Policy Refinement of Network Services for MANETs

Hang Zhao
Dept. of Computer Science
Columbia University
zhao@cs.columbia.edu

Jorge Lobo
IBM T. J. Watson
Hawthorne, New York USA
jlobo@us.ibm.com

Arnab Roy
IBM T. J. Watson
Hawthorne, New York USA
arnabroy@us.ibm.com

Steven M. Bellovin
Dept. of Computer Science
Columbia University
smb@cs.columbia.edu

***Abstract*—In this paper, we describe a framework for a refinement scheme located in a centralized policy server that consists of three components: a knowledge database, a refinement rule set, and a policy repository. The refinement process includes two successive steps: policy transformation and policy composition. Our refinement scheme takes policies written in our logic-based abstract policy language as input and generates low level rules directly implementable by individual enforcement points. We provide concrete policy examples in a coalition scenario that forms a mobile ad hoc network (MANET). We demonstrate policy composition using a distributed firewall scheme named ROFL (ROuting as the Firewall Layer) and access control list as enforcement mechanisms.**

Keywords: Policy, Refinement, Authorization, MANETs

I. Introduction

It is increasingly important to develop policy refinement that automates high level requirements into low level implementation in policy-based system management. The goal of policy refinement is to generate low level rules so that syntax and semantics can be understood by individual enforcement points. Policy refinement fills the gap between policy authoring and enforcement. While these two techniques have been studied intensively, only limited work has addressed policy refinement.

In this paper, we propose a framework to automatically transform security policies into implementable and enforceable rules. We introduce a centralized policy server consisting of three components: a *knowledge database* recording information on the policy domain, a *refinement rule set* defining rules for policy transformation and composition, and a *policy repository* storing policies written at different levels of specification. This systematic approach is able to cope with generic access control policies written in the format proposed in [16]:

{Subject} can (or cannot) {Action} {Target} if {Condition}.

Given the expressiveness of such template, we focus on a subset of stateless access control policies, where action is either permitting or prohibiting a service provided by a target. Such policies on network services are widely used, like the access control list implementation for mandatory access control (MAC), firewall policies, etc. Specifically, the refinement rules presented here work for policies of the following form:

{Subject} can (or cannot) {access Service provided by} {Target} if {Condition}.

We propose a generic framework of policy refinement for access control policies (Section II); In Section III, we introduce a logic-based abstract policy language to assist refinement, and define rules for network service policies in Section IV; In Section V, we discuss mechanisms to handle policy updates due to database maintenance. We provide concrete examples on policy refinement in a coalition scenario with access control list and ROFL [20], a distributed firewall mechanism implemented using routing techniques, as enforcement mechanisms.

II. System Overview

A. Distributed Policy Scenario

We will work with the policy scenario introduced in [11] for the study of distributed policy analysis and refinement. Each organization in this scenario owns devices, networks, command centers, sensors and other equipment; each party keeps its organization-specific domain knowledge private; and each organization has its own *policy server*, which stores policies and performs policy analysis and refinement tasks. A coalition is formed of US forces, UK forces, and the Red Cross (RX). It is a reasonable assumption that the US and the UK domain have a similar structure. We address the US domain, focusing specifically on two sample policies written in natural language:

[Each US device is permitted to access location information from one US location server with high quality, if communication is encrypted and both sides are from the same quad.] (1)

[Devices belonging to non-US organizations for coalition ITA are prohibited to query sensor data from any US sensor fabric with high quality between 9am and 5pm.] (2)

B. Policy Server

Before introducing our policy refinement scheme, we describe the components and functionalities of a centralized *policy server* (Figure 1) where policy analysis and refinement take place. Each organization participating in a coalition has

Research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

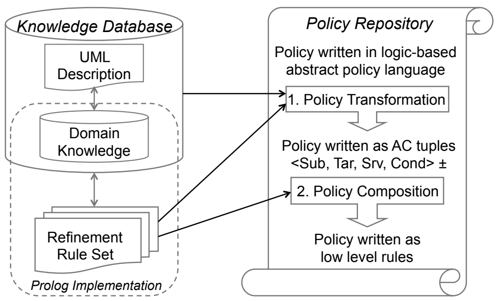


Fig. 1: A centralized policy server for each domain

a centralized policy server consisting of three main components: 1) A *Knowledge Database*: records information on a policy domain; 2) A *Refinement Rule Set*: defines refinement procedure in terms of rules; 3) A *Policy Repository*: stores policies written at different levels of representation.

The *Knowledge Database* captures two categories of knowledge: intra-domain knowledge of confidential information disclosed within an organization, and inter-domain knowledge of coalition participants outside the organization. Inter-domain knowledge is limited by how much information an organization is willing to share with other coalition participants. For example, the UK force knows the existence of a sensor fabric in the US domain as part of inter-domain knowledge, but may not have further details of the structure and components of the US sensor fabric. There are different representations to model a knowledge database. In this work, we describe our database using UML description and logic representation.

The *Refinement Rule Set* defines two types of rules: transformation rules that transform policies written in a logic-based abstract language into access control (AC) tuples $\langle Sub,\ Tar,\ Srv,\ Cond\rangle\pm$; composition rules that generate low-level rules from those tuples. Our policy refinement scheme is domain-independent, such that modifications on a policy domain do not affect refinement rules. Transformation rules are language-independent, taking policies in our abstract language as input; composition rules are highly language-specific, following exactly the syntax and semantics of low-level rules implementable by individual enforcement points.

The *Policy Repository* stores policies written at different levels of representation. For instance, it stores policies written in a structured natural language, ROFL rules ready to be shipped to individual enforcement points, and any intermediate results generated during refinement process.

In the rest of this section, we present a 2-step procedure for constructing a knowledge database: 1) drawing an UML class diagram to describe the structure and components of a policy domain; 2) building a database using logic representation.

