Representation-Independent Data Usage Control

Alexander Pretschner, Enrico Lovat, Matthias Büchler

Karlsruhe Institute of Technology, Germany
{pretschner,lovat,buechler}@kit.edu

Abstract. Usage control is concerned with what happens to data after access has been granted. In the literature, usage control models have been defined on the grounds of *events* that, somehow, are related to data. In order to better cater to the dimension of data, we extend a usage control model by the explicit distinction between *data* and *representation* of data. A data flow model is used to track the flow of data in-between different representations. The usage control model is then extended so that usage control policies can address not just one single representation (e.g., delete file1.txt after thirty days) but rather all representations of the data (e.g., if file1.txt is a copy of file2.txt, also delete file2.txt). We present three proof-of-concept implementations of the model, at the operating system level, at the browser level, and at the X11 level, and also provide an ad-hoc implementation for multi-layer enforcement.

1 Introduction

If usage control requirements are to be enforced on data, one must take into account that this data exists in multiple representations. For instance, there can be multiple copies of a file, or multiple clones of an object. Similarly, an image can exist as network packet, Java object, window pixmap, data base record, or file. The representations potentially reside at different system layers, including operating system, runtime system, window manager, and DBMS. High-level usage control requirements such as "don't copy" have different meanings at different layers (copy a file, take a screenshot, duplicate a database record, copy&paste in a word processor). While in principle, it is possible to enforce these requirements at the level of CPU instructions, it turns out to be hard to identify, *in general*, precisely those instructions that pertain to copying a file, taking a screenshot, etc. Therefore, we consider it convenient to simultaneously enforce usage control requirements at all relevant system layers. This, however, requires following the flow of data from one representation to another within and across system layers.

We present a framework and its implementation for combining usage control enforcement with data flow tracking technology. One example of the resulting system is a social network in which users may view pictures in their browsers (first representation, first layer) but not copy cache files (second representation, second layer) or take screenshots (third representation, third layer) [1]. We describe the model and its prototypical implementation; detailed security and performance analyses are not in the scope. We organize our paper along six steps. *Step 1: Specification-level usage control policies based on events* We start with a policy language from the literature [2] that allows us to state requirements on

future events ("at most three copies," "whenever data is accessed, notify me," "don't delete for five years," "do delete after thirty days"). In this model, usage control policies are interpreted as sets of allowed sequences of sets of events. We call these policies *specification-level policies*.

Step 2: Data and containers; data state In order to cater to the dimension of data, we distinguish between data and containers. Containers (files, pixmaps, memory regions, network packets) reflect different representations of data. This is captured by the *data state* $\sigma \in \Sigma$ of a system which essentially maps containers to sets of data items. Independent of any policies, the data state of a system changes with every step of the system. We capture this by a transition function ϱ that maps a data state and a set of events to another data state. This data flow model has been described and instantiated to various levels of the system before [3–5]. In this paper, we embody the data flow model in a usage control policy language and an integrated semantic model; together with the prototype implementation, this constitutes the core contribution of this paper.

Step 3: Specification-level usage control policies based on data In the language of step 1, we can only express container usages, i.e., usage events that pertain to *one specific representation*. Since we deem it natural to express *data usages* which pertain to all representations of the same data as well, we augment the language by (1) data usages and (2) special operators for data rather than containers—e.g., some *data* may not flow into a specific container such as a network socket.

Step 4: Implementation-level policies based on data Specification-level policies are enforced by mechanisms that are configured by *implementation-level* policies. Implementation-level policies are event-condition-action (ECA) rules that perform an action provided that a trigger event has happened and the respective condition has evaluated to true. The action can be to inhibit or to modify the trigger event (which requires the distinction between desired and actual events) or to execute some other event (which does not require this distinction). Since these mechanisms are actually implemented, it is convenient to express the condition part of the ECA rules in a language that expresses requirements on the past rather than on the future. This language then is the natural past dual of the language of step 1 [6]. In this paper, we also augment this implementation-level policy language by data usages and special state-based operators.

Step 5: Runtime monitors for events and data Using our specification-level (future) and implementation-level (past) languages, we can leverage results from runtime monitoring to synthesize efficient monitors both for specification-level and (the condition part of) implementation-level policies. In the first case, we can detect violations (detective enforcement) whereas in the second case, we can also prevent violations from happening by blocking or modifying attempted events, and by performing compensating, penalizing, or notifying actions. Efficient runtime monitoring technology is readily available [7].

