Formal Verification of a Certified Policy Language

Amir Eaman(✉) and Amy Felty

School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, Canada
{aeama028,afelty}@uottawa.ca

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. 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 [6]. 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 semantics as it satisfies important formal properties designed for this purpose [22]. 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.

In order to keep the core of the language simple, in this study, we focus on developing a new certified policy language for the *Type Enforcement* mechanism, which is a subset of the SELinux security module [16] 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) [20] 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 [8] 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 [13] 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 [3,21] (version 8.12) to write machine-checked mathematical proofs for TEpla's properties. The Coq development of TEpla contains approximately 4700 lines of script and is available at http://www.site.uottawa.ca/~afelty/vecos20/. This online appendix also contains a mapping from names used in this paper to names used in the Coq code.

In Sect. 2, we present most of the infrastructure of TEpla, including rules, decisions, queries, and policies, and present the part of the semantics that involves evaluating queries against the rules. This section also defines ordering relations on TEpla decisions, policies, and queries. In Sect. 3, we present the syntax and semantics of constraints. A constraint can be considered as an additional form of a policy rule, which takes a user-defined predicate as an argument. We present the syntax and semantics of constraints, and discuss the conditions that must hold for predicates. In Sect. 4, we discuss the main properties that we have proved about TEpla, and Sect. 5 concludes the paper.

The work presented here appears in the Ph.D. thesis of the first author [7], and the reader is referred there for details, including a BNF grammar of TEpla's syntax. Here, in Sects. 2 and 3, we informally describe the TEpla language structures and their meaning, and present parts of the Coq encoding to illustrate.

2 Rules, Decisions, Queries, and Policies

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. 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. Basic and group types together form the notion of a *type*, which is the main building block of TEpla.

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. 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 [15].

2.1 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 [21]. These definitions plus some examples are below.

```
Definition ℂ := ℕ.    Definition ℙ := ℕ.    Definition basic𝕋 := ℕ.
Definition File : ℂ := 600.
Definition mail_t : basic𝕋 := 300.   Definition http_t : basic𝕋 := 301.
Definition networkManager_ssh_t : basic𝕋 := 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 basic𝕋.
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:=
  | single𝕋 : basic𝕋 → 𝕋
  | group𝕋 : 𝔾 → 𝕋.
```

As an 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.

The access control rules that are used to form policies are defined inductively as type ℝ. These rules consist of *Allow* and *Type Transition* rules. The definition of ℝ below is followed by an example *Allow* rule.

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

Rules are implemented using tuples. *Allow* rules enable policy writers to express eligible access from subjects (whose type is expressed by the first component) to objects (whose type is expressed by the second component). The third component specifies the object class of the object. The fourth component expresses possible actions that the object can perform on the subject. The last component is a Boolean condition, which we do not use here; it is discussed in future work.

The second kind of rule provides support for transition of types in security contexts from one value to another. *Type Transition* rules in policies express which types can switch to other types, which is an important feature that we adapt from SELinux but do not discuss further in this paper.

TEpla has a three-valued decision set for access requests including *NotPermitted*, *Permitted* and *UnKnown*. In TEpla, queries are denied by default, i.e., every access request that should be granted must be expressed explicitly by rules in a policy. Decisions are defined by the inductive type $\mathbb{DCS}$:

```
Inductive 𝔻ℂ𝕊: Set := Permitted | NotPermitted | UnKnown.
```

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. For example, a specific *Allow* rule may permit a particular query, but there is a constraint in the policy that is not satisfied by the query. Such conflicts signify errors that must be corrected by a policy administrator.

Access requests or *queries* are inquiries into the policy to check the possibility that the subject is allowed to perform the specified action on the object. In TEpla, they consist of four components: the types of the subject and object, the object class of the object, and an action. The definition of queries ($\mathbb{Q}$) is shown below, along with an example.

```
Definition ℚ : Set := 𝕋 * 𝕋 * ℂ * ℙ.
Definition sampleℚ : ℚ := (single𝕋 mail_t, single𝕋 http_t, File, Write).
```

Processing of a query with respect to a policy involves an attempt to check the authorization of a subject with the given type to carry out a specific action on an object having the given type and class. In the example, a subject of type `mail_t` is requesting to write to an object whose class is `File` and whose type is `http_t`.