C. UML Description

We present a UML description that describes a subset of the US policy domain in Figure 2. Each class in the UML diagram represents a group of objects stored in the knowledge database. A class also has properties, such as attributes and methods. For example, the *Device* class has four attributes named *devID*, *devName*, *devType* and *devLoc* automatically inherited by all

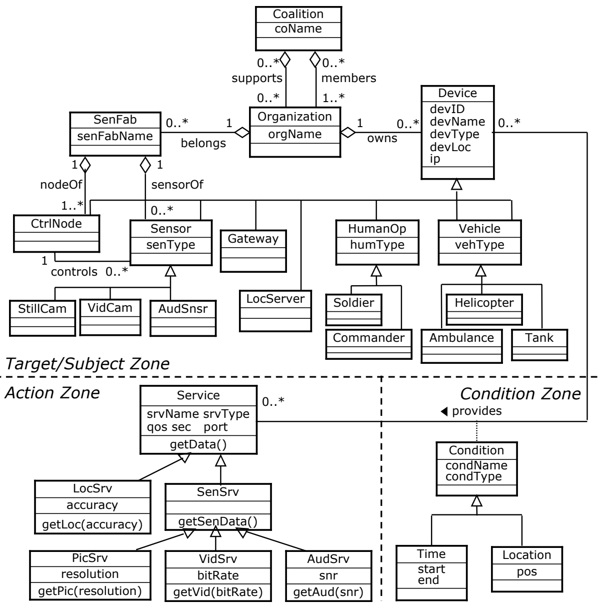


Fig. 2: UML class diagram for the policy domain

its child classes. Child classes are also allowed to have their own attributes and methods.

A line connecting two classes defines an association between them. Three types of associations are supported: *generalization* (i.e., IS-A relationship) describing relationship between parent and child classes; *aggregation* (or *composition*) representing relationship between aggregate (or whole) class and part class; and regular associations not falling into the first two categories. Each association, except for generalization, is named uniquely, and can have their own attributes to form an class. For example, association class *Condition* describes additional information on service provision (i.e., association named *provides*) between class *Device* and class *Service* with two attributes *condName* and *condType*.

To facilitate policy representation, we group classes into different zones. Instances from the *target zone* define targets in such policies; those from the *subject zone* represent subjects. Those two zones coincide in Figure 2. The *action zone* describes actions that a target can take. Finally, the association class *Condition* captures additional constraints on service provision.

D. Domain Knowledge

We construct domain knowledge using logic representation from the UML class diagram defined previously. Six types of definitions are maintained. Each class in the UML diagram is defined using predicate *class(C)*, where C is a class name.

$$class(device).\quad class(service).\quad class(condition).$$

Instances of individual classes are described by predicate *obj(O, C)*, where O is an object name and C is the name of class that O belongs to.

$$obj(sc1, device).\quad obj(pic1, service).\quad obj(time1, condition).$$

We use predicate *att(O, X, Y)* to represent an attribute named X with value Y for instance O.

$$att(sc1, devName, sc1).\quad att(sc1, devType, stillCam).$$

Predicate *assType(X, C_1, A, C_2)* defines an association of type X (*agg* for aggregation, *comp* for composition and *reg* for regular association), identifiable by an unique name A between two classes C_1 and C_2. Service provision (i.e., *provides*) is a regular association between *device* class and *service* class.

$$assType(agg, organization, members, coalition).$$
$$assType(reg, device, provides, service).$$

The actual relationship between two instances O_1 and O_2 is described using predicate *ass(X, O_1, A, O_2)* that takes instances rather than classes as arguments.

$$ass(agg, us, members, ita). \quad ass(agg, uk, members, ita).$$
$$ass(agg, sc1, owns, us). \quad ass(reg, sc1, provides, pic1).$$

Unlike other associations, a generalization is defined as *isa(C_1, C_2)* between a pair of child class C_1 and parent class C_2:

$$isa(sensor, device). \quad isa(stillCam, sensor).$$
$$isa(senSrv, service). \quad isa(picSrv, senSrv).$$

Attribute *port* (i.e. port number) from class *service* is often determined by other attributes of the same class as follows:

$$\mathcal{F}(att_1,\ \ldots,\ att_n) = port$$

As an example, port number for web service is determined by its security feature such that regular http traffic goes through port 80 and encrypted https traffic goes through port 443.

E. Implementation

We choose Prolog [10], a popular general purpose logic programming language, for a reference implement of the policy server in Figure 1. We use an open source implementation called SWI-Prolog [3] together with its plugin ProDT [2] developed for Eclipse [1].

III. Policy Language

To assist policy refinement, we present a logic-based abstract language, intended to serve as a generic formal language which multiple policy languages can be translated into and out of during the refinement process. The language grammar written in Backus–Naur Form (BNF) is summarized in Figure 3. Comparison operators ($=$, $\neq$, $>$, $\geq$, $<$, $\leq$), logical connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$) and quantifiers ($\forall$, $\exists$) are integrated into our language to provide expressiveness. Terms in italic are ground *Instance*, *Class*, *Attribute*, *Association* names and association types from the UML description and the knowledge database.

A $policy$ consists of an authorization rule $auth$, a sign to indicate positive authorization $+$ (permitting a service) and negative authorization $-$ (prohibiting a service). Each $auth$ rule is a tuple $\langle Sub,\ Perm,\ Cond\rangle$ of three fields: Sub is the subject of an authorization policy; $Perm$ is further defined as a tuple $\langle Tar,\ Srv\rangle$ that represents a service Srv provided by target Tar; $Cond$ denotes an optional condition field. More specifically, Sub defines that a refined subject Sub' is an object O satisfying predicate Exp, where O belongs to class C (i.e. $obj(O,\ C)$) from *subject zone* satisfying both attribute constraints (*C_att*) and association constraints (*C_ass*). Tar and Srv are also defined in a similar way. Quantifier Q appears preceding with Sub, $Perm$, Tar and Srv for greater expressiveness. $Cond$ is represented as a logic expression on condition element d and cross-field attribute constraints C_cf.