It is straightforward to implement the evolution of the data state. At each moment in time, we intercept the current event and update the data state by consulting the transition function ϱ. This simple implementation yields a state machine that computes the data state extraction function *states*.

In terms of the combined model, if a *data usage* is specified in a policy (and thus in the synthesized monitor), we consult the state machine that implements the information state $\sigma \in \Sigma$ from within the usage control monitor to retrieve all the containers that contain the respective data item, and evaluate the policy w.r.t. *all these containers*. Function *states* is independent of any given policy; since our framework is intended to be deployed at different system layers, there hence is one data state tracker per system layer, and one runtime monitor per layer per policy. While pure usage control monitors [8] as well as data flow tracking systems [3–5] have been implemented before, we provide implementations of *combined* data flow tracking and usage control enforcement mechanisms here.

Step 6: Multi-Layer enforcement As the above example of the social network application shows, data representations may exist at several different layers of abstraction (cache file, pixmap, web page content), and we must track the flow of data and enforce usage control requirements not only at single layers of the system, but also across different layers. In full generality, this problem is still subject of investigation. However, tailored solutions are possible. For instance, in this paper, we propose as an example an ad-hoc solution for a multi-layer instantiation in a social network scenario: we show that it is possible to enforce usage control requirements with data flow tracking for a picture when this is represented as a browser object, as a cache file and as content of a window.

Research Problem In sum, we tackle the problem of how to do usage control on data that exists in multiple representations at different system layers.

Solution We present, firstly, a formal model that extends one usage control model by the notion of data representations and that hence allows us to track data flows within and in-between different representations at different layers of the system. Secondly, as a proof of concept, we show how to implement such a system. We do not present a security analysis, and we do not claim that our implementation cannot be circumvented [9]).

Contribution Data flow tracking at specific system layers has been done in a multitude of ways [3, 4, 10–19], also in the form of information flow analyses where implicit flows are also taken into account [20, 21]. As far as we are aware, this work ends where sensitive (or tainted) data is moved to illegal sinks, e.g., when a file is written or an http post request is sent. If such an illegal sink is reached, something bad has happened, and an exception is thrown. In contrast, our work adds the dimension of usage control that allows to specify and enforce more fine-grained constraints on these sinks. Conversely, usage control models are usually defined on the grounds of technical events, including specific technologies such as complex event processing or runtime verification [22, 7], but do not cater to the flow of data. We add the distinction between representation and data to these models. We see our contribution in the marriage of the research areas of usage control and dynamic data flow tracking.

Organization §2 recapitulates (1) a semantic model and a language for usage control and (2) a semantic model for data flow. §3 presents our combined model. §4 describes three different instantiations as well as an ad-hoc multi-layer enforcement implementation. §5 puts our work in context, and §6 concludes. An extended version of this paper is available as a technical report [23].

2 Background

In this section, we recapitulate the specification-level policy specification language [2] and the data flow model [3–5] that we will combine in §3.

Step 1: Usage Control We consider a usage control system model [2] based on classes of parameterized events where parameters represent attributes. Every event in set $Event \subseteq EventName \times Params$ consists of the event's name and parameters, represented as a partial ($\nrightarrow$) function from names to values: $Params \subseteq ParamName \nrightarrow ParamValue$ for basic types $ParamName$, $ParamValue$, $EventName$. We denote event parameters by their graph, i.e., as $(name, value)$ pairs. We assume a reserved parameter name, *obj*, to indicate on which data item the event is performed. An example is the event $(show, \{(obj, x)\})$, where *show* is the name of the event and the parameter *obj* has value x. We reserve a Boolean parameter *isTry* which indicates if the event is desired or actual (this is necessary if events should be blocked or modified in order to enforce policies) [6].

