Automated Translation of End User Policies for Usage Control Enforcement

Prachi Kumari, Alexander Pretschner

Automated Translation of End User Policies for Usage Control Enforcement

Prachi Kumari and Alexander Pretschner

Technische Universität München, Germany
{kumari, pretschn}@cs.tum.edu

Abstract. In existing implementations of usage control, policies have been specified at the implementation level by intelligent users who understand the technical details of the systems in place. However, end users who want to protect their data are not always technical experts. So they would like to specify policies in abstract terms which could somehow be translated in a format that technical systems understand. This paper describes a generic and automated policy derivation where end users can specify their usage control requirements in structured natural language sentences, from which system-understandable technical policies are derived and deployed without further human intervention.

1 Introduction

The problem of enforcing policies about what can and what must not happen to specific data items is ubiquitous. Different enforcement infrastructures, both generic [1] and specific [2], have been proposed. The configuration of these infrastructures requires technical *implementation-level policies* (ILPs). End users, however, often cannot directly specify ILPs but require a higher level of abstraction, that of *specification-level policies* (SLPs). The problem we tackle in this paper is the automatic derivation of ILPs from SLPs. We show how to translate SLPs written in OSL [3] to ILPs that come as ECA rules [4].

The object of data usage control enforcement is abstract *data* (photo, song etc.), that we need to distinguish from concrete technical representations called *containers* (specific files, dom elements etc.). Policies on data usually need to be interpreted as policies on *all representations* of that data. We therefore rely on a data flow model [5] that defines system states as mappings between data and containers at a moment in time. The transition relation on states is the change in these mappings. Policy enforcement on data is done on the basis of data flow tracking which is monitored on grounds of the transition relation [5].

Because data and actions on data are specific to an application domain, a three-layered domain meta-model for refining data and actions has been proposed in [6]. This meta-model (Fig. 1) describes a domain to be composed of classes of data and actions at the end user's level (platform-independent model: PIM), both of which are refined in terms of classes of technical concepts (platform-specific model: PSM) and classes of various implementations of them

(implementation-specific model: ISM) at the lower levels in the meta-model. The refinement of user action and data is given by the vertical mappings between the model elements (data classes mapped to container classes and action classes mapped to transformer classes which are further mapped to other classes of containers and transformers at the PSM and the ISM levels). Fig 2 (taken from [7]) shows an example instance of the meta-model in Fig 1 that refines "copy photo" in different ways in an Online Social Network (OSN).

The formal semantics of the refinements are given by combining the domain meta-model with the aforementioned data flow model. Policies are derived using this formal action refinement. The detailed formalism is described in [7].

Running Example. In an OSN implementation, user Alice wants to specify and enforce a policy "friends must never copy this data" for her photos, without any technical knowledge. We will refer to this SLP and the action refinement in Fig 2 throughout this paper. In §3, we describe the technicalities of one implementation that helps Alice prevent her friends from copying her photos.

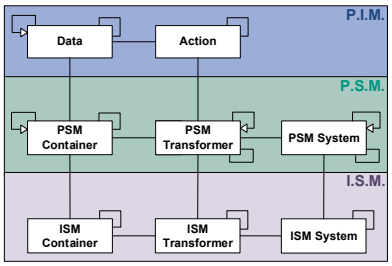


Fig. 1: The Domain Meta-model

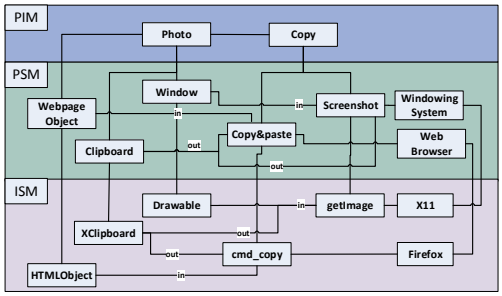


Fig. 2: An OSN instance of Fig 1

Contribution. This paper presents a generic methodology for automating policy derivation for usage control enforcement. A proof of concept is realized for a specific context (i.e. OSN). However, the approach is generic and can be applied to any other domain model.