Our language supports compound constraints by defining C_att as an arbitrary propositional composition of constraint

```
policy  ::=  auth sign ;
  auth  ::=  "⟨" Q Sub "," Q Perm [ "," Cond " ] ⟩" ;
  Perm  ::=  "⟨" Q Tar "," Q Sub ⟩" ;
   Sub  ::=  "Sub′" ∈ "{" O "|" Exp "}" ;
   Tar  ::=  "Tar′" ∈ "{" O "|" Exp "}" ;
   Srv  ::=  "Srv′" ∈ "{" O "|" Exp "}" ;
  Cond  ::=  C_d | C_cf | Cond ∧ Cond

     O  ::=  Instance ;
     C  ::=  Class ;
     X  ::=  AssType ;
   Att  ::=  Attribute ;
   Ass  ::=  Association ;

     Q  ::=  ∀ | ∃ ;
  sign  ::=  + | − ;
    op  ::=  = | ≠ | > | ≥ | < | ≤ ;
     s  ::=  O "." Att | a ;
     c  ::=  s op s ;
 C_att  ::=  c | ¬ C_att | C_att ∧ C_att | C_att ∨ C_att
     F  ::=  "Tar′" | "Srv′" | "Sub′" ;
     e  ::=  F "." Att ;
     f  ::=  e op e ;
  C_cf  ::=  f | ¬ C_cf | C_cf ∧ C_cf | C_cf ∨ C_cf
     d  ::=  obj "(" O "," C ")" ∧ C_att ;
   C_d  ::=  d | C_d ∧ C_d | C_d ∨ C_d

   l_c  ::=  Q O "(" ass "(" X "," O "," Ass "," O ")"
                  [ ∧ C_att ] → C_att ")" ;
   l_o  ::=  Q O "(" ass "(" X "," O "," Ass "," O ")"
                  [ ∧ C_att ] → [ C_att ∧ ] ;
     L  ::=  l_c | ¬ L | L ∧ L | L ∨ L
 C_ass  ::=  L | l_o C_ass ")" | ¬ C_ass |
             C_ass ∧ C_ass | C_ass ∨ C_ass
   Exp  ::=  obj "(" O "," C ")" [ ∧ C_att ] [ ∧ C_ass ] ;
```

Fig. 3: Grammar for a logic-based abstract policy language. Meta-symbol | specifies multiple choices. Optional items are enclosed in [and]. Repetitive items (zero or more times) are enclosed in { and }. Terminals of one character are surrounded by quotes (") and ; is the termination symbol.

element c, including negation ($\neg$) given that op is closed under negation. Each constraint element c compares two subexpressions of form s, where s is either an instance attribute $O.Att$ or a constant a. We also support cross-field attribute constraints C_cf that compares attributes of different fields (i.e., Sub', Tar' and Srv'). The condition C_att is limited to the object and class in the field where they appear.

C_ass defines the association constraints held for an object O. Expression *l_c*, the basic building block for any compound association constraints, describes a closed relationship between two objects that traverse exactly one link (association). Quantifiers are required for one-to-many or many-to-many associations with scope of the entire expression *l_c*. Since association name Ass uniquely defines the two end classes, the class definition for O' is omitted. Notice that *l_c* contains two optional attribute constraints: *C_att* appearing before "$\rightarrow$" further restricts the selection of O'; whereas *C_att*$'$ after "$\rightarrow$" specifies properties held for selected O'. Thus *l_c* states that for all objects (or exists one object) O' associated with object O through Ass with property *C_att* held, *C_att*$'$ must also hold for those O'. Figure 4 depicts the four cases. Shaded nodes represent instances that satisfy $ass(X, O, Ass, O')$ with optional attribute constraints *C_att* on O'. Underlined nodes are instances further selected by the quantifier. The four cases are able to describe any subset of objects O' associated with O. L defines an arbitrary propositional composition of *l_c*, as O can be associated with objects through multiple associations. Consider the following *C_ass* associate constraint *[an organization that is a member organization of a coalition*

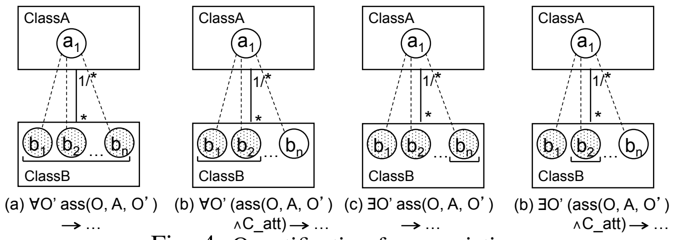

ClassA
a_1
1/*
*
b_1 b_2 … b_n
ClassB
(a) ∀O' ass(O, A, O') → …
(b) ∀O' (ass(O, A, O') ∧C_att) → …
(c) ∃O' ass(O, A, O') → …
(b) ∃O' (ass(O, A, O') ∧C_att) → …

Fig. 4: Quantification for association.

named ita with all its sensor fabric located at west quad, or it is a supporting organization of the same coalition].

$$\begin{aligned} C\_ass \;\equiv\; & (\exists O' \;(ass(agg,\ O,\ members,\ O') \\ & \quad \rightarrow (O'.coName = ita)) \wedge \\ & \forall O'' \;(ass(agg,\ O,\ belongs,\ O'') \\ & \quad \rightarrow (O''.strLoc = westquad))) \vee \\ & (\exists O' \;(ass(agg,\ O,\ supports,\ O') \\ & \quad \rightarrow (O'.coName = ita))) \end{aligned}$$

Unlike l_c, l_o is an open link between O and O' that allows O' to be further associated with other objects in a recursive manner. The last element on a path of consecutive association constraints must be L for C_ass to terminate properly.

$Cond$ is an arbitrary propositional composition of condition element d, excluding negation. Each condition element d is of the form $obj(O,\ C) \wedge C\_att$, where O is an instance of class C from the *condition zone*, like time and location, satisfying attribute constraints C_att.