In policies, events are usually under-specified. For instance, if a policy specifies that event $(show, \{(obj, x)\})$ is prohibited, then the actual event $(show, \{(obj, x), (window, w)\})$ should also be prohibited. For this reason, events are partially ordered with respect to a refinement relation *refinesEv*. Event e_2 refines event e_1 iff e_2 has the same event name as e_1 and all parameters of e_1 have the same value in e_2. e_2 can also have additional parameters specified, which explains the subset relation in the definition. Let $x.i$ identify the i-th component of a tuple x. Formally, we then have $refinesEv \subseteq Event \times Event$ with $\forall\, e_1, e_2 \in Event \bullet e_2\ refinesEv\ e_1 \Leftrightarrow e_1.1 = e_2.1 \wedge e_1.2 \subseteq e_2.2$. In the semantic model, we will assume traces to be maximally refined (all parameters carry values; this seems natural in an actually running system): $maxRefinedEv = \{e \in Event : \forall\, e' \in Event \bullet e'\ refinesEv\ e \Rightarrow e' = e\}$. The semantics of the usage control policy language is defined over traces. Traces map abstract points in time —the natural numbers— to possibly empty sets of maximally refined actual and desired events: $Trace : \mathbb{N} \to \mathbb{P}(maxRefinedEv)$.

Specification-level usage control policies are then described in language Φ^+ (+ for future). It is a temporal logic with explicit operators for cardinality and permissions where the cardinality operators turn out to be mere macros [24], and where we omit the permission operator for brevity's sake. We distinguish between purely propositional (Ψ) and temporal and cardinality operators (Φ^+).

$$\Psi ::= \underline{true} \mid \underline{false} \mid E(Event) \mid T(Event) \mid \underline{not}(\Psi) \mid \underline{and}(\Psi, \Psi) \mid \underline{or}(\Psi, \Psi) \mid \underline{implies}(\Psi, \Psi)$$
$$\Phi^+ ::= \Psi \mid \underline{not}(\Phi^+) \mid \underline{and}(\Phi^+, \Phi^+) \mid \underline{or}(\Phi^+, \Phi^+) \mid \underline{implies}(\Phi^+, \Phi^+) \mid$$
$$\underline{until}(\Phi^+, \Phi^+) \mid \underline{after}(\mathbb{N}, \Phi^+) \mid \underline{within}(\mathbb{N}, \Phi^+) \mid \underline{during}(\mathbb{N}, \Phi^+) \mid$$
$$\underline{always}(\Phi^+) \mid \underline{repmax}(\mathbb{N}, \Psi) \mid \underline{replim}(\mathbb{N}, \mathbb{N}, \mathbb{N}, \Psi) \mid \underline{repuntil}(\mathbb{N}, \Psi, \Phi^+)$$

We also distinguish between *desired* or *attempted* (T) and *actual* (E) events. These syntactically reflect the (semantic-level) parameter *isTry* introduced above. The semantics of events is captured by relation $\models_\varepsilon \subseteq Event \times \Phi^+$ that relates events (rather than traces) to formulae of the form $E(\cdot)$ or $T(\cdot)$ as follows:
$\forall\, e, e' \in Event \bullet e \models_\varepsilon E(e') \Leftrightarrow e\ refinesEv\ e' \wedge e.2(isTry) = false$ and
$\forall\, e, e' \in Event \bullet e \models_\varepsilon T(e') \Leftrightarrow e\ refinesEv\ e' \wedge e.2(isTry) = true$.

not, *and*, *or*, *implies* have the usual semantics. The *until* operator is the weak-until operator from LTL. Using *after*(n), which refers to the time after n time steps, we can express concepts like *during* (something must constantly hold during a specified time interval) and *within* (something must hold at least once during a specified time interval). Cardinality operators restrict the number of occurrences or the duration of an action. The *replim* operator specifies lower and upper bounds of times within a fixed time interval in which a given formula holds. The *repuntil* operator does the same, but independent of any time interval. Instead, it limits the maximal number of times a formula holds until another formula holds (e.g., the occurrence of some event). With the help of *repuntil*, we can also define *repmax*, which defines the maximal number of times a formula may hold in the indefinite future. As an example of a cardinality operator, *replim*$(100, 0, 3, E((login, \{(user, Alice), (obj, \varnothing)\})))$ specifies that user Alice may login at most 3 times in the next 100 time units.