Policies, defined below as the type $\mathbb{TEPLCY}$, consist of a sequence of rules and a sequence of constraints. Constraints, denoted as $\mathbb{CSTE}$, will be defined in the next section.

```
Inductive 𝕋𝔼ℙ𝕃ℂ𝕐: Set := TEPolicy : list ℝ * list ℂ𝕊𝕋𝔼 → 𝕋𝔼ℙ𝕃ℂ𝕐.
```

2.2 Evaluating Queries Against Policy Rules

We define the semantics of TEpla as a mapping from policies and access requests to decisions, in the form of five translation functions implemented in Coq, 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 Listing 1, evaluates a query against a single rule leading to a decision of either `Permitted` or `NotPermitted`.

```
Definition ℝ_EvalTE (ℝ_policy:ℝ) (q:ℚ) : 𝔻ℂ𝕊:=
 match ℝ_policy with
 |Allow (alw_src𝕋,alw_dst𝕋,alw_ℂ,alw_ℙ,alw_𝔹) ⇒
     match q with
         |(qsrc𝕋, qds𝕋, qℂ, qℙ) ⇒
```

```
            if  ((𝕋Subset qsrc𝕋 alw_src𝕋) && (𝕋Subset qds𝕋 alw_dst𝕋) &&
                  (Nat.eqb qℂ alw_ℂ) && (Nat.eqb qℙ alw_ℙ) && (alw_𝔹)
            then Permitted else NotPermitted
      end
 |Type_Transition (trn_src𝕋, trn_dst𝕋, trn_ℂ) ⇒ ...
 end.
```

Listing 1. Evaluation of a rule and a query

For *Allow* rules, the first four conditions of the if statement check to see if the rule applies to the query. The first two check that the types of the subject and object in the query are a subset of the corresponding types in the rule. (𝕋Subset performs this check.) The next two conditions check that the object class and permitted action are the same, using the built-in function `Nat.eqb`. The last condition checks that the last component of rule, which is a Boolean condition, evaluates to *true*. If all conditions are satisfied, the result is `Permitted`. A *Type Transition* rule 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 function for evaluating queries against constraints is presented in Sect. 3; we also discuss the other three functions there.

2.3 Ordering Relation on Decisions, Queries and Policies

We define a *Partially Ordered Set (poset)* [9] 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*. 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 for more details about this property).

Additionally, we define a relation on queries ($\mathbb{Q}$, <<=). Two queries $\mathbb{Q}_1$ = (Source$\mathbb{T}$_$\mathbb{Q}_1$, Dest$\mathbb{T}$_$\mathbb{Q}_1$, $\mathbb{C}_1$, $\mathbb{P}_1$) and $\mathbb{Q}_2$ = (Source$\mathbb{T}$_$\mathbb{Q}_2$, Dest$\mathbb{T}$_$\mathbb{Q}_2$, $\mathbb{C}_2$, $\mathbb{P}_2$) are in relation $\mathbb{Q}_1$ <<= $\mathbb{Q}_2$ if and only if ($\mathbb{T}$`Subset` Source$\mathbb{T}$_$\mathbb{Q}_2$ Source$\mathbb{T}$_$\mathbb{Q}_1$) and ($\mathbb{T}$`Subset` Dest$\mathbb{T}$_$\mathbb{Q}_2$ Dest$\mathbb{T}$_$\mathbb{Q}_1$) 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:

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

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 $p_1 \subseteq p_2$ means that p_2 has more authorization rules and it contains all the authorization rules in p_1.[1]

3 Constraints and Predicates

As discussed earlier, rules alone cannot always accommodate the security requirements of systems precisely enough. TEpla's *constraints* and *predicates*, described in this section, represent one of the powerful features of TEpla, which distinguish it from other languages that lack this feature. TEpla constraints allow policy writers not only to rely on conditions or constraints defined in the language but also to define their complementary security logic.