Authorization policies written in natural language with a constrained lexicon and syntax designed for policy expression can be translated into our language. We assume an automated process that accomplishes the translation. Therefore, our policy refinement scheme starts from an authorization policy already written in our language. We translate initial policies (1) and (2) into our abstract language as follows:

[Each US device is permitted to access location information from one US location server with high quality, if communication is encrypted and both sides are from the same quad.]

$$\begin{aligned} & \Downarrow \\ & policy \equiv \langle \forall Sub,\ \forall\langle \exists Tar,\ \forall Srv\rangle,\ Cond\rangle + such\ that, \\ & Sub \equiv Sub' \in \{O \mid obj(O, device) \wedge (\forall O'\ (ass(agg, O, owns, O') \\ & \qquad\qquad \rightarrow O'.orgName = us))\} \\ & Tar \equiv Tar' \in \{O \mid obj(O, locServer) \wedge (\forall O'\ (ass(agg, O, belongs, O') \\ & \qquad\qquad \rightarrow O'.orgName = us))\} \\ & Srv \equiv Srv' \in \{O \mid obj(O, locSrv) \wedge (O.qos = high \wedge O.sec = crypto)\} \\ & Cond \equiv Tar'.devLoc = Sub'.devLoc \end{aligned} \tag{3}$$

[Devices belonging to non-US organizations for coalition ITA are prohibited to query sensor data from any US sensor fabric with high quality between 9am and 5pm.]

$$\begin{aligned} & \Downarrow \\ & policy \equiv \langle \forall Sub,\ \forall\langle \forall Tar,\ \forall Srv\rangle,\ Cond\rangle - such\ that, \\ & Sub \equiv Sub' \in \{O \mid obj(O, device) \wedge (\forall O'\ (ass(agg, O, owns, O') \\ & \qquad\qquad \rightarrow ((O'.orgName \neq us) \wedge \\ & \qquad (\exists O''\ (ass(agg, O', members, O'') \rightarrow O''.coName = ita) \vee \\ & \qquad \exists O'''\ (ass(agg, O', supports, O''') \rightarrow O'''.coName = ita))))\} \\ & Tar \equiv Tar' \in \{O \mid obj(O, senFab) \wedge (\forall O'\ (ass(agg, O, belongs, O') \\ & \qquad\qquad \rightarrow O'.orgName = us))\} \\ & Srv \equiv Srv' \in \{O \mid obj(O, senSrv) \wedge (O.qos = high) \\ & Cond \equiv obj(O, time) \wedge (O.start = 9am \wedge O.end = 5pm) \end{aligned} \tag{4}$$

IV. Policy Refinement

The refinement scheme presented in this section starts with authorization policies written in our logic-based abstract policy language (Section III). The goal of policy refinement is to translate those policies to low level rules so that their syntax and semantics can be understood by individual devices, i.e. enforcement points.

Definition 1. We define the following transitive closure on generalization (IS-A relationship):

$$\begin{aligned} & isa\_trans(C, C') \leftarrow isa(C, C') \\ & isa\_trans(C, C'') \leftarrow isa(C, C'),\ isa\_trans(C', C'') \end{aligned} \tag{5}$$

Similarly, we define transitive closure on predicate $obj(O, C)$:

$$\begin{aligned} & obj\_trans(O, C) \leftarrow obj(O, C) \\ & obj\_trans(O, C') \leftarrow obj(O, C),\ isa\_trans(C, C') \end{aligned} \tag{6}$$

Definition 2. In UML, the difference between *aggregation* and *composition* is subtle. Aggregation is more like a *has-a* relationship (also known as weak-aggregation); composition is more like a *part-of* relationship (also known as strong-aggregation). Thus we define the following transitive closure on aggregation and composition associations:

$$\begin{aligned} & ass\_trans(ac, O, Ass, O') \leftarrow ass(agg, O, Ass, O'). \\ & ass\_trans(ac, O, Ass, O') \leftarrow ass(comp, O, Ass, O'). \\ & ass\_trans(ac, O, Ass + Ass', O'') \\ & \qquad\qquad \leftarrow ass(ac, O, Ass, O'), \\ & \qquad\qquad\quad ass\_trans(ac, O', Ass', O''). \end{aligned} \tag{7}$$

where $Ass + Ass'$ indicates that object O is associated with object O'' that traverses an aggregation (weak or strong) link Ass and a path Ass' in sequence.

Definition 3. Association $provides$ describing service provision is treated as a regular association between services and their providers. We define transitive closure on predicate ass for $provides$ as follows:

$$\begin{aligned} & ass\_trans(reg, T, provides, V) \leftarrow ass(reg, T, provides, V). \\ & ass\_trans(reg, T', provides, V) \leftarrow ass(reg, T, provides, V), \\ & \qquad\qquad\qquad ass\_trans(ac, T, Ass, T'). \\ & ass\_trans(reg, T, provides, V') \leftarrow ass(reg, T, provides, V), \\ & \qquad\qquad\qquad ass\_trans(ac, V', Ass, V). \end{aligned} \tag{8}$$

The above definition states that target object T' transitively provides service V if T' is an aggregate of object T and T provides V. Similarly, target object T transitively provides service V' if V' is a part of service object V provided by T.

Definition 4. Let $policy$ be an authorization policy written in the logic-based abstract policy language defined in Section III. A refinement rule is an expression:

$$\begin{aligned} & policy \equiv \langle Q_S Sub,\ Q_P\langle Q_T Tar,\ Q_V Srv\rangle,\ Cond\rangle \pm \\ & \Downarrow \text{ref} \\ & policy' \equiv \langle Sub',\ Tar',\ Srv',\ Cond'\rangle \pm \end{aligned} \tag{9}$$

that translates higher-level policy $policy$ into a rule $policy'$ at the lowest level through a gradual refinement. Positive and negative authorization signs are preserved automatically. There are many argument values of $policy'$ to satisfy the refinement rule (9). The selection of $policy'$ is determined by quantifiers $Q_S \dots Q_V$ in the $policy$ expression, and will be discussed in details during the refinement process.

We propose a policy refinement process of two successive phases (see Figure 1):

1) A *policy transformation phase*, that transforms policies written in that logic-based abstract language to tuples by querying the pre-constructed knowledge database using refinement rule (9);
2) A *policy composition phase*, that composes policy rules at lowest level from query results.

A. Policy Transformation

The implementation of refinement rule (9) consists of six successive steps. It starts with policies written in our abstract language so one can easily adapt it to another policy domain.

