Formal Verification of a Certified Policy Language

Amir Eaman
Amy Felty

School of Electrical Engineering and Computer Science (SITE)
University of Ottawa
Ottawa, Canada

Amir 

Amy 

Introduction

Access Control is a security service that guards protected resources against unauthorized access (Granting or rejecting access to resources is a very important aspect of computer systems).

Policy Language is a language to express authorization policies (in the form of set of rules)

Type Enforcement An access control mechanism which exploits security context of resources to regulate accesses (subject-object set of rules).

Certified Language By certified policy language, we mean a policy language with formal semantics and mathematical proofs of important properties.

TEpla a new certified TE access control policy language called *TEpla*.

Access control policy languages, especially those that are widely used in practice, often do not have formal semantics. Inconsistencies and contradictions in the design of a language can lead to possibly serious unintended errors, especially as policies grow large.

Motivation: Analysis of Access-Control SELinux Policies

The motivation comes from our previous work [Eaman, Sistany, Felty, MCETech, 2017]

- For having a verified security policy, it is crucial to formally reason about the policy language in which the policy is written.
- The SELinux policy language requires third-party analysis tools to help security administrators write policies and check various properties.
- The inherent complexity of the SELinux policy language as well as its lack of formal semantics have led to the development of many policy analysis tools to try to translate SELinux policies to other intermediate language.
- There is no proof for the correctness of policy analysis tools to make sure their results are reliable.
- Overall SELinux lacks clarity as an access control language. The clarity of an access control policy language can provide better decision making for incremental policy writing, ease of analysis, and ease of reasoning.

Using a Certified TE Policy Language to Express TE policies

- Our solution to the challenges mentioned earlier is a certified Type Enforcement policy Language
- A small and certifiably correct Type Enforcement policy language can be a good candidate for SELinux style access control.
- TEpla is a certified policy language.
- The Coq Proof Assistant has been used to develop proofs for theorems
- We analyzed the behavior of the language by defining different ordering relations on policies, queries, and decisions.
- The behavior of the language are presented by a set of formal properties including *order preservation*, *independent composition*, *non-decreasing*, and *determinism* (defined in [Tschantz, Krishnamurthi, SACMAT, 2006]). A deterministic language always produces the same decision for the same policies and queries.
- This insight into language behavior provides a formal way to analyze and reason about language specifications, i.e., policies written in the language.

Outline

Syntax in Coq

The Coq Proof Assistant

- A proof assistant to develop machine-checked proofs
- Often used to verify the correctness of programs: Certified programs
- Properties and proofs are formalized in a special language
- Proofs are developed in a semi-interactive manner dependent on human guidance.
- A further strength of Coq is that executable programs can also be extracted from constructive proofs.

TEpla Syntax and its Encoding in Coq

```
Definition ℂ := ℕ.
Definition ℙ := ℕ.
Definition basic𝕋 := ℕ.
(*  examples *)
Definition File : ℂ := 600.
Definition mail_t : basic𝕋 := 300.
Definition Read : ℙ := 702.
Definition networkManager_ssh_t : basic𝕋 := 302.

Definition 𝔾 : Set := list (basic𝕋).
 Inductive 𝕋 : Type:=
  | single𝕋 : basic𝕋 → 𝕋
  | group𝕋 : 𝔾 → 𝕋.

(*  examples *)
Definition program_𝔾 : 𝔾 := [mail_t ; http_t ].
Definition program_𝕋 : group𝕋 program_𝔾.
```

TEpla Syntax and its Encoding in Coq (contd.)

```
Inductive ℝ : Set :=
   | Allow : 𝕋 * 𝕋 * ℂ * ℙ * 𝔹 → ℝ
   | Type_Transition : 𝕋 * 𝕋 * ℂ → ℝ.

  (* example *)
Definition ℝ_A : ℝ :=
  Allow (group𝕋 program_𝔾, single𝕋 mail_t, File, Read, true ).

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

TEpla Syntax and its Encoding in Coq (contd.)

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

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


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

Ordering Relations on Decisions, Policies, Queries

Ordering Relations (poset)

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

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

$(\mathbb{Q}, <<=)$ In TEpla,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.

$(\mathbb{DCS}, <::)$ We define the binary relation "<::" on this three element set as *NotPermitted* <:: *Permitted* <:: *UnKnown* to define the poset $(\mathbb{DCS}, <::)$

The *UnKnown* decision arises from conflicts in policies. The semantics of TEpla is a *homomorphism* on the posets we defined on $\mathbb{TEPLCY}$, $\mathbb{Q}$ and $\mathbb{DCS}$.

Semantics in Coq

Semantics in Coq - Evaluating Queries Against Policy Rules

Evaluating a query against single rule:

```
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.
```

Semantics in Coq (contd.) - Evaluating a list of TErule

Evaluation of a query against a list of rules:

```
Fixpoint listℝ_EvalTE
  (listℝ: list ℝ) (qry: ℚ) : 𝔻𝒞𝕊:=
  match listℝ with
      |rule_h :: rule_body ⇒
               maximal𝔻𝒞𝕊 (ℝ_EvalTE rule_h qry)
                          (listℝ_EvalTE rule_body qry)
      |[]   ⇒ NotPermitted
 end.
```

Semantics in Coq (contd.)-Evaluating Queries against Constraints

After evaluating a query against the list of rules in the policy, we evaluate the query against constraints. This function evaluates a query against a constraint.