2 The Generic Approach to Policy Derivation

Connecting Instances to Classes. The domain model specifies general *classes* of data. A specific policy is concerned with specific data items, which we need to connect to the classes in the domain model. Conceptually, we use a generic function *getClass* for this. The definition of the *getClass* function varies according to the application domain. We illustrate this with a simple example: In the OSN context, a specific image (data d) can be a profile photo, logo, banner or a cover photo (data classes). Different data classes might mean different storage and different technical refinements of the instances. E.g. (small) logos stored as blobs vs. (large) photos stored as links in a table. Relevant data classes and assignment of data elements to one or many of them is intuitive and is driven by the domain design and its implementation. This is because data classes describe how users understand different types of data in a particular context and there

is no universal rule for classifying them. However, for a given context, it is easy to define *getClass*. In §3 we show how this can be done for a particular case.

The Policy Derivation Methodology. Although end users would like to specify security requirements, in particular usage control policies [8], they are in general not capable of reasoning holistically about such policies [9]. This motivates the need for a policy specification and derivation framework that requires simple and limited user input for the specification of usage control policies. Our policy derivation approach is based on *policy templates* that are *classes of policies*. Human intervention is in two roles: a technical expert called the **power user** specifies the domain model and other configurations, using which, an **end user** specifies and automatically translates and deploys the policies. The power user only configures the policy derivation at the start, no further human intervention is needed. Policies are automatically derived in the following *five* steps:

Step 1: Setting up the Policy Derivation. The power user specifies the domain model used for action refinement. The power user also specifies two types of policy templates: a *first* set of templates for the SLPs, to be instantiated by an end user for each data element to be protected. We studied several application domains and recognized that most relevant usage control policies could be specified using limited combinations of OSL operators. Based on this, we came up with *classes of SLPs that specify constraints upon classes of data and actions*. As one SLP can be enforced in several ways [4], the *second* set of templates specifies enforcement *strategies* for each class of SLPs. E.g. a company enforces a policy "don't copy document" by inhibition because the company wants to prevent data leakage and its infrastructure supports it; or, a film producer enforces "don't play unpaid videos" by allowing corresponding events with a lower quality video because either it's technically not feasible to inhibit the events or he wants to give a limited preview to its prospective customers. As the reasons for deciding upon an enforcement strategy don't change very often, it is reasonable to specify the enforcement strategy for ECA rules generation in a template in order to automate the process.

Step 2: Policy Specification. End users instantiate the policy classes using templates to specify data elements, actions and other constraints like time, cardinality etc. (e.g. max. copies allowed, min. payable amount, currency etc.).

Step 3: Policy Derivation. First, we get the class of data addressed in the policy using *getClass*. Then, data and actions are refined according to the formal policy derivation described in [7] and the resulting OSL formulas are converted into ECA rules using predefined templates.

Step 4: Connecting Data and Container. As such, data does not exist in real world, except in the mind of the humans who want to protect or abuse it. In real systems, only corresponding containers exist. Therefore, an end user writes policies on containers and specifies if he means to address the data in that container (i.e. all copies of the container) or the container itself. This is done by specifying the *policy type* in the SLP. Conceptually, data enters a system in an *initial container* and gets a data ID assigned. This process of initially mapping containers to data is called the *initial binding of data and container*. After

the initial binding, policies that are finally deployed for enforcement are of two types: *dataOnly* policies apply to all representations of data at and across layers of abstraction in a machine while *containerOnly* policies apply to the specific container mentioned in the policy.

Step 5: Adding Context-Specific Details. According to the domain and its implementation, there are many context details that might be added to the ECA rules before deployment. E.g. policies in an OSN might address specific users and their relationships to the profile owner, and this information is known only at runtime by identifying the browsing session. Together, step 4 and 5 are called *policy instantiation* and they might be switched in order, depending upon the context detail to be added and the corresponding system implementation.

Generic Architecture: We build upon an existing generic usage control infrastructure with three main components: a *Policy Enforcement Point* (PEP), able to observe, intercept, possibly modify and generate events in the system; a *Policy Decision Point* (PDP), representing the core of the usage control monitoring logic; and a *Policy Information Point* (PIP), which provides the data-container mapping to the PDP [5]. This infrastructure was extended with a *Policy Management Point* (PMP) with a dedicated sub-component, the *Policy Translation Point* (PTP) for policy derivation using action refinement. Policy specification, instantiation and deployment is handled by the PMP.