1) Permission Refinement:

A *permission*, i.e., $perm(T,V)$, defines a service V provided by a target T. Given the target (Tar) and service (Srv) expressions specified in our logic-based abstract language, we get a set of permissions using rule (10):

$$\begin{aligned}
&refPerm(\overbrace{obj(T,C_T), C\_att_T, C\_ass_T}^{Tar},\\
&\qquad \overbrace{obj(V,C_V), C\_att_V, C\_ass_V}^{Srv}, C\_cf, perm(T,V)) \leftarrow\\
&\quad obj\_trans(T,C_T),\\
&\quad checkAttConst(T,C\_att_T),\\
&\quad checkAssConst(T,C\_ass_T),\\
&\quad obj\_trans(V,C_V),\\
&\quad checkAttConst(V,C\_att_V),\\
&\quad checkAssConst(V,C\_ass_V),\\
&\quad ass\_trans(reg,T,provides,V),\\
&\quad checkCFConst(C\_cf, perm(T,V)).
\end{aligned} \tag{10}$$

such that, T is an object transitively belonging to target class C_T, satisfying attribute constraints C_att_T and association constraints C_ass_T; V is an object transitively belonging to service class C_att_V, satisfying attribute constraints C_att_V and association constraints C_ass_V. Besides, target T transitively provides service V. Each resulting permission also satisfies any cross-field constraint specified in C_cf comparing attributes from Tar' and Srv'. The resulting set of permissions are further selected using Eq.(13) based on the quantifiers Q_T, Q_V and their order. As $\forall$ and $\exists$ are not commutative, we have all together six different cases.

2) Subject Refinement:

Subject refinement finds all the subjects S that transitively belong to class C_S and satisfy attribute constraints C_att_S and association constraints C_ass_S based on rule (11).

$$\begin{aligned}
&refSub(\overbrace{obj(S,C_S), C\_att_S, C\_ass_S}^{Sub}, sub(S)) \leftarrow\\
&\quad obj\_trans(S,C_S),\\
&\quad checkAttConst(S,C\_att_S),\\
&\quad checkAssConst(S,C\_ass_S).
\end{aligned} \tag{11}$$

3) Access Refinement:

Access refinement generates a set of access predicates $acc(S, perm(T,V))$ by computing the Cartesian product of permission set $\{perm(T,V)\}$ and subject set $\{sub(S)\}$ using rule (12). The resulting access predicates must also satisfy corresponding cross-field constraints specified in C_cf. Similarly, the set of accesses are further selected based on the value of Q_S, Q_P and their order using Eq.(13).

$$\begin{aligned}
&refAcc(C\_cf, Q_S, Q_P, acc(S, perm(T,V))) \leftarrow\\
&\quad perm(T,V),\\
&\quad sub(S),\\
&\quad checkCFConst(C\_cf, acc(S, perm(T,V))).
\end{aligned} \tag{12}$$

4) Quantification Refinement:

Let $P = \{(x,y)$: value pairs that make predicate $p(X,Y)$ true by assigning variables $X = x$ and $Y = y\}$. Quantification refinement selects elements from P based on the value of quantifiers Q_1 and Q_2 for X and Y respectively. The order of quantifiers also matters. In Eq.(13), $\mathcal{R}$ is a non-deterministic function that returns a maximal set of refined value pairs P' for the initial set of value pairs P given the combination of quantifiers Q_1 and Q_2.[1] Notation $p.X$ ($p.Y$) returns the X (Y) value of an element p.

$$\begin{aligned}
&\mathcal{R}(Q_1, Q_2, P),\ where\ P = \{(x,y)\}\\
&= \begin{cases}
P' = P & \text{if } \forall X \forall Y;\\
P' \subseteq P \land |P'| = 1 & \text{if } \exists X \exists Y;\\
P' \subseteq P \land \forall i \forall j (i \neq j \land p_i, p_j \in P' & \text{if } \forall X \exists Y;\\
\quad \Rightarrow p_i.X \neq p_j.X) & \\
P' \subseteq P \land \forall i \forall j (i \neq j \land p_i, p_j \in P' & \text{if } \forall Y \exists X;\\
\quad \Rightarrow p_i.Y \neq p_j.Y) & \\
P' \subseteq P \land \forall i \forall j (i \neq j \land p_i, p_j \in P' & \text{if } \exists X \forall Y;\\
\quad \Rightarrow p_i.X = p_j.X \land p_i.Y \neq p_j.Y) & \\
P' \subseteq P \land \forall i \forall j (i \neq j \land p_i, p_j \in P' & \text{if } \exists Y \forall X;\\
\quad \Rightarrow p_i.Y = p_j.Y \land p_i.X \neq p_j.X) &
\end{cases}
\end{aligned} \tag{13}$$

5) Granularity Refinement:

Given a set of access predicates after quantification refinement, the goal of granularity refinement is to traverse all the aggregation and composition associations for each field and produce the actual low-level objects that participate in the enforcement of access control. Note that comparing with ass_trans predicate for $provides$, here we want the low level target Tar' and low level service Srv' directly associated through ass predicate.

$$\begin{aligned}
&refGran(acc(Sub', perm(Tar', Srv'))) \leftarrow\\
&\quad acc(S, perm(T,V)),\\
&\quad ass(reg, Tar', provides, Srv'),\\
&\quad ass\_trans(ac, Tar', \_, T),\\
&\quad ass\_trans(ac, Srv', \_, V),\\
&\quad ass\_trans(ac, Sub', \_, S).
\end{aligned} \tag{14}$$

6) Condition Refinement:

Since $Cond$ is an arbitrary propositional composition of condition element d, in rule (15), each $d \equiv obj(O,C) \land C\_att$

[1]The non-determinism can be restricted by other semantic considerations that select, for example, the most appropriate target to perform a service in the current situation.

is refined to a list of condition instances connected using disjunctions, that belong to condition class C and satisfy attribute constraints C_att. Logic connectives among condition elements are automatically preserved. Notice that cross-field constraints C_cf have already been refined in previous steps.

$$\begin{aligned}&Cond \equiv d \mid C\_d \wedge C\_d \mid C\_d \vee C\_d\\&\Downarrow \text{refCond}\\&Cond' \equiv \vee_i^{i\geq 1} O_i \mid C\_d' \wedge C\_d' \mid C\_d' \vee C\_d'\end{aligned} \tag{15}$$

7) Examples:

Following our previous policy examples, we apply refinement rule (9) on policies (1) and (2) by querying the pre-constructed knowledge database to produce the following results:

$$\begin{aligned}&policy \equiv \langle \forall Sub,\ \forall\langle \exists Tar,\ \forall Srv\rangle,\ Cond\rangle +\\&\Downarrow \text{ref}\\&policy' \equiv \langle sc1,\ ls1,\ loc3,\ \emptyset\rangle +\end{aligned} \tag{16}$$

$$\begin{aligned}&policy \equiv \langle \forall Sub,\ \forall\langle \forall Tar,\ \forall Srv\rangle,\ Cond\rangle -\\&\Downarrow \text{ref}\\&policy' \equiv \langle ls3,\ sc1,\ pic3,\ time1\rangle -\end{aligned} \tag{17}$$

Rule (16) refines policy (1) into a low-level rule $policy'$ saying that US still camera $sc1$ is allowed to access location service $loc3$ (high quality with encryption) provided by US location server $ls1$. There are many other $policy'$ satisfying rule (16) but rule (13) ensures that each subject is allowed to access one location server, i.e., $\langle sc1,\ ls2,\ loc3,\ \emptyset\rangle +$ is not a valid refinement if $\langle sc1,\ ls1,\ loc3,\ \emptyset\rangle +$ already exists. Similarly, $policy'$ in rule (17) is one possible refinement of policy (2), saying that UK location server $ls3$ is prohibited to access picture service $pic3$ (high quality) provided by US still camera $sc1$ at given time $time1$ ($9am - 5pm$).

B. Policy Composition

It is often the case that access control tuples generated from the *policy transformation* phase cannot be directly implemented because their syntax may not be understood by low-level devices. Thus the goal of *policy composition* is to generate low-level policies from those tuples. This step is highly language-dependent, because the final output is a set of low-level rules written in a policy language specification determined by the choice of underlying enforcement mechanism.

So far we have been focusing on network services in MANETs, therefore we choose the following two mechanisms for enforcement: 1) Access control lists (ACLs) that are maintained locally at each service provider; 2) ROFL scheme that implements packet filtering using routing mechanisms.

1) ACLs:

Local access control lists maintained at servers are composed from results of policy transformation using rule (18):

$$\begin{aligned}&policy' \equiv \langle Tar',\ Srv',\ Sub',\ Cond'\rangle \pm\\&\Downarrow \text{ACL}\\&acl(Tar') = \langle Sub',\ \pm Act',\ Cond'\rangle\end{aligned} \tag{18}$$

where $policy'$ is a refined policy produced by transformation rule (9), $acl(Tar')$ denotes an access control list on object Tar'. The operation field Act' is defined as a method provided by the object class C of Tar'. This method may take zero or more attributes of Tar' as parameters, such that $Act' = C.method(Tar'.Att_1,\ \ldots,\ Tar'.Att_n)$, where $n \geq 0$ and $obj(Tar', C)$. The positive or negative authorization sign is placed in front of Act' to indicate whether certain operation is allowed to performed or not. Alternatively, only positive authorization rules are maintained in ACLs. Hence any operation that is not explicitly granted is prohibited. Finally, the condition field $Cond'$ is required only if the implementation support complex ACLs with additional constraints.

As concrete examples, we generate ACLs for refined rules (16) and (17) respectively.

$$\begin{aligned}&policy' \equiv \langle sc1,\ ls1,\ loc3,\ \emptyset\rangle +\\&\Downarrow \text{ACL}\\&acl(ls1) \equiv \langle sc1,\ +getLoc(loc3.accuracy),\ \emptyset\rangle\end{aligned} \tag{19}$$

$$\begin{aligned}&policy' \equiv \langle ls3,\ sc1,\ pic3,\ time1\rangle -\\&\Downarrow \text{ACL}\\&acl(sc1) \equiv \langle ls3,\ -getPic(pic3.resolution),\ time1\rangle\end{aligned} \tag{20}$$

2) ROFL Scheme:

Now we demonstrate the composition process for rules written as ROFL advertisements. ROFL is based on a simple notion: services — that is, port numbers — should be treated as part of the IP address in the routing system. (Full details are given in [20], [19].) If a certain service is not advertised to a particular network, no host on that network can reach it; the routing system will not deliver the packets. We thus use every router along the path as a firewall. There are many benefits to this scheme, especially in MANETs where battery power is limited. If unwanted packets are dropped very early, a lot of power can be saved by not transmitting those packets. A ROFL route advertisement looks like the following:

$$R = \{d : s/m,\ S,\ L,\ M\}$$

where d denotes the *target* host IP address, s specifies the *service* provided by that target, m is the destination prefix length (and $m \leq 48$), S represents a set of authorized *subjects*, L is a list of traffic labels, with $M \neq \infty$ indicating a positive authorization and $M = \infty$ indicating a negative authorization. R is only disseminated to hosts in the *subject* set S. Upon receiving R, all the fields in R including traffic label L enter into a receiver's local routing table.

With ROFL, a host knows which ports are needed for which sources, and can emit proper route advertisements. Nor are routers turned into firewalls, except in effect; they simply listen to routing advertisements and forward packets as usual, albeit with longer addresses. No extra administration or state is needed, [20] presents calculations showing that the increase in table size is acceptable. Other potential issues are routing table computations ([20]) and increased routing traffic ([19]). We have shown that the increase in traffic for routing messages is more than outweighed by the savings by early drops of unwanted traffic.

Generation of ROFL advertisements from refined policy rule is defined in rule (21):

$$\begin{array}{ll} policy' \equiv \langle Tar',\ Srv',\ Sub',\ Cond' \rangle \pm & \\ \Downarrow \text{ROFL} & (21) \\ R \equiv \{Tar'.ip : Srv'.port/48,\ Sub'.ip,\ \mathcal{L}(Cond'),\ M\} & \end{array}$$

where $M = \infty$ for negative authorization policy. Now let us discuss the mapping for each field in more details.