```
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.
```

Semantics in Coq (contd.) - Finding the decisions of policies

This function takes two decisions as arguments and applies the logic of combining the decisions of the two components of policies.

```
Definition maximalpolicy_𝔻𝕔𝕤
        (compont_ℝ compont_ℂ𝕊𝕋𝔼 : 𝔻𝕔𝕤) : 𝔻𝕔𝕤:=
   if(Permitted <:: compont_ℝ) then
         maximal𝔻𝕔𝕤 compont_ℝ compont_ℂ𝕊𝕋𝔼
   else    NotPermitted.
```

Semantics in Coq (contd.) - Evaluating a query against a policy

The functions for evaluating a query against a set of rules and a set of constraints and calling the helper function maximalpolicy_$\mathbb{DCS}$ to combine the results.

```
Fixpoint TEPLCY_EvalTE
 (policy: TEPLCY) (q: ℚ) : DCS:=
 match policy with
  | TEPolicy (CompOne_ℝList, CompTwo_CSTEList) ⇒
     maximalpolicy_DCS
      (listℝ_EvalTE CompOne_ℝList q)
      (listCSTE_EvalTE CompTwo_CSTEList q CompOne_ℝList) end.
```

Conditions on Predicates

Conditions on Predicates

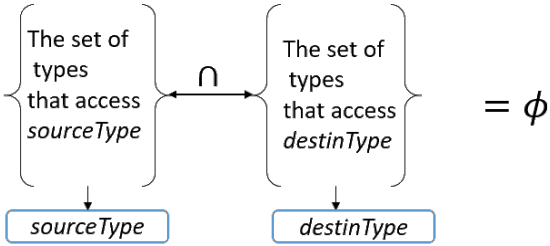
Predicates defined by policy writers must satisfy three conditions and they must prove these conditions are met. The conditions are:

1. `Predicate_`$\mathbb{Q}$`_condition`: Evaluating predicates given any two queries $\mathbb{Q}_1$ and $\mathbb{Q}_2$ such that $\mathbb{Q}_1 <<= \mathbb{Q}_2$, if d_1 and d_2 are the decisions resulting from evaluating $\mathbb{Q}_1$ and $\mathbb{Q}_2$, respectively, then $d_1 <:: d_2$
2. `Predicate_plc_cdn`: 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.
3. `Predicate_plc_cdn_Transition`: given a query $\mathbb{Q}$ and two rule lists `list`$\mathbb{R}$, and `list`$\mathbb{R}$ ++ `list`$\mathbb{R}'$, if d_1 and d_2 are the decisions resulting from evaluating $\mathbb{Q}$ against the list of constraints, respectively, then $d_1 <:: d_2$

An Example Predicate to Express Separation of Duty (SoD)

Encoding SoD security goal

```
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_emptylist T ( IntersectionList
                          (listℝSearch_subjectTs listℝ PRDTsrcT)
                          (listℝSearch_subjectTs listℝ PRDTDesT))
  else true.
```



Verifying the predicate Prd_SoD

Verifying the predicate Prd_SoD satisfies the three conditions on predications:

```
Lemma  qry_condition_SoD 𝕄𝕊𝔻𝕋:
           Predicate_ℚ_condition Prd_SoD.

Lemma  plc_conditionS_SoD 𝕄𝕊𝔻𝕋:
          Predicate_plc_cdn  Prd_SoD.

Lemma  plc_conditionF_SoD 𝕄𝕊𝔻𝕋:
             Predicate_plc_cdn_Transition  Prd_SoD.
```

Formal Language Properties of TEpla

Order Preservation of TEpla Queries

Order Preservation

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 $\mathbb{TEPLCY}\_\texttt{EvalTE}$ on q_1 and q_2, respectively, are in the relation $d_1 <:: d_2$.

```
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.
```

Non-Decreasing Property of TEpla Policies

Non-Decreasing

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 $\oplus$ 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.

```
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.
```

Independent Composition of TEpla Policies

Independent Composition

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.

```
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.
```

Future Work and Conclusion

Conclusion and Future Work

- The infrastructure as well as formal properties of TEpla is encoded in the Coq proof assistant.
- TEpla is certified in terms of formal semantics and machine-checked proofs of a particular set of properties.
- TEpla provides the language constructs for allowing security administrators to encode different security goals in policies.
- TEpla is a certified solution for challenges and drawbacks that security administrators face in developing or analyzing security policies.
- There are some limitations in the language, such as the limited number of arguments for constraints
- The certification of TEpla done so far will help with the development of the next versions of TEpla.
- Program extraction is possible to OCaml, Scheme, Haskell

Thank you for your attention. Questions?

Links:

- TEpla Coq Code: TEpla Coq code
- Modeling and Analysis of Access-Control Policies (SELinux): Modeling and Analysis of access-control policies

References:

- Eaman, A.: TEpla: A Certified Type Enforcement Access Control Policy Language.,Ph.D. thesis, University of Ottawa (2019), TEpla: A Certified TE Policy Language
- Eaman, A., Sistany, B., Felty, A.: Review of existing analysis tools for SELinux security policies: Challenges and a proposed solution. In: 7th International Multi-disciplinary Conference on e-Technologies (MCETECH). pp. 116–135 (2017) SELinux Policy Challenges