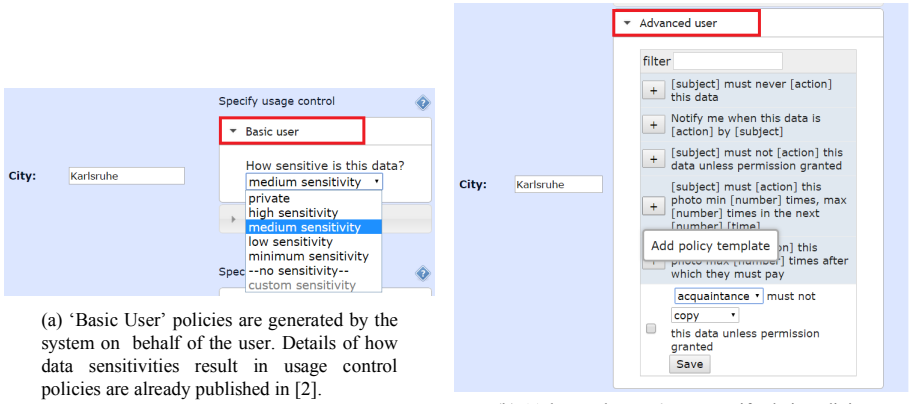
During policy enforcement, when an event is intercepted and notified by the PEP to the PDP, the object parameter is always a container as only concrete containers exist in running systems. For a *dataOnly* policy, the PDP queries the PIP for the data-container mapping before deciding upon the enforcement mechanism. In case of *containerOnly* policies, the PDP needs no communication with the PIP as the policy and the event notification address containers.

3 Implementation & Evaluation

We instantiated the generic architecture to derive policies for the online social network SCUTA, which already had basic usage control capabilities [2]. We extended SCUTA with two types of policy specification for two classes of end users. As shown in Fig. 3, *basic users* only specify how sensitive they consider a particular data. Based on the sensitivity rating and a predefined trust model, SLPs are generated by the system on behalf of the user (for details, see [2]). *Advanced Users* specify their policies by instantiating admin-defined SLP templates that are loaded every time the page is requested. In Fig. 3, we also see an SLP template in edit mode.

In our attacker model, all the policies *protect end users' data from misuse by other social network users*. The SCUTA provider is trusted. SLPs are specified by Alice. They are translated at the data receiver Bob's end, to be enforced at two layers of abstraction in the system: Mozilla Firefox web browser [2] and Windows 7 operating system (for protecting cache files) [10].[1]

[1] A video demonstrating policy specification and translation for this use case is available at https://www.youtube.com/watch?v=6i9Mfmbj2Xw.



(a) 'Basic User' policies are generated by the system on behalf of the user. Details of how data sensitivities result in usage control policies are already published in [2].

(b) 'Advanced Users' can specify their policies using admin-defined SLP templates

Fig. 3: Screenshot of SCUTA showing basic and advanced policy specification

Connecting instances to classes. In order to connect data elements to their classes, we used the database schema of SCUTA: the class of a data element is represented by the name of the table where that element is stored. E.g. an image stored in table "profile_photo" is a profile photo. As mentioned in §1, a data flow model is used to keep track of data in multiple containers. The data flow model of a particular system describes all sets of containers in that system. E.g., C_{File} is the set of all files in the data flow model of Windows 7. We reused this information in our implementation to connect containers to their classes, e.g., all members of C_{File} are of the type File.

Example Policy Derivation. Now we describe the derivation of our example policy introduced in §1: "friends must never copy this data". Action refinement is based on the domain model shown in Fig 2. The SLP templates (specified by the power user) and their instances (specified by end users) are stored in the SCUTA database. Figure 4 shows an example SLP template.

Template ID	102
Template Text	#{[subject]}# must never _[[action]]_ this data
Data Class	city;email;photo
Policy Class	<policy subject=\"#{subject}#\"><obligation><always><not><action name=\"_[action]_\" type=\"desiredEv\"><params><param class=\"@{class}@\" name=\"object\" policyType=\"dataUsage\" value=\"@{object}@\"/></params></action></not></always></obligation></policy>

Fig. 4: An example SLP template, specified by the power user

Each template is identified by an *ID*. The *template text* appears in the front end without the special character delimiters; the delimiters are used by the web application logic to create the GUI. *Data class* refers to the type of data for which the policy could be instantiated. *Policy class* is the XML representation of the template text with placeholders delimited by special symbols. In the GUI, the end user sees the template text with editable parts. When a template is instan-

tiated, data and other parameter values substitute the placeholders. Generated policies are shown to the user as structured sentences in natural language.