Step 2: Data Flow Tracking We base our work on data flow tracking on approaches from the literature [3–5]. In this model, data flow is defined by a transition relation on states that essentially map data representations, so-called *containers*, to data. Transitions are triggered by principals that perform actions. Formally, we describe systems as tuples $(P, Data, Event, Container, \Sigma, \sigma_i, \varrho)$ where P is a set of principals, *Data* is a set of data elements, *Event* is the set of events (or actions), *Container* is a set of data containers, Σ is the set of states of the system with σ_i being the initial state $(\varnothing, \varnothing, \varnothing)$, and ϱ is the state transition function. In the following, we assume that the principals executing actions (making an event happen) are provided as a parameter of the action.

States are defined by three mappings (for simplicity's sake, we concentrated on just one mapping in the introduction): a *storage function* of type $Container \to \mathbb{P}(Data)$, to know which set of data is stored in which container; an *alias function* of type $Container \to \mathbb{P}(Container)$ that captures the fact that some containers may implicitly get updated whenever other containers do; and a *naming function* that provides names for containers and that is of type $F \to Container$. F is a set of identifiers. We need identifiers to correctly model renaming activities. We thus define $\Sigma = (Container \to \mathbb{P}(Data)) \times (Container \to \mathbb{P}(Container)) \times (F \to Container)$. We define transitions between two states by $\varrho : \Sigma \times \mathbb{P}(Event) \to \Sigma$. For simplicity's sake, in this paper, we assume independent actions only. This means that if $(\sigma, E) \in \varrho$, then the target state of this transition is independent of the ordering in which the actions in E are executed in an actual implementation. In real systems, however, is usually possible to sort events within the same timestep (e.g. by timestamp), hence this assumption is, in general, not restrictive.

3 A Combined Model

In the usage control model of § 2, data is addressed by referring to its specific representations as event parameters. For instance, *after*(30, *always*(*not*(*E*(*play*, {(*obj,song1.mp3*)}))))) stipulates that a file (a specific representation and a specific container) called *song1.mp3* must not be played after thirty days. We address the situation where a copy of that file, *song2.mp3*, should not be played either. To this end, we extend the semantic model by *data usages* that allow us

to specify protection requirements for all representations rather than just one. Using the data flow tracking model, we compute, at each moment in time t, the current data state of the system: we simply take the usage control model's system trace until t, extract the respective events in each step, iteratively compute the successor data states for each data state and eventually get the data state at time t. In an implementation, we will not store the system history but rather use state machines to record the data state at each moment in time (step 5).

Data, Containers, and Events We need to distinguish between *data* items and *containers* for data items. At the specification level, this leads to the distinction between two classes of events according to the "type" of the *obj* parameter: events of class *dataUsage* define actions on data objects. The intuition is that these pertain to *every representation*. In contrast, events of class *containerUsage* refer to one single container. In a real system, only events of class *containerUsage* can happen. This is because each monitored event in a trace is related to a specific representation of the data (a file, a memory region, etc). *dataUsage* events are used only in the definition of policies, where it is possible to define a rule abstracting from the specific representation of a data item. We define a function *getclass* that extracts if an event is a data or a container usage.

$$EventClass = \{dataUsage, containerUsage\} \qquad getclass : Event \to EventClass$$
$$Data \cup Container \subseteq ParamValue \qquad \{(obj, d) \mid d \in Data\} \subseteq Params$$
$$Container \cap Data = \varnothing \qquad \{(obj, c) \mid c \in Container\} \subseteq Params$$
$$\forall\, e : Event \bullet getclass(e) = dataUsage \Leftrightarrow \exists\, x : ParamValue \bullet ((obj, x) \in e.2) \wedge x \in Data$$
$$\wedge\ getclass(e) = containerUsage \Leftrightarrow \exists\, x : ParamValue \bullet ((obj, x) \in e.2) \wedge x \in Container$$

