sdaction) then
    alw_src 𝕋 :: Search_subject reclistrule dsc
        Type sdaction
         else
         Search_subject reclistrule dscType
             sdaction

|Type_Transition (trn_src 𝕋 ,trn_dst 𝕋 ,trn_ ℂ )::
    reclistrule ⇒ ...
end.
```

Listing 3 The selector function Search_subject

Listing 4 expresses the distributive property of the selector function Search_subject. 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 ℝ) (tp: 𝕋) (act: ℙ):
 Search_subject (l1 ++l2) tp act =
     Search_subject (l1) tp act ++Search_subject (
         l2) tp act.
```

Listing 4 The distributive property of the selector function Search_subject

The operator function we develop next is IntersectionList. Listing 5 expresses this function, which returns the set of common elements of two lists of elements of type represented by lstFirst, and lstSecond input arguments. The operator function IntersectionList checks if its two input lists contain any common elements, and the operator is_empty_list checks to see if its input list has no elements.

```
Fixpoint IntersectionList (lstFirst lstSecond:
    list 𝕋 ): list 𝕋 :=
 match lstFirst with
   | (a1) :: l' ⇒
       match lstSecond with
         | l ⇒ if (∃b ( 𝕋 _Equal a1) lstSecond)
                  then (a1) :: IntersectionList l'
                       lstSecond
                  else ...
   | ...
```

Listing 5 The operator function IntersectionList

The operator function IntersectionList uses existsb (denoted by ∃b in code), defined in the Coq library Coq.Lists.List (depicted in Listing 6), to check whether or not an element of the first input list lstFirst is 𝕋_Equal to any elements of the input list lstSecond.

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

Listing 6 The function existsb

Listing 7 depicts two basic properties of the operator function IntersectionList. 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.

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

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

Listing 7 Some properties of the operator function IntersectionList

3.4 Predicates and their conditions

Returning to our example, the Coq implementation of the Prd_SoD predicate exploits *selector* function Search_subject along with the *operator* functions IntersectionList and is_emptylist. In general, *selector* and *operator* functions enable policy developers to extract access information from *Allow* rules that might be useful for applying security goals that are encoded in constraints and predicates. The Coq implementation of Prd_SoD predicate is shown in Listing 8.

```
Fixpoint Prd_SoD(list ℝ:list ℝ)(List 𝕋:list 𝕋)(
    sClass: ℂ)(perm: ℙ)
          (ℚ Src 𝕋 : 𝕋)(ℚ Des 𝕋 : 𝕋)(ℙℝ𝔻𝕋 src 𝕋
              : 𝕋)(ℙℝ𝔻𝕋 Des 𝕋 : 𝕋): 𝔹 :=
 if(𝕋 Subset ℚ Src 𝕋 ℙℝ𝔻𝕋 src 𝕋 && 𝕋 Subset ℚ
     Des 𝕋 ℙℝ𝔻𝕋 Des 𝕋)
 then is_emptylist(IntersectionList
                   (Search_subject list ℝ ℙℝ𝔻𝕋
                       src 𝕋)
                   (Search_subject list ℝ ℙℝ𝔻𝕋
                       Des 𝕋))
 else true.
```

Listing 8 The predicate Prd_SoD

Returning to our example constraint ℂ𝕊𝕋𝔼_SoD in Sect. 3.1, we have now completed the definition of its last component, and thus we can now see how a query is evaluated against this constraint by ℂ𝕊𝕋𝔼_EvalTE in Listing 2. When Prd_SoD is called inside ℂ𝕊𝕋𝔼_EvalTE, it first checks whether or not the predicate is applicable to the query, by checking that the subject and object types of the query (arguments ℚSrc𝕋 and ℚDes𝕋) are subsets of the input arguments ℙℝ𝔻𝕋src𝕋 and ℙℝ𝔻𝕋Des𝕋, respectively. The predicate returns true if this condition is false. When the condition is true, it gathers all the types of subjects in rules that act on objects of types mail_t and/or http_t, and gathers all the types of subjects in rules that act on objects of type networkManager_ssh_t, and ensures that there is no overlap. It checks all rules in a policy, which can be seen by the fact that the first argument to Prd_SoD is passed on directly to both calls to Search_subject.