3.1 Syntax in Coq

Constraints are defined below as the type ℂ𝕊𝕋𝔼.

```
Inductive ℂ𝕊𝕋𝔼: Set :=
  | Constraint : ℂ ∗ ℙ ∗ 𝕋 ∗ 𝕋 ∗ list 𝕋 ∗
        (list ℝ → list 𝕋 → ℂ → ℙ → 𝕋 → 𝕋 → 𝕋 → 𝕋 → 𝔹) → ℂ𝕊𝕋𝔼.
```

Constraints have six arguments. When a constraint is checked against a query or access request, the values of the first two arguments are compared to the values of the ℂ and ℙ components of a query, and the constraint is only applicable when the values of these components match. A constraint takes a function as its last argument, which returns a Boolean; these functions act as predicates that express when the constraint is satisfied. To express specific security goals, administrators can define different predicates by using various arguments provided for the function. These arguments supply a comprehensive set of values by which policy developers can define the required security criteria. To illustrate constraints and predicates, we use a "separation of duty" running example, which includes the constraint ℂ𝕊𝕋𝔼_SoD, defined below, and the predicate Prd_SoD, defined later.

```
Definition ℂ𝕊𝕋𝔼_SoD : ℂ𝕊𝕋𝔼:= Constraint(File, Read, group𝕋 program_𝔾,
                                 single𝕋 networkManager_ssh_t ,  [],  Prd_SoD ).
```

This constraint only allows subjects whose types are elements of program_𝔾 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 program_𝔾 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, and it will be defined after presenting the implementation of the function for evaluating constraints against queries. 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 `program_𝔾`, then the object in the other rule cannot have type `networkManager_ssh_t`. Similarly, if the object in

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

one rule has type `networkManager_ssh_t`, then the object in the other rule cannot have a type that is a subset of `program_𝔾`. The constraint `ℂ𝕊𝕋𝔼_SoD` is applicable to all queries whose ℂ and ℙ are `File` and `Read`, respectively.

Using the above example constraint, and the example rule in Sect. 2.1, we can define the example policy below, where both the rule component and the constraint component are lists of length 1.

```
Definition 𝕋𝔼ℙ𝕃ℂ𝕐_example : 𝕋𝔼ℙ𝕃ℂ𝕐:= TEPolicy ([ℝ_A], [ℂ𝕊𝕋𝔼_SoD]).
```

3.2 Evaluating Queries Against Constraints

The function `ℂ𝕊𝕋𝔼_EvalTE` implemented in Listing 2 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.

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

Listing 2. Evaluation of a constraint

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 `cstrn_ℙℝ𝔻𝕋` 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 is not applicable, the default value `NotPermitted` is returned.

A query must be evaluated against all the rules and constraints in a policy. We omit the other three functions that are defined to complete this task, and just remark that the main function that calls the others is called `𝕋𝔼ℙ𝕃ℂ𝕐_EvalTE`. We note that they are implemented so that `ℂ𝕊𝕋𝔼_EvalTE` is always passed the complete list of rules in a policy as its third argument, and thus the complete list of rules is always passed as input (as the first argument) when the constraint predicate is called inside ℂ𝕊𝕋𝔼_EvalTE.

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*. Here, we follow the general approach in [11], where it is shown that such operators form a suitable formalism for expressing security conditions and goals formulated as constraints. Those that are useful for our separation of duty example include *selector functions*, which retrieve various kinds of information from a list of rules, and *operator functions*, which apply certain operations on the results of selector functions along with other arguments of the predicate. We include a selector function called listℝSearch_subject𝕋s, which receives a list of rules and an object type as inputs and returns a list of types, which we view here as a set. This function 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. The result is a list (set) containing these subject types. The definition of Prd_SoD uses this function along with operator functions called IntersectionList, which returns the set of common elements of two lists, and is_emptylist𝕋, which checks whether or not a list of types is empty. The Prd_SoD predicate is defined in Listing 3.

```
Fixpoint Prd_SoD (listℝ:list ℝ) (List𝕋:list 𝕋) (sClass:ℂ) (perm:ℙ)
                 (ℚSrc𝕋:𝕋) (ℚDes𝕋:𝕋) (ℙℝ𝔻𝕋src𝕋:𝕋) (ℙℝ𝔻𝕋Des𝕋:𝕋) : 𝔹:=
  if (𝕋Subset ℚSrc𝕋 ℙℝ𝔻𝕋src𝕋 && 𝕋Subset ℚDes𝕋 ℙℝ𝔻𝕋Des𝕋)
  then is_emptylist𝕋 (IntersectionList
                           (listℝSearch_subject𝕋s listℝ ℙℝ𝔻𝕋src𝕋)
                           (listℝSearch_subject𝕋s listℝ ℙℝ𝔻𝕋Des𝕋))
  else true.
```

Listing 3. 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 listℝSearch_subject𝕋s.