Step 3: Adding Data State In our semantic model, policies are defined on traces. We want to describe certain situations to be avoided or enforced. In practice there usually is an almost infinite number of different sequences of events that lead to the same situation, e.g., the creation of a copy or the deletion of a file. Instead of listing all these sequences, it appears more convenient in situations of this kind to define a policy based on the description of the (data flow state of the) system at that specific moment. To define such formulas we introduce a new set of *state-based operators*, $\Phi_i ::= \underline{isNotIn}(Data, \mathbb{P}\, Container) \mid \underline{isCombinedWith}(Data, Data) \mid \underline{isOnlyIn}(Data, \mathbb{P}\, Container)$ and define $\Phi_i^+ ::= \Phi^+ \mid \Phi_i$. Intuitively, $\underline{isNotIn}(d, C)$ is true if data d is not present in any of the containers in set C. This is useful to express constraints such as "song s must not be distributed over the network", which becomes $\underline{always}(\underline{isNotIn}(s, \{c_{net}\}))$ for a network container (any socket) c_{net}. The rule $\underline{isCombinedWith}(d_1, d_2)$ states whether data items d_1 and d_2 are combined in one container. This is useful to express Chinese Wall policies. $\underline{isOnlyIn}(d, C)$ is syntactic sugar for $\underline{isNotIn}(d, Container \setminus C)$ and expresses that data d can only be in containers of set C, e.g., $\underline{isOnlyIn}(d, \varnothing)$ for "data d has been deleted."

We have seen above that we implicitly quantify over unmentioned parameters when specifying events in policies by using relation *refinesEv*. We now extend this definition to *dataUsages*. An event of class *dataUsage* is refined by an event of class *containerUsage* if the latter is related to a specific representation of the

data the former refers to. As in the original definition, in both cases the more refined event may have more parameters than the more abstract event. An event e_2 refines an event e_1 if (1) e_1 and e_2 both have the same class (*containerUsage* or *dataUsage*) and we have e_2 *refinesEv* e_1; or (2) if e_1 is a *dataUsage* and e_2 a *containerUsage* event. In this latter case, e_1 and e_2 must have the same event name, and there must exist a data item d stored in a container c such that $(obj, d) \in e_1.2$; $(obj, c) \in e_2.2$; all parameters (except for obj) of e_1 have the same value in e_2; and e_2 can possibly have additional parameters. Formally, these requirements are specified by relation $\mathit{refinesEv}_i \subseteq (\mathit{Event} \times \Sigma) \times \mathit{Event}$, which checks whether one event e_2 refines another event e_1 also w.r.t. data and containers (Σ is needed to access the current information state):

$$\begin{aligned}
&\forall\, e_1, e_2 \in \mathit{Event};\ \sigma \in \Sigma \bullet (e_2, \sigma)\ \mathit{refinesEv}_i\ e_1 \Leftrightarrow \\
&\quad (\mathit{getclass}(e_1) = \mathit{getclass}(e_2) \wedge e_2\ \mathit{refinesEv}\ e_1) \\
&\quad \vee ((\mathit{getclass}(e_1) = \mathit{dataUsage} \wedge \mathit{getclass}(e_2) = \mathit{containerUsage} \wedge e_1.1 = e_2.1 \\
&\qquad \wedge \exists\, d \in \mathit{Data}, c \in \mathit{Container} \bullet d \in \sigma.1(c) \\
&\qquad \wedge e_1.2(\mathit{obj}) = d \wedge e_2.2(\mathit{obj}) = c \wedge (e_1.2 \backslash \{(\mathit{obj}, d)\} \subseteq e_2.2 \backslash \{(\mathit{obj}, c)\})))
\end{aligned}$$

With the help of $\mathit{refinesEv}_i$, we now define the satisfaction relation for event expressions in the context of data and container usages. We simply add one argument to $\models_\varepsilon$ and obtain $\models_{\varepsilon,i} \subseteq (\mathit{Event} \times \Sigma) \times \Phi_i^+$ as follows:

$$\begin{aligned}
\forall\, e, e' \in \mathit{Event}, \sigma \in \Sigma \bullet (e, \sigma) \models_{\varepsilon,i} E(e') \Leftrightarrow (e, \sigma)\ \mathit{refinesEv}_i\ e' \wedge e.2(\mathit{isTry}) = \mathit{false} \\
\wedge\, (e, \sigma) \models_{\varepsilon,i} T(e') \Leftrightarrow (e, \sigma)\ \mathit{refinesEv}_i\ e' \wedge e.2(\mathit{isTry}) = \mathit{true}
\end{aligned}$$