Policy derivation and deployment is a one-click process. ECA templates are specified per SLP template. Listing 1.1 shows the ECA template (as a JSON object) for the SLP template shown in Fig 4. Note how the template ID from Fig 4 is used to connect the two templates. This ECA template states that in the generated ECA rule, the trigger *event* is a *-event which matches all events in the system; the *condition* comes from the action refinement whose input is the past form of the specified OSL policy; the *action* is to inhibit the event.

```
{"id": "102",
 "templates":[
 {"event":"<*/>",
  "condition":"<actionRef(pastForm)
       />",
  "action":["<inhibit/>"],
  "type":"preventive"
 } ]  },
```

Listing 1.1: ECA generation template for "inhibit" enforcement strategy

```
{"id":"102",
 "templates":[
  {"event":"<*/>",
   "condition":"<actionRef(pastForm)
        />",
   "action":["<modify/>", "object",
        "icon/error.jpg"],
   "type":"preventive"
  } ] },
```

Listing 1.2: ECA generation template for "modify" enforcement strategy

Using this ECA template, ECA rules are generated automatically. An ECA rule for the Firefox web browser is shown in Listing 1.3. The scope in line 3 of Listing 1.3 is added to the policy as part of the context information that identifies each logged-in SCUTA user (and his browsing session) uniquely. "cmd_copy" in line 6 comes from the refinement of copy action (Fig 2). In line 7, "dataUsage" means that the container "9c73d9b7ff.jpg" is to be interpreted as data and substituted by a data ID.

```
1  <preventiveMechanism name="Mechanism_102_1_preventive">
2    <trigger action="*" tryEvent="true">
3      <paramMatch name="scope" value="536a624a87644"/>
4    </trigger>
5    <condition>
6      <eventMatch action="cmd_copy" tryEvent="true">
7        <paramMatch name="object" value="9c73d9b7ff.jpg" type="dataUsage"/>
8      </eventMatch>
9    </condition>
10   <authorizationAction name="Authorization␣1">
11     <inhibit/>
12   </authorizationAction>
13 </preventiveMechanism>
```

Listing 1.3: ECA rule for "inhibit copy" in Firefox web browser

Listing 1.2 shows another ECA template that generates ECA rules with "modify" enforcement strategy for the SLP template of Fig 4: the *action* part says that the enforcement mechanism must replace the object in question by "icon/error.jpg". One of the generated ECA rules is shown in Listing 1.4.

```
1  <preventiveMechanism name="Mechanism_102_1_preventive">
2    <trigger action="*" tryEvent="true">
3      <paramMatch name="scope" value="536a62466a42e"/>
4    </trigger>
```

```
5   <condition>
6     <eventMatch action="cmd_copy" tryEvent="true">
7       <paramMatch name="object" value="9c73d9b7ff.jpg" type="dataUsage"/>
8     </eventMatch>
9   </condition>
10  <authorizationAction name="Authorization␣1">
11    <allow>
12      <modify>
13        <parameter name="object" value="icon/error.jpg"/>
14      </modify>
15    </allow>
16  </authorizationAction>
17 </preventiveMechanism>
```

Listing 1.4: ECA rule for "modify copy" in Firefox web browser

As we use templates for ECA rules generation, changing enforcement strategies is easy. The power user modifies the ECA templates and notifies the PMP for retranslating, revoking and deploying policies. In the current implementation, all deployed policies are overwritten. Modifying the PMP to selectively revoke and redeploy only those policies whose enforcement strategy has changed is trivial.

4 Related Work & Relevance

Policy derivation for access control has been investigated based on resource hierarchies [11], goal decomposition [12], action decomposition [13], data classification [14], ontology [15], etc. In **usage control**, several enforcements exist at and across different layers of abstraction in various types of systems [2,3,5,16–18]. In all these implementations, the focus has been on the implementation of event monitors; policy derivation has not been addressed. While [19] describes an approach where high-level usage control policies are automatically refined to implementation-level policies using policy refinement rules, the refinement rules are specific to the domain in consideration. In contrast, the very idea of our work is a more generic approach that relies on instantiated domain meta models; this is the gap in usage control policy refinement that is filled by this work. A model-based semi-automated approach to usage control policy derivation is described in [6, 7]. We build on this work and achieve automated policy derivation.