Recall that in the definition of ℂ𝕊𝕋𝔼, a predicate takes eight arguments. Note that arguments 1 as well as 5–8 are the important ones for expressing Prd_SoD; the fact that the second argument is not used is why an empty list [] appears as the fifth component of ℂ𝕊𝕋𝔼_SoD.

We have used Prd_SoD and some other predicates to develop a security policy called TEpla_policy as a case study, which can be found in the Coq code. This example policy has twenty rules and five constraints. All the predicates used there satisfy the conditions on predicates that we now present in the next section.

As mentioned in Sect. 2.8, policy writers have to verify three conditions on predicates using a library of lemmas we provide for this purpose. We describe the Coq encoding of these conditions briefly here. The first one is about queries (involving the <<= relation) and the other two are about policies (involving the $\lesssim$ relation).

Two of the conditions involve a relation on Boolean values called transition_Verify_Decision that relates the Boolean results of applying a predicate twice with some argument or collection of arguments differing between the two calls.

The first condition is one of the two that uses this relation on Booleans. It is called Predicate_Query_condition and the specific arguments that differ in the two calls are the query subject and object types. This condition is used in a lemma called predicate_query_condition_implication, which simply states that whenever a predicate ℙ satisfies Predicate_Query_condition, then given any two queries $\mathbb{Q}_1$ and $\mathbb{Q}_2$ such that $\mathbb{Q}_1 \ll= \mathbb{Q}_2$, a constraint ℂ whose last argument is ℙ, and any list of rules listℝ, if d_1 and d_2 are the decisions resulting from evaluating $\mathbb{Q}_1$ and $\mathbb{Q}_2$, respectively, against ℂ and listℝ (i.e., applying function ℂ𝕊𝕋𝔼_EvalTE in Listing 2), then $d_1 < ::d_2$.

The second and third conditions involve evaluating a predicate in a constraint on a single query but with two sets of rules (the first argument of the predicate). The second condition simply states that the same result is obtained from applying the predicate on the two lists of rules, whenever the two lists differ only in the order of the rules. This condition is called Predicate_plc_cdn.

The third condition, called Predicate_plc_cdn_Transition, is the other condition that uses the relation transition_Verify_Decision on Booleans. The condition states that given two lists of rules, listℝ and listℝ′, the transition_Verify_Decision relation holds between the results of applying the predicate to listℝ and listℝ ++ listℝ′. This condition and the second condition are used in a lemma called constraintEvalPropSnd.

```
Lemma constraintEvalPropSnd:
 ∀((listRuleA listRuleB: list ℝ)(listCnstrt:list
     ℂ𝕊𝕋𝔼)(q: ℚ)),
 (cnsrt_prd_plcyRulesList(listCnstrt)) →
 (list ℂ𝕊𝕋𝔼_EvalTE listCnstrt q listRuleA) <::
   (list ℂ𝕊𝕋𝔼_EvalTE listCnstrt q(listRuleA++
       listRuleB)) = true.
```

Listing 9 Lemma constraintEvalPropSnd

This lemma states that whenever all the constraints in a given listℂ of constraints satisfy both conditions (expressed by lemma cnsrt_prd_plcyRulesList), then given a query ℚ and two lists of rules listRuleA and listRuleB, if d_1 and d_2 are the decisions resulting from evaluating ℚ against listℂ and the two rule lists listRuleA, and listRuleA ++ listRuleB, respectively, (i.e., applying function listℂ𝕊𝕋𝔼_EvalTE), then $d_1 < ::d_2$.

As mentioned above, predicates have to satisfy the three conditions predicate_query_condition, Predicate_plc_cdn, and Predicate_plc_cdn_Transition.

Three lemmas (qry_condition_SoDpdct, plc_conditionS_SoDpdct, and plc_conditionF_SoDpdct) express that the predicate Prd_SoD satisfies

these three conditions and the proofs are available in the source code [9].