As a last ingredient, we need function $\mathit{states} : (\mathit{Trace} \times \mathbb{N}) \to \Sigma$ to compute the information state at a given moment in time via $\mathit{states}(t, 0) = \sigma_i$ and $n > 0 \Rightarrow \mathit{states}(t, n) = \varrho(\mathit{states}(t, n-1), t(n-1))$. On these grounds, we finally define the semantics of the specific data usage operators in Φ_i with semantics $\models_i \subseteq (\mathit{Trace} \times \mathbb{N}) \times \Phi_i$:

$$\begin{aligned}
&\forall\, t \in \mathit{Trace};\ n \in \mathbb{N};\ \varphi \in \Phi_i;\ \sigma \in \Sigma \bullet\ (t, n) \models_i \varphi \Leftrightarrow \sigma = \mathit{states}(t, n) \wedge \\
&\exists\, d \in \mathit{Data}, C \in \mathbb{P}\, \mathit{Container} \bullet \varphi = \underline{\mathit{isNotIn}}(d, C) \wedge \forall\, c' \in \mathit{Container} \bullet d \in \sigma.1(c') \Rightarrow (c' \notin C) \\
&\vee \exists\, d_1, d_2 \in \mathit{Data} \bullet \varphi = \underline{\mathit{isCombinedWith}}(d_1, d_2) \wedge \exists\, c' \in \mathit{Container} \bullet d_1 \in \sigma.1(c') \wedge d_2 \in \sigma.1(c')
\end{aligned}$$

This leads to the definition of the semantics augmented by data flow, $\models_i^+ \subseteq (\mathit{Trace} \times \mathbb{N}) \times \Phi_i^+$ depicted in Figure 1. The definitions for the cardinality operators are complex because of the refinement relation: it is possible that two simultaneously happening events e_1, e_2 that both refine the same event e both make $E(e) \in \Psi$ true. For a trace t, it is thus not sufficient to simply count those moments in time, n, that satisfy $(t, n) \models_i^+ E(e)$ [2, 6].

Step 4: Mechanisms enforce specification-level policies Specification-level policies expressed in Φ_i^+ describe which runs of a system are allowed and which ones are not. There are usually several ways of enforcing such policies, by modification, inhibition, or execution [19]. Since there is not the one right choice, a user must explicitly stipulate this by selecting an operational mechanism. These operational mechanisms embody *implementation-level policies* and are conveniently expressed as event-condition-action (ECA) rules [6]; whether or not satisfaction of an implementation-level usage control policy entails enforcement of