In rule (21), mapping from target or subject to its IP address is straightforward as IP is an attribute for Tar' or Sub'. However, it is often the case that a ROFL announcement R propagates to a set of permitted subjects. Thus it is more efficient to enclose the whole set of subjects in a single announcement to minimize overhead. Therefore, it is possible to implement it as a Bloom filter [8] on the set of source addresses or networks of a given prefix length. Bloom filters are a space-efficient data structure that can compress the representation of a set of members in a compact manner, albeit with some chance of false positives. Bloom filters are particularly useful in MANETs, where there is little topological structure and each allowed node may be identified by a flat address.

Mapping from service to port number is obtained by calling $Srv'.port$. Function $\mathcal{F}$ in section II computes port number from service attributes. The more attributes involved in $\mathcal{F}$, the more effective a ROFL advertisement is as more unwanted traffic is filtered out. For instance, if $port$ is determined by both qos and sec, an announcement R can filter more unwanted traffic if one of the attribute constraints unsatisfied.

$Cond'$ is mapped to labels by calling function $\mathcal{L}(Cond')$:

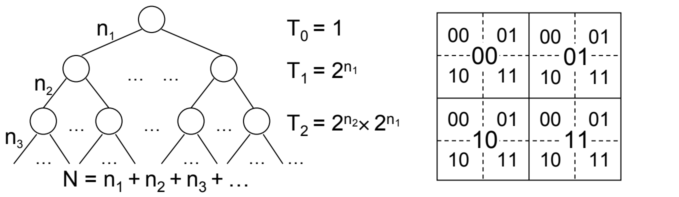
$$\begin{array}{rcl} \mathcal{L}(Cond') & = & \mathcal{L}(\vee_{j=1}^{n} \wedge_{i=1}^{m} (\vee_{k=1}^{p} O_{ij_k})) \\ & = & \mathcal{L}(\wedge_{j'=1}^{n'} \vee_{i'=1}^{m'} O_{i'j'}) \\ & = & l_{\wedge_{j'=1}^{n'}} ({}_{i'=1}^{m'} l_{i'j'}) \end{array}$$

where it is firstly rewritten into a conjunctive normal form by applying distributive property of logical connectives; then each condition object $O_{i'j'}$ is mapped into a label $l_{i'j'}$. Logical operator $\vee$ is automatically implied between consecutive labels, and operator $\wedge$ is replaced by a special label $l_{\wedge}$.

Now we describe the encoding algorithm from a condition object O to a label l. Each l is a string of 8 consecutive bits, where the first 4 bits denote $O.condType$ and the remaining represent $O.condName$. Together they uniquely identify a condition instance in the knowledge database. Condition type 0000 is reserved, and $l_{\wedge} = 00000000$. Thus our scheme supports 15 types of conditions (although Figure 2 shows only two types of conditions in this scenario) and 16 different values for each type. More bits can be added to represent more labels.

Optimization can be made in different ways. The goal is to minimize the length of label list L in a ROFL advertisement, and hence reduce processing time. Wildcard character $*$ represents a bit value of either 0 or 1. Thus 2^n consecutive labels (logical connective $\vee$ implied) different by n bits at fixed positions $b_1, \ldots, b_n$ can be replaced by one label with wildcard $*$ at those positions and the rest remain unchanged. For example, $00010010|00010000$ (vertical bar $|$ is for presentation purpose only) can be replaced by $000100*0$.

A more efficient encoding mechanism can also reduce the length of L. We propose an encoding tree (Figure 5(a)) to generate labels of the same type, i.e. those have the first 4 bits in common. Root node represents the entire value space for a



(a) Encoding tree for label value of *N* bits (b) Encoding scheme for *Location* label

Fig. 5: Encoding scheme for label value

certain type of label; T_1 denotes 2^{n_1} sub-spaces identified by first n_1 bits; similarly, T_2 further divides each sub-space into 2^{n_2} partitions using the next n_2 bits, and so on until all N bits for label values are used. Thus, only one label is necessary to represent a node or a subtree. Figure 5(b) depicts a possible encoding scheme for *Location* labels with $N = 4$. The entire battle field is divided into 4 quads identified by the first 2 bits, and each quad is further split into four sub-quads using the remaining 2 bits. Due to the space limitation, we will not further discuss other optimization schemes for label encoding.

As a concrete example, we compose ROFL advertisements from the refinement results obtained previously.

$$\begin{array}{ll} policy' \equiv \langle sc1,\ ls1,\ loc3,\ \emptyset \rangle + & \\ \Downarrow \text{ROFL} & (22) \\ R \equiv \{10.0.0.1 : 443/48,\ 10.0.0.10,\ 00000000,\ M\} & \\ policy' \equiv \langle ls3,\ sc1,\ pic3,\ time1 \rangle - & \\ \Downarrow \text{ROFL} & (23) \\ R \equiv \{10.0.0.10 : 80/48,\ 20.0.0.1,\ 00010001,\ \infty\} & \end{array}$$

V. Policy Updates

In this section, we focus on how our policy refinement process can cope with policy updates when the knowledge database changes. We will not discuss situations when new initial policies are introduced, as those new policies will go through the same refinement process as existing ones.

A. Correctness of Knowledge Database

To generate consistent policies, knowledge database $\mathcal{D}$ must be conflict-free with the following requirements enforced:

1) The definition of class and associations among classes is self-contained, such that: if $isa(C, C') \in \mathcal{D}$, then $class(C), class(C') \in \mathcal{D}$; if $assType(X, C, A, C') \in \mathcal{D}$, then $class(C)$, $class(C') \in \mathcal{D}$.
2) The definition of instances and associations among them is self-contained, such that: if $obj(O, C) \in \mathcal{D}$, then $class(C) \in \mathcal{D}$, $att(O, Att_i, V_i) \in \mathcal{D}$, for all Att_i of $class(C)$; if $ass(X, O, A, O') \in \mathcal{D}$, then $class(C)$, $class(C')$, $assType(X, C, A, C')$, $obj(O, C)$, $obj(O', C') \in \mathcal{D}$.
3) The definition of service instances cannot stay alone without their service providers: if $obj(O, C) \in \mathcal{D}$ and $class(C)$ is from the *service zone*, then $obj(O', C') \in \mathcal{D}$ such that $class(C')$ is from the *target zone*, and $ass(reg, O', Ass, O) \in \mathcal{D}$, where Ass is an association describing service provision.