We note here that the expressive power of predicates is limited by the conditions discussed above that they are required to satisfy. Alternatively, however, we propose two methods to extend the expressive power of predicates. The first is simply to relax the restriction and not require these conditions to be verified, which would allow constraints to violate the ordering on decisions by changing an `UnKnown` to a `Permitted`. Allowing this freedom provides policy developers with the same expressive power as the studies that use sets to express security goals, such as [14], which empirically illustrates that practical binary constraints can be expressed by comparisons of two sets. The second method is to replace the above constraints with a structural restriction on policies that requires that the rule component never changes. Such a situation can occur, for example, when different departments in an organization have different security goals, but they all have the same set of rules defined by a central security administrator. With this change, some formal properties we present in Sect. 4 will still hold. This solution eliminates the need for an expert in Coq to verify conditions.

4 Formalization of language properties

In this section, we express the Coq encoding of the main 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 Sect. 2.8. In other words, defining posets ($\mathbb{TEPLCY}$, $\lesssim$), ($\mathbb{Q}$, <<=) and ($\mathbb{DCS}$, < ::) allows us to evaluate language properties. In each subsection, we present the Coq statement of one of the main theorems. In addition, we provide further information about the proofs in Sects. 4.1 and 4.2. In particular, Sect. 4.1 provides information about a common proof technique in Coq, *proof by reflection*, which is used to prove decidability of decisions, and Sect. 4.2 discusses the mathematical proof before presenting the Coq implementation, and also discusses Coq proof strategies.

4.1 Decidability of decisions

Determinism is one of the important properties of policy languages discussed in [2]. A deterministic language always produces the same decision for the same policies and queries. Recall that the function `TEPLCY_EvalTE` evaluates a query against a policy. The behavior of this function specifies the overall semantics of TEpla. Thus, TEpla satisfies determinism simply because evaluation is defined as a function.

Here, we prove the decidability of decisions using proof by reflection [8, 15]. 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 theorems. In particular, Boolean interpretations provide computations and propositional representations provide logical facts. 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* [16] that can be used in a case analysis of inductive types. We prove the decidability of decisions in TEpla through the Boolean reflection approach for partial orders presented by the author of [17].

We define the Boolean relation `compare_decisions` (in listing 10) for comparing decisions by considering the partial order of decisions expressed in Sect. 2.8.

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

Listing 10 The function `compare_decisions` to compare decisions

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 11. As depicted in listing 11, 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 : 𝔻ℂ𝕊 → 𝔻ℂ𝕊 → Prop :=
  | Prp_refl : ∀ d, ans_compr_Prop d d
  | Prp_trans : ∀ dnp dp duk,
              ans_compr_Prop dnp dp →
                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 11 The inductively defined relation `ans_compr_Prop`

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 12).

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

Listing 12 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 theorem `decidability_of_decisions` in listing 13.

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

Listing 13 Decidability of TEpla decisions

The theorem in listing 13 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.

The running time of $\mathbb{TEPLCY}$`_EvalTE` is $O(n + mn)$ in which n and m are the number of rules and constraints, respectively. The running time of checking a query against rules is $O(n)$, and if granted, the running time of checking the query against one constraint is $O(n)$ because, in worst case, to process access information needed in the constraint, the entire list of rules should be traversed. Assuming that the number of rules is greater than constraints, the running time of $\mathbb{TEPLCY}$`_EvalTE` is $O(n^2)$.

4.2 Order preservation of queries

TEpla has in fact been designed so $\mathbb{TEPLCY}$`_EvalTE` is *order-preserving* for the relation $\lesssim$ on policies, $<<=$ on queries, and $<$ :: on decisions. This means that $\mathbb{TEPLCY}$`_EvalTE` acts as a *homomorphism* [13] on the posets we defined on $\mathbb{TEPLCY}$, $\mathbb{Q}$, and $\mathbb{DCS}$.

Of particular importance is the preservation of order on decisions with respect to queries: if $q_1 <<= q_2$, then the decisions d_1 and d_2 that result applying function $\mathbb{TEPLCY}$`_EvalTE` on q_1 and q_2, respectively, are in the relation $d_1 < ::d_2$. The $<<=$ relation is defined (see Sect. 2.8) to be as general as possible; it involves only subject and object types, which are elements that queries in any language must have. 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 [18], is impossible for queries that have incomplete information [2], which is helpful for policy reasoning. In addition, having an unambiguous ordering of queries facilitates sorting, filtering, and optimizing query evaluations.