$$
\begin{aligned}
&\forall\, t \in \mathit{Trace}, n \in \mathbb{N}, \varphi \in \Phi_i^+ \bullet (t,n) \models_i^+ \varphi \Leftrightarrow \\
&\exists\, e, e' \in \mathit{Event} \bullet (\varphi = E(e) \vee \varphi = T(e)) \wedge e' \in t(n) \wedge (e', \mathit{states}(t,n)) \models_{\varepsilon,i} \varphi \\
&\vee\ \varphi \in \Phi_i \wedge (t,n) \models_i \varphi \\
&\vee\ \exists\, \psi \in \Phi_i^+ \bullet \varphi = \underline{\mathit{not}}(\psi) \wedge \neg\, ((t,n) \models_i^+ \psi) \\
&\vee\ \exists\, \psi, \chi \in \Phi_i^+ \bullet \varphi = \underline{\mathit{or}}(\psi,\chi) \wedge ((t,n) \models_i^+ \psi \vee (t,n) \models_i^+ \chi) \\
&\vee\ \exists\, \psi, \chi \in \Phi_i^+ \bullet \varphi = \underline{\mathit{until}}(\psi,\chi) \\
&\qquad \wedge (\exists\, u \in \mathbb{N} \bullet (((t, n+u) \models_i^+ \chi \wedge (\forall\, v \in \mathbb{N} \bullet v < u \Rightarrow (t, n+v) \models_i^+ \psi)) \\
&\qquad\qquad \vee \forall\, v \in \mathbb{N} \bullet (t, n+v) \models_i^+ \psi)) \\
&\vee\ \exists\, i \in \mathbb{N};\ \psi \in \Phi_i^+ \bullet \varphi = \underline{\mathit{after}}(i,\psi) \wedge (t, n+i) \models_i^+ \psi \\
&\vee\ \exists\, l, x, y \in \mathbb{N};\ \psi \in \Psi \bullet \varphi = \underline{\mathit{replim}}(l,x,y,\psi) \\
&\qquad \wedge\ x \leq \textstyle\sum_{j=1}^{l} \big|\{S \subseteq \mathit{Event} \mid S \subseteq t(n+j) \wedge \exists\, t' \in \mathit{Trace}\, \forall\, m \in \mathbb{N} \bullet \\
&\qquad\qquad\qquad t'(n+j) = S \wedge (m < n+j \Rightarrow t'(m) = t(m)) \wedge (t', n+j) \models_i^+ \psi \\
&\qquad\qquad\qquad \wedge \nexists\, S' \subseteq \mathit{Event} \bullet S' \subset S \wedge \exists\, t' \in \mathit{Trace}\, \forall\, m \in \mathbb{N} \bullet \\
&\qquad\qquad\qquad t'(n+j) = S' \wedge (m < n+j \Rightarrow t'(m) = t(m)) \wedge (t', n+j) \models_i^+ \psi\}\big| \leq y \\
&\vee\ \exists\, l, u \in \mathbb{N};\ \psi \in \Psi;\ \chi \in \Phi^+ \bullet \varphi = \underline{\mathit{repuntil}}(l,\psi,\chi) \\
&\qquad \wedge\ \big((t, n+u) \models_i^+ \chi \wedge \forall\, v \in \mathbb{N} \bullet v < u \Rightarrow \neg((t, n+v) \models_i^+ \chi) \\
&\qquad\qquad \wedge \textstyle\sum_{j=1}^{u} \big|\{S \subseteq \mathit{Event} \mid S \subseteq t(n+j) \wedge \exists\, t' \in \mathit{Trace}\, \forall\, m \in \mathbb{N} \bullet \\
&\qquad\qquad\qquad t'(n+j) = S \wedge (m < n+j \Rightarrow t'(m) = t(m)) \wedge (t', n+j) \models_i^+ \psi \\
&\qquad\qquad\qquad \wedge \nexists\, S' \subseteq \mathit{Event} \bullet S' \subset S \wedge \exists\, t' \in \mathit{Trace}\, \forall\, m \in \mathbb{N} \bullet \\
&\qquad\qquad\qquad t'(n+j) = S' \wedge (m < n+j \Rightarrow t'(m) = t(m)) \wedge (t', n+j) \models_i^+ \psi\}\big| \leq l\big) \\
&\qquad \vee \textstyle\sum_{j=1}^{\infty} \big|\{S \subseteq \mathit{Event} \mid S \subseteq t(n+j) \wedge \exists\, t' \in \mathit{Trace}\, \forall\, m \in \mathbb{N} \bullet \\
&\qquad\qquad\qquad t'(n+j) = S \wedge (m < n+j \Rightarrow t'(m) = t(m)) \wedge (t', n+j) \models_i^+ \psi \\
&\qquad\qquad\qquad \wedge \nexists\, S' \subseteq \mathit{Event} \bullet S' \subset S \wedge \exists\, t' \in \mathit{Trace}\, \forall\, m \in \mathbb{N} \bullet \\
&\qquad\qquad\qquad t'(n+j) = S' \wedge (m < n+j \Rightarrow t'(m) = t(m)) \wedge (t', n+j) \models_i^+ \psi\}\big| \leq l \\
&\vee\ \exists\, \psi, \chi \in \Phi_i^+ \bullet \varphi = \underline{\mathit{and}}(\psi,\chi) \wedge (t,n) \models_i^+ \underline{\mathit{not}}(\underline{\mathit{or}}(\underline{\mathit{not}}(\psi), \underline{\mathit{not}}(\chi))) \\
&\vee\ \exists\, \psi, \chi \in \Phi_i^+ \bullet \varphi = \underline{\mathit{implies}}(\psi,\chi) \wedge (t,n) \models_i^+ \underline{\mathit{or}}(\underline{\mathit{not}}(\psi), \chi) \\
&\vee\ \exists\, \psi \in \Phi_i^+ \bullet \varphi = \underline{\mathit{always}}(\psi) \wedge (t,n) \models_i^+ \underline{\mathit{until}}(\psi, \underline{\mathit{false}}) \\
&\vee\ \exists\, i \in \mathbb{N};\ \psi \in \Phi_i^+ \bullet \varphi = \underline{\mathit{within}}(i,\psi) \wedge (t,n) \models_i^+ \textstyle\bigvee_{x=1}^{i} \underline{\mathit{after}}(i,\varphi) \\
&\vee\ \exists\, i \in \mathbb{N};\ \psi \in \Phi_i^+ \bullet \varphi = \underline{\mathit{during}}(i,\psi) \wedge (t,n) \models_i^+ \textstyle\bigwedge_{x=1}^{i} \underline{\mathit{after}}(x,\varphi) \\
&\vee\ \exists\, l \in \mathbb{N};\ \psi \in \Psi \bullet \varphi = \underline{\mathit{repmax}}(l,\psi) \wedge (t,n) \models_i^+ \underline{\mathit{repuntil}}(l,\psi,\underline{\mathit{false}})
\end{aligned}
$$