Recall that in the definition of ℂ𝕊𝕋𝔼, a predicate takes eight arguments, but there is of course no requirement that the predicate uses them all. In Prd_SoD, note that the second, third, and fourth arguments are not relevant for expressing separation of duty. 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.

3.3 Conditions on Predicates

Policy writers have to verify three conditions on predicates using a library of lemmas we provide for this purpose. The conditions express that given two queries related by $<<=$ or two policies related by $\lesssim$, the evaluation of a query against a policy preserves the defined order on decisions $<::$. We describe them briefly here.[2] 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 $\mathbb{P}$ satisfies `predicate_query_condition`, then given any two queries $\mathbb{Q}_1$ and $\mathbb{Q}_2$ such that $\mathbb{Q}_1 <<= \mathbb{Q}_2$, a constraint $\mathbb{C}$ whose last argument is $\mathbb{P}$, and any list of rules list$\mathbb{R}$, if d_1 and d_2 are the decisions resulting from evaluating $\mathbb{Q}_1$ and $\mathbb{Q}_2$, respectively, against $\mathbb{C}$ and list$\mathbb{R}$ (i.e., applying function $\mathbb{CSTE}$`_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, list$\mathbb{R}$ and list$\mathbb{R}'$, the `transition_Verify_Decision` relation holds between the results of applying the predicate to list$\mathbb{R}$ and list$\mathbb{R}$ ++ list$\mathbb{R}'$. This condition and the second condition are used in a lemma called `constraintEvalPropSnd`. This lemma states that whenever all the constraints in a given list$\mathbb{C}$ of constraints satisfy both conditions, then given a query $\mathbb{Q}$ and two lists of rules list$\mathbb{R}$ and list$\mathbb{R}'$, if d_1 and d_2 are the decisions resulting from evaluating $\mathbb{Q}$ against list$\mathbb{C}$ and the two rule lists list$\mathbb{R}$, and list$\mathbb{R}$ ++ list$\mathbb{R}'$, respectively, (i.e., applying function list$\mathbb{CSTE}$`_EvalTE`), then $d_1 <:: d_2$.

[2] The lemmas stating that these three conditions hold for our running example `Prd_SoD` are called `qry_condition_SoDpredicate`, `plc_conditionS_SoDpredicate`, and `plc_conditionF_SoDpredicate` in the Coq code.

The expressive power of predicates is limited by the conditions that they have to satisfy, as presented in this section. 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 [11], 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 Properties of TEpla and Their Formalization

Determinism is one of the important properties of policy languages discussed in [22]. A deterministic language always produces the same decision for the same policies and queries. Recall that the function $\mathbb{TEPLCY}$_`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.

4.1 Order Preservation of TEpla 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* [9] 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. 2.3) 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. Having an unambiguous ordering facilitates sorting, filtering, and optimizing query evaluation by administrators. This property can be compared to the *safety* property defined in [22].

Theorem `Order_Preservation_TEpla` (in Listing 4) expresses this order preservation on queries.

```
Theorem Order_Preservation_TEpla :
  ∀ (listℝ:list ℝ) (listℂ𝕊𝕋𝔼:list ℂ𝕊𝕋𝔼) (q q' : ℚ),
      (q <<= q') ∧ const_imp_prd_List listℂ𝕊𝕋𝔼 →
```

```
((𝕋𝔼ℙ𝕃ℂ𝕐_EvalTE (𝕋𝔼ℙ𝕃ℂ𝕐 (listℝ, listℂ𝕊𝕋𝔼)) q) <::
 (𝕋𝔼ℙ𝕃ℂ𝕐_EvalTE (𝕋𝔼ℙ𝕃ℂ𝕐 (listℝ, listℂ𝕊𝕋𝔼)) q')) = true.
```

Listing 4. 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 `listℂ𝕊𝕋𝔼` satisfy the first condition from Sect. 3.3 (`predicate_query_condition`).

4.2 Non-decreasing Property of TEpla 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 [22]. We state and prove the property in Listing 5, 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 5. 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. 3.3 (`Predicate_pl_cdn` and `Predicate_plc_cdn_Transition`). The predicate `validCnstrtListPolicy` applies this check to every policy in `Pol_list`. The ⊕ 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`)).