5 Conclusions & Future Work

This paper provides a methodology for automating usage control policy derivation. We have also proposed the usage of policy templates in order to enable end users to specify usage control policies without having to know the underlying policy language. We do not propose any concrete syntax or design for the templates. We have *deliberately* kept our implementation of the templates simple with user inputs limited to multiple-choice format. In principle, policy templates could be written in XML, in plain structured text, in simple graphical format with dropdowns and checkboxes or, in advanced graphical format by dragging and connecting different blocks to write policies (along the lines of MIT Open

Blocks). Policies could also be specified using free text in any natural language or standardized solutions like ODRLC (a translation from OSL to ODRLC and vice versa is shown in [3]). Also, technically-proficient users can write policies directly at the implementation level. This work does not exclude such possibility.

One limitation of this work is the static domain structure. This is a deliberately introduced limitation which does not affect the automation of policy derivation, which was the goal of this work. This limitation is being addressed in ongoing work and is therefore out of the scope of this paper.

References

1. F. Kelbert and A. Pretschner. Towards a policy enforcement infrastructure for distributed usage control. In *Proc. SACMAT '12*, pages 119–122.
2. P. Kumari, A. Pretschner, J. Peschla, and J. Kuhn. Distributed data usage control for web applications: a social network implementation. CODASPY '11.
3. M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. A policy language for distributed usage control. In *Proc. ESORICS*, pages 531–546, 2007.
4. A. Pretschner, M. Hilty, D. Basin, C. Schaefer, and T. Walter. Mechanisms for Usage Control. In *Proc. ASIACCS*, pages 240–245, 2008.
5. A. Pretschner, E. Lovat, and M. Buechler. Representation-independent data usage control. In *Proc. 6th Intl. Workshop on Data Privacy Management*, 2011.
6. P. Kumari and A. Pretschner. Deriving implementation-level policies for usage control enforcement. CODASPY '12, pages 83–94. ACM, 2012.
7. P. Kumari and A. Pretschner. Model-based usage control policy derivation. In *Proc. ESSOS 2013*, pages 58–74. 2013.
8. M. Rudolph, R. Schwarz, C. Jung, A. Mauthe, and N. Shirazi. Deliverable 3.2 - policy specification methodology. Technical report, SECCRIT, June 2014.
9. Lujun Fang and Kristen LeFevre. Privacy wizards for social networking sites. In *Proc. WWW '10*, pages 351–360, New York, NY, USA, 2010. ACM.
10. T. Wüchner and A Pretschner. Data loss prevention based on data-driven usage control. In *Proc. ISSRE 2012*, pages 151–160, Nov 2012.
11. L. Su, D. Chadwick, A. Basden, and J. Cunningham. Automated decomposition of access control policies. In *Proc. POLICY 2005*, pages 6–8.
12. A.K. Bandara, E.C. Lupu, J. Moffett, and A. Russo. A goal-based approach to policy refinement. In *Proc. POLICY 2004*, pages 229–239.
13. R. Craven, J. Lobo, E. Lupu, A. Russo, and M. Sloman. Decomposition techniques for policy refinement. In *Proc CNSM '10*, pages 72 –79, 2010.
14. Y.B. Udupi, A. Sahai, and S. Singhal. A classification-based approach to policy refinement. In *Proc. 10th IFIP/IEEE IM*, 2007.
15. A. Guerrero, V.A. Villagrá, J.E. López de Vergara, A. Sánchez-Macián, and J. Berrocal. Ontology-based policy refinement using swrl rules for management information definitions in owl. In *DSOM*, pages 227–232, 2006.
16. M. Harvan and A. Pretschner. State-based Usage Control Enforcement with Data Flow Tracking using System Call Interposition. In *Proc. NSS*, pages 373–380, 2009.
17. A. Pretschner, M. Buechler, M. Harvan, C. Schaefer, and T. Walter. Usage control enforcement with data flow tracking for x11. In *Proc. STM 2009*, pages 124–137.
18. P. Kumari, F. Kelbert, and A. Pretschner. Data Protection in Heterogeneous Distributed Systems: A Smart Meter Example. In *DSCI*, 2011.
19. R. Neisse and J. Doerr. Model-based specification and refinement of usage control policies. In *Proc. PST'2013*, pages 169–176.