Standard techniques [17], [15] for constraint verification and integrity checking on knowledge database can be applied there. We implement those techniques using logic programming.

B. Update of Knowledge Database

From the system point of view, adding new objects (or classes) or removing existing ones only affects knowledge database not the rest of the policy server (Figure 1). Table I summarizes the changes one needs to perform for operations in the first column, where Y means the modification is mandatory, – implies optional, and N means not required.

Operation	***class***	***isa***	***obj***	***att***	***assType***	***ass***
Add/Remove Tar/Sub O	N	N	Y	Y	N	–
Add/Remove Tar/Sub C	Y	–	N	N	–	N
Add/Remove Srv O	N	N	Y	Y	N	Y
Add/Remove Srv C	Y	–	N	N	Y	N
Add/Remove Cond O	N	N	Y	Y	N	–
Add/Remove Cond C	Y	–	N	N	–	N

TABLE I: Update of knowledge database upon operations

Clearly, adding or removing target (or subject) objects only affects predicate obj and att. It might affect associations, such as aggregation, composition, etc. On the other hand, adding or removing target (or subject) classes only affects the class definition $class$, and definition on associations among classes (i.e. isa and $assType$) may be updated as well. Update on service objects (or classes) is handled in a similar way except that predicates $assType$ and ass must be updated to enforce requirement 3) discussed in previous subsection. Updates on condition objects (or classes) is handled the same as target objects (or classes).

VI. Related Work

Early research [4] performs theoretical work on refinement mappings to prove that a lower-level specification correctly implements a higher-level one. Recent studies [5], [6], [7], [9], [18] address subsets of the problem, such as taking the goal-oriented approach for goal decomposition using Event Calculus, mapping policy objectives to specific configuration details using transformation algorithms, etc. In [12], some initial work on policy transformation is presented that applies syntactic and algorithmic ideas adapted from the concepts of data integration. In [13], the same group proposes action decomposition techniques towards a framework for automated distributed refinement of both authorization and obligation policies. Our approach describes a framework systematically for access control policies in general, specifically focusing on network services enforced by authorization policies. We address the needs of policy composition to produce directly enforceable low-level rules through concrete examples. Other related work includes [14], where harnessing knowledge embodied in information models and ontologies is used to represent relationships between policy components that could indicate potential conflicts between policies.

VII. Conclusion

In this paper, we have described a refinement process for network service policies in a generic policy refinement framework. Future work includes further development to support access control policies with complex actions. Moreover, refinement and consistency checking could be interleaved to verify that refined policies are consistent with respect to existing policies. On the other hand, in a fully distributed scenario, partial refinement may be more desirable with local domain knowledge and a relevant subset of refinement rules.

References

[1] "Eclipse," http://www.eclipse.org/.

[2] "ProDT: Prolog development tools," http://prodevtools.sourceforge.net/.

[3] "SWI-Prolog," http://www.swi-prolog.org/.

[4] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theoretical Computer Science*, vol. 82, pp. 253–284, 1988.

[5] A. K. Bandara, E. C. Lupu, J. Moffett, and A. Russo, "A goal-based approach to policy refinement," in *POLICY '04: Proceedings of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks*. IEEE Computer Society, 2004, p. 229.

[6] A. K. Bandara, E. C. Lupu, A. Russo, N. Dulay, M. Sloman, P. Flegkas, M. Charalambides, and G. Pavlou, "Policy refinement for diffserv quality of service management," in *IM 2005: Proceedings of the 9th IFIP/IEEE International Symposium on Integrated Network Management*, 2005.

[7] M. S. Beigi, S. Calo, and D. Verma, "Policy transformation techniques in policy-based systems management," in *Proc. 5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY*, 2004, pp. 13–22.

[8] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of ACM*, vol. 13, no. 7, pp. 422–426, July 1970.

[9] G. A. Campbell and K. J. Turner, "Goals and policies for sensor network management," in *SENSORCOMM '08: Proceedings of the 2008 Second International Conference on Sensor Technologies and Applications*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 354–359.

[10] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, 1984.

[11] R. Craven, J. Lobo, E. Lupu, J. Ma, A. Russo, and M. Sloman, "Distributed policy scenario," ITA Technical Report, 2010.

[12] R. Craven, J. Lobo, E. Lupu, A. Russo, and M. Sloman, "Security policy refinement using data integration: a position paper," in *SafeConfig '09: Proceedings of the 2nd ACM workshop on Assurable and usable security configuration*. New York, NY, USA: ACM, 2009, pp. 25–28.

[13] ——, "Decomposition techniques for policy refinement," in *To appear in proc. of the 6th International Conference on Network and Service Management*, 2010.

[14] S. Davy, "Harnessing information models and ontologies for policy conflict analysis," Ph.D. dissertation, 2008.

[15] P. Grefen and J. Widom, "Protocols for integrity constraint checking in federated databases," in *Proceedings 1st IFCIS International Conference on Cooperative Information Systems*, 1996, pp. 38–47.

[16] M. Johnson, J. Karat, C.-M. Karat, and K. Grueneberg, "Usable policy template authoring for iterative policy refinement," in *POLICY '10: Proceedings of the IEEE International Workshop on Policies for Distributed Systems and Networks, POLICY*, 2010.

[17] R. Kowalski, F. Sadri, and P. Soper, "Integrity checking in deductive databases," in *Proceedings of the VLDB International Conference*. Morgan Kaufmann Publishers, 1987, pp. 61–69.

[18] J. Rubio-loyola, J. Serrat, M. Charalambides, P. Flegkas, and G. Pavlou, "A functional solution for goal-oriented policy refinement," in *Proc. 7th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, 2006.

[19] H. Zhao and S. M. Bellovin, "High performance firewalls in MANETs," in *The 6th International Conference on Mobile Ad-hoc and Sensor Networks (MSN'10)*, Hangzhou, P.R. China, December 2010.

[20] H. Zhao, C.-K. Chau, and S. M. Bellovin, "ROFL: Routing as the firewall layer," in *New Security Paradigms Workshop*, September 2008, a version is available as Technical Report CUCS-026-08.