We first introduce a lemma which helps us to prove order preservation of queries. As mentioned earlier, TEpla policies ($\mathbb{TEPLCY}$) consist of two parts *TErules* and *TEconstraints*. Thus, it is logical to prove this property for the elements of these components: a *TErule* and a *TEconstraint*. The 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 $<<=$, then the decisions of applying Algorithm 2 with same *TErule*, have the relation $<$ ::.

$$\begin{aligned}&\forall\,(q, q' \in \mathbb{Q}), (r \in TErule),\\ &\quad q <<= q' \implies Algorithm\ 2\ (r, q)\\ &\quad < ::Algorithm\ 2\ (r, q').\end{aligned} \tag{1}$$

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 $<<=$, then the decisions of applying Algorithm 3 with the same list of *TErules* and *TEconstraint*, have the relation $<$ ::.

$$\begin{aligned}&\forall\,(q, q' \in \mathbb{Q}), (c \in TEconstraint), (l \in list\ TErule),\\ &\quad q <<= q' \implies Algorithm\ 3\ (c, q, l)\\ &\quad < ::Algorithm\ 3\ (c, q', l).\end{aligned} \tag{2}$$

We now express the order preservation of queries as a theorem for TEpla policies. Proving this theorem requires the two lemmas we mentioned earlier.

$$\begin{aligned}&\forall\,(p \in \mathbb{TEPLCY}), (q, q' \in \mathbb{Q}),\\ &\quad q <<= q' \implies \mathbb{TEPLCY}\texttt{\_EvalTE}\ (p, q)\\ &\quad < ::\mathbb{TEPLCY}\texttt{\_EvalTE}\ (p, q').\end{aligned}$$

As the theorem states, for the same policies, queries with the relation $<<=$ render decisions with the relation $<$ ::. The Coq version of this theorem is stated below as Theorem `Order_Preservation_TEpla` in Listing 14.

```
Theorem Order_Preservation_TEpla:
 ∀(listrule:list ℝ)(listconstraint:list ℂSTE)(q
     q' : ℚ),
   (q <<= q') ∧ const_imp_prd_List list ℂSTE →
   (( TEPLCY _EvalTE ( TEPLCY (listrule,
       listconstraint)) q) <::
    ( TEPLCY _EvalTE ( TEPLCY (listrule,
       listconstraint)) q')) = true.
```

Listing 14 Order preservation of decisions with respect to queries

The `const_imp_prd_List` predicate in this theorem expresses that all the predicates of the input list of constraints `list`$\mathbb{CSTE}$ satisfy the first condition from Sect. 3.4 (`predicate_query_condition`).

The proof script for theorem `Order_Preservation_TEpla` is shown in Fig. 2.

In the Coq proof, *tactics* are used to decompose the proof into subgoals, or to transform a subgoal to a simpler one until the proof is complete. This proof is by induction on the length of the list of constraints and the `induction` tactic on the first line of the proof breaks the proof into these two cases. The symbols "-," "+," and "*" mark the cases of proofs

Fig. 2 The proof script of Theorem `Order_Preservation_TEpla`

```
Theorem Order_Preservation_TEpla :
  ∀ (listrule:list ℝ) (listCnstrt:list ℂ𝕊𝕋𝔼) (q q' : ℚ),
     (q <<= q') ∧ const_imp_prd_List listℂ𝕊𝕋𝔼 →
     ((𝕋𝔼ℙ𝕃ℂ𝕐_EvalTE (𝕋𝔼ℙ𝕃ℂ𝕐 (listrule, listCnstrt)) q) <::
      (𝕋𝔼ℙ𝕃ℂ𝕐_EvalTE (𝕋𝔼ℙ𝕃ℂ𝕐 (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(𝕋𝔼ℙ𝕃ℂ𝕐_EvalTE (𝕋𝔼ℙ𝕃ℂ𝕐 (listRule, a :: listCnstrt)) q' =
        (maximalDcs  (𝕋𝔼ℙ𝕃ℂ𝕐_EvalTE (𝕋𝔼ℙ𝕃ℂ𝕐 (listRule, [a])) q')
        (𝕋𝔼ℙ𝕃ℂ𝕐_EvalTE (𝕋𝔼ℙ𝕃ℂ𝕐 (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.
```

that correspond to each subgoal on different levels. Here, the subproof starting with the first "-" is the base case of the induction, when the list is empty. The subproof starting with the second "-" is for the inductive case. The application of the `induction` tactic generates the induction hypothesis for this case.