4.3 Independent Composition of TEpla 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 [22], 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 6. 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. 3.3 (`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.., [1,2,4,23]. 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 [11], which shows that complex access control constraints such as separation of duty [12, 20] 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 [5] 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 [10], Seng [14], Please [17], and CDSFramework [18] 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 [8], 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.

References

1. Abadi, M., Burrows, M., Lampson, B., Plotkin, G.: A calculus for access control in distributed systems. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 1–23. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-46766-1_1
2. Archer, M., Leonard, E.I., Pradella, M.: Analyzing security-enhanced Linux policy specifications. In: Proceedings POLICY 2003, IEEE 4th International Workshop on Policies for Distributed Systems and Networks, pp. 158–169 (2003)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-662-07964-5
4. Brucker, A.D., Brügger, L., Wolff, B.: The unified policy framework (UPF). Archive of Formal Proofs (2014). https://www.isa-afp.org/entries/UPF.html
5. Capretta, V., Stepien, B., Felty, A., Matwin, S.: Formal correctness of conflict detection for firewalls. In: ACM Workshop on Formal Methods in Security Engineering (FMSE), pp. 22–30 (2007)
6. Chlipala, A.: Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant. The MIT Press, Cambridge (2019). https://mitpress.mit.edu/books/certified-programming-dependent-types

7. Eaman, A.: TEpla: a certified type enforcement access control policy language. Ph.D. thesis, University of Ottawa (2019). https://ruor.uottawa.ca/handle/10393/39876
8. Eaman, A., Sistany, B., Felty, A.: 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 (2017)
9. Harzheim, E.: Ordered Sets. Springer, Boston (2005). https://doi.org/10.1007/b104891
10. Hurd, J., Carlsson, M., Finne, S., Letner, B., Stanley, J., White, P.: Policy DSL: high-level specifications of information flows for security policies. In: High Confidence Software and Systems (HCSS) (2009)
11. Jaeger, T., Tidswell, J.: Practical safety in flexible access control models. ACM Trans. Inf. Syst. Secur. (TISSEC) **4**, 158–190 (2001)
12. Jaeger, T., Zhang, X., Edwards, A.: Policy management using access control spaces. ACM Trans. Inf. Syst. Secur. (TISSEC) **6**(3), 327–364 (2003)
13. Jang, M., Messier, R.: Security Strategies in Linux Platforms and Applications, 2nd edn. Jones and Bartlett Publishers Inc., Burlington (2015)
14. Kuliniewicz, P.: SENG: an enhanced policy language for SELinux (2006). Presented at Security Enhanced Linux Symposium. http://selinuxsymposium.org/2006/papers/09-SENG.pdf
15. Mayer, F., Caplan, D., MacMillan, K.: SELinux by Example: Using Security Enhanced Linux. Prentice Hall, Upper Saddle River (2006)
16. National Security Agency: Security-Enhanced Linux (2019). https://www.nsa.gov/what-we-do/research/selinux/
17. Quigley, D.P.: PLEASE: policy language for easy administration of SELinux. Master's thesis, Stony Brook University (2007). Technical report FSL-07-02. www.fsl.cs.sunysb.edu/docs/dquigley-msthesis/dquigley-msthesis.pdf
18. Sellers, C., Athey, J., Shimko, S., Mayer, F., MacMillan, K., Wilson, A.: Experiences implementing a higher-level policy language for SELinux (2006). Presented at Security Enhanced Linux Symposium. http://selinuxsymposium.org/2006/papers/08-higher-level-experience.pdf
19. Sistany, B., Felty, A.: A certified core policy language. In: 15th Annual International Conference on Privacy, Security and Trust (PST), pp. 391–393 (2017)
20. Stallings, W., Brown, L.: Computer Security, Principles and Practices. Pearson Education, London (2018)
21. The Coq Development Team: The Coq Reference Manual: Release. INRIA (2020). https://coq.inria.fr/refman/
22. Tschantz, M.C., Krishnamurthi, S.: Towards reasonability properties for access-control policy languages. In: 11th ACM Symposium on Access Control Models and Technologies (SACMAT), pp. 160–169 (2006)
23. Wu, C., Zhang, X., Urban, C.: A formal model and correctness proof for an access control policy framework. In: Gonthier, G., Norrish, M. (eds.) CPP 2013. LNCS, vol. 8307, pp. 292–307. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03545-1_19