In general, theorems and lemmas in Coq are 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 subgoals (G). Here, we explain a few of the other tactics used in the proof in Fig. 2. 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` .... 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 `assert` 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 `assert (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 `apply` tactic can be used to apply previously proved lemmas. Note that the proof in Fig. 2 uses many such lemmas. For example, they include `validconst_prop`, `maximalDcs_prop`, and `max_appendAns4`, shown in listings 15, 16 and 17, respectively. The binary function `maximalDcs`, used in listings 16 and 17, takes two input decisions and returns the greater decision according to the order of decisions defined in Sect. 2.8.

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

Listing 15 Lemma validconst_prop

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

Listing 16 Lemma maximalDcs_prop

```
Lemma max_appendAns4:
 ∀ (a:list R ) (h: CSTE )(l:list CSTE )
 (q: Q ),
 ( TEPLCY _EvalTE (TEPolicy (a, h :: l)) q) =
      maximalDcs
           ( TEPLCY _EvalTE (TEPolicy (a, [h])) q)
           ( TEPLCY _EvalTE (TEPolicy (a, l)) q).
```

Listing 17 Lemma max_appendAns4

In addition, the OrdPreserv_cstrt lemma is the Coq version of Lemma (2), and the max_prop2 lemma is a corollary of Lemma (1) above.

4.3 Non-decreasing property of policies

It is common to add new policy statements as new regulations arise. The next property states that when adding new rules, policies do not change their decisions in the reverse direction of the order on decisions (i.e., $<::$). When adding new rules, changing decisions, for example, from *Permitted* to *NotPermitted*, is impossible. Thus, granted requests will never be revoked. 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 property is aligned with *monotonicity* defined in [2]. We state and prove the property in Listing 18, which expresses that TEpla is *non-decreasing*.

```
Theorem Non_Decreasing_TEpla:
∀ (Pol_list: list TEPLCY ) (Single_pol: TEPLCY ) (q:
     Q ) (d d': DCS ),
 validCnstrtListPolicy Pol_list ∧ validConstrt
      Single_pol →
  ( TEPLCY _EvalTE (⊕ (Pol_list)) q) = d →
  ( TEPLCY _EvalTE (⊕ (Single_pol::Pol_list)) q) = d'
       →
  (d <:: d') = true.
```

Listing 18 Theorem Non_Decreasing_TEpla

This theorem states that adding a policy Single_pol, to any list of policies Pol_list can change the decisions only according to the order relation $<:$: on decisions. The predicate validCnstrt expresses that the constraints in Single_pol satisfy the second and third conditions from Sect. 3.4 (Predicate_pl_cdn and Predicate_plc_cdn_Transition). The predicate validCnstrtListPolicy applies this check to every policy in Pol_list. The $\oplus$ operator extracts the rule lists of all the policies in its argument list of policies and combines them into one list, and similarly for constraints, forming a single policy from these rules and constraints. Note that in this theorem, $(\oplus$ Pol_list$) \lesssim (\oplus$ (Single_pol::Pol_list)).

4.4 Independent composition of policies

It is important to be able to analyze the behavior of access control policies based on their components or *sub-policies*, as the decisions for the combined policies can be determined from the decisions of included policies. Similar to *independent composition* in [2], we codify the following property of TEpla.

```
Theorem Independent_Composition:
∀ (PLCY_ DCS _pair : list ( TEPLCY * DCS )) (q : Q ) (
     dstar : DCS ),
 Foreach q (map fst PLCY_ DCS _pair) (map snd PLCY_
      DCS _pair) ∧
 ( TEPLCY _EvalTE (⊕ (map fst PLCY_ DCS _pair)) q) =
      dstar →
 (maximum (map snd PLCY_ DCS _pair) <:: dstar) = true.
```

Listing 19 Theorem Independent_Composition

In this statement, PLCY_DCS_pair is a list of policies and a list of decisions of the same length such that for each index i into these lists, if p_i and d_i are the policy and decision at this index, respectively, then (p_i, d_i) is an *evaluation pair on* q, which means that (TEPLCY_EvalTE p_i q) $= d_i$, i.e., that d_i is the decision returned from evaluating policy p_i on query q. Although we do not show its definition, the Foreach predicate is defined to express this property. It also expresses that all the constraints in each policy satisfy the second and third conditions from Sect. 3.4 (Predicate_pl_cdn and Predicate_plc_cdn_Transition). The independent composition theorem states that whenever a pair of lists satisfies this property, then the decision obtained by evaluating the combined policy on q is the maximum of the decisions resulting from evaluating each policy independently. The function maximum takes a list of decisions and returns the maximum according to the binary relation $<::$.

5 Conclusion

We have presented the infrastructure of the TEpla Type Enforcement policy language, and formally verified some of its important properties in Coq. TEpla, with formal semantics and verified properties, is an essential step toward developing

certifiably correct policy-related tools for Type Enforcement policies.

The properties that we have considered here, namely *determinism*, *order preservation*, *independent composition*, and *non-decreasing*, analyze 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.

Moreover, we provide the language constructs (in particular, the integration of user-defined predicates) 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.

In related work, ACCPL (A Certified Core Policy Language) [19] represents some preliminary work using our approach, i.e., building in formal semantics from the start, but in the domain of web services and digital resources, with some very basic properties proved, which include determinism, but not the other properties considered here. In other work, a variety of other studies have included the formalization of various aspects of access control policies using different and sometimes quite complex logics and algorithms, e.g., [20–23]. In our approach, we start with a simple language, and some simple notions of orderings and relations on sets, and show that it is possible to express fairly complex access control requirements. We were inspired, for example, by the work in [14], which shows that complex access control constraints such as separation of duty [4, 24] can be expressed using set operators. Additionally, although we began with the particular domain of policies for operating systems, one of our goals is to develop general ideas that can be adapted to other domains such as the web and distributed platforms. Future work will include exploring such extensions. Eventually, we plan to 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 [25] for firewall policy evaluation.

With regard to work on SELinux in particular, different studies have been carried out that put forward some possible tools for helping policy writers write policies that are more easily understood and reasoned about. Languages such as Lobster [26], Seng [27], Please [28], and CDS-Framework [29] are intended 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, as analyzed in [5], 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.

Our future work will also include addressing some of the current limitations of the language, including extending the kinds of constraints provided, as well as designing and developing certified tools for policy-related tasks such as automating various kind of policy analyses. We expect to be able to reuse many definitions and lemmas of the current Coq development.

Author Contributions AE contributed to this work as part of his doctoral thesis under the supervision of Dr. AF. As the primary author, he conceptualized the research idea, designed the study, carried out the verification, and wrote the initial draft of the manuscript. AF served as the thesis supervisor and provided significant guidance throughout the process. She contributed to refining the research idea, provided critical feedback on the study design. She also played a significant role in revising and finalizing the manuscript for publication, providing critical intellectual input. Both authors have contributed equally to this work.

Funding Not applicable.

Availability of data and materials Not applicable.

Declarations

Conflict of interest The authors declare that they have no competing interests, financial or personal, that could have appeared to influence the work reported in this paper.

Ethical approval Not applicable.

Appendix A The BNF grammar of TEpla

Figure 3 represents the BNF grammar of TEpla. 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.

Fig. 3 The BNF grammar of TEpla

```
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 | ···                           ; 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
```

References

1. Chlipala A (2019) Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant. The MIT Press, Cambridge
2. Tschantz MC, Krishnamurthi S (2006) Towards reasonability properties for access-control policy languages. In: 11th ACM symposium on access control models and technologies (SACMAT), pp. 160–169
3. National Security Agency: Security-Enhanced Linux. https://github.com/SELinuxProject (2022)
4. Stallings W, Brown L (2018) Computer security, principles and practices. Pearson Education, Hoboken
5. Eaman A, Sistany B, 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), pp. 116–135
6. Jang M, Messier R (2015) Security strategies in Linux platforms and applications, 2nd edn. Jones and Bartlett Publishers Inc., Burlington
7. Bertot Y, Castéran P (2004) Interactive theorem proving and program development. Coq'Art: the calculus of inductive constructions. Springer, Berlin
8. The Coq Development Team: The Coq Reference Manual: Release. INRIA (2020). INRIA. https://coq.inria.fr/refman/
9. Eaman A (2022) The Coq development of TEpla. https://www.site.uottawa.ca/~afelty/isse22/
10. Eaman A, Felty A (2020) Formal verification of a certified policy language. In: 14th international conference on verification and evaluation of computer and communication systems (VECoS), pp. 180–194

11. Eaman A (2019) TEpla: a certified type enforcement access control policy language. Ph.D. thesis, University of Ottawa. https://ruor.uottawa.ca/handle/10393/39876
12. Mayer F, Caplan D, MacMillan K (2006) SELinux by example: using security enhanced Linux. Prentice Hall, Hoboken
13. Harzheim E (2005) Ordered sets. Springer, Boston
14. Jaeger T, Tidswell J (2001) Practical safety in flexible access control models. ACM Trans Inf Syst Secur (TISSEC) 4:158–190
15. Grégoire B, Tassi E (2016) Boolean reflection via type classes. In: Coq workshop, Nancy, France. HAL Id: hal-01410530
16. Gonthier G, Mahboubi A (2010) An introduction to small scale reflection in Coq. J Formaliz Reason ASDD-AlmaDL 3(2):95–152
17. Azevedo de Amorim A (2016) Binding operators for nominal sets. Electron Notes Theor Comput Sci 325:3–27
18. Parrend P, 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), pp 80–96. Springer, Germany
19. Sistany B, Felty A (2017) A certified core policy language. In: 15th annual international conference on privacy, security and trust (PST), pp 391–393
20. Brucker AD, Brügger L, Wolff B (2014) The unified policy framework (UPF). Archive of formal proofs. https://www.isa-afp.org/entries/UPF.html
21. Abadi M, Burrows M, Lampson B, Plotkin G (1992) A calculus for access control in distributed systems. In: Advances in cryptology (CRYPTO 1991), vol 576, pp 1–23. Springer
22. Wu C, Zhang X, Urban C (2013) A formal model and correctness proof for an access control policy framework. In: Gonthier G, Norrish M (eds) Certified programs and proofs (CPP), vol 8307. Springer, Melbourne, pp 292–307
23. Archer M, Leonard EI, Pradella M (2003) Analyzing security-enhanced Linux policy specifications. In: Proceedings POLICY 2003. IEEE 4th international workshop on policies for distributed systems and networks, pp 158–169
24. Jaeger T, Zhang X, Edwards A (2003) Policy management using access control spaces. ACM Trans Inf Syst Secur (TISSEC) 6(3):327–364
25. Capretta V, Stepien B, Felty A, Matwin S (2007) Formal correctness of conflict detection for firewalls. In: ACM workshop on formal methods in security engineering (FMSE), pp 22–30
26. Hurd J, Carlsson M, Finne S, Letner B, Stanley J, White P (2009) Policy DSL: high-level specifications of information flows for security policies. In: High confidence software and systems (HCSS)
27. Kuliniewicz P (2006) SENG: an enhanced policy language for SELinux presented at security enhanced Linux symposium. http://selinuxsymposium.org/2006/papers/09-SENG.pdf
28. Quigley DP (2007) PLEASE: policy language for easy administration of SELinux. Master's thesis, Stony Brook University
29. Sellers C, Athey J, Shimko S, Mayer F, MacMillan K, Wilson A (2006) Experiences implementing a higher-level policy language for SELinux presented at security enhanced Linux symposium. http://selinuxsymposium.org/2006/papers/08-higher-level-experience.pdf