Fig. 1. Semantics of Φ_i^+

a specification-level policy can be checked automatically [24]. In our case, the semantics is as follows: if a triggering event is detected, the condition is evaluated; if it evaluates to true, the action (modify, inhibit, execute) is performed. Since mechanisms are operational in nature, we decided to formulate the conditions in a past variant of our language, Φ^- with semantics $\models^-$ [6, 24]. The fact that mechanisms can inhibit or modify motivates the conceptual distinction between desired and actual events ($E(\cdot)$ and $T(\cdot)$; we could well have restricted the usage of Ψ in specification-level policies to actual events ($E(e)$) which, however, slightly complicates the combined definitions).

Implementation-level policies—for the time being without data flow tracking semantics—hence come in the following forms. We assume a trigger event e and a condition $\varphi \in \Phi^-$. Modifiers are formulas $(T(e) \wedge (E(e) \Rightarrow \varphi)) \Rightarrow T(e') \wedge \neg E(e)$ where e' is like e but with some parameters modified. The idea is that if e is attempted ($T(e)$) and the actual execution of e makes the trigger true ($E(e) \Rightarrow \varphi$), then e' should happen in lieu of e ($T(e') \wedge \neg E(e)$; the reason for having $T(e')$ rather than $E(e')$ is that there might be multiple

$$
\begin{aligned}
&\forall\, t \in \mathit{Trace}, n \in \mathbb{N}, \varphi \in \Phi_i^- \bullet (t,n) \models_i^- \varphi \Leftrightarrow \\
&\exists\, e, e' \in \mathit{Event} \bullet (\varphi = E(e) \vee \varphi = T(e)) \wedge e' \in t(n) \wedge (e', \mathit{states}(t,n)) \models_{\varepsilon,i} \varphi \\
&\vee\ \varphi \in \Phi_i \wedge (t,n) \models_i \varphi \\
&\vee\ \exists\, \psi, \chi \in \Phi_i^- \bullet \varphi = \underline{\mathit{since}}^-(\chi, \psi) \\
&\qquad \wedge\ (\exists\, u \in \mathbb{N} \bullet u \leq n \wedge (t, n-u) \models_i^- \chi \wedge (\forall\, v \in \mathbb{N} \bullet u < v \leq n \Rightarrow (t, n-v) \models_i^- \psi) \\
&\qquad\qquad \vee\ \forall\, v \in \mathbb{N} \bullet v \leq u \Rightarrow (t, n-v) \models_i^- \psi) \\
&\vee\ \exists\, i \in \mathbb{N};\ \psi \in \Phi_i^- \bullet \varphi = \underline{\mathit{before}}^-(i, \psi) \wedge i \leq n \wedge (t, n-i) \models_i^- \psi \\
&\vee\ \exists\, l, x, y \in \mathbb{N};\ \psi \in \Psi \bullet \varphi = \underline{\mathit{replim}}^-(l, x, y, \psi) \\
&\qquad \wedge\ x \leq \textstyle\sum_{j=0}^{\mathit{min}(l,n)} \big|\{S \subseteq \mathit{Event} \mid S \subseteq t(n-j) \wedge \exists\, t' \in \mathit{Trace}\, \forall\, m \in \mathbb{N} \bullet \\
&\qquad\qquad t'(n-j) = S \wedge (n \geq m > n-j \Rightarrow t'(m) = t(m)) \wedge (t', n-j) \models_i^- \psi \\
&\qquad\qquad \wedge \nexists\, S' \subseteq \mathit{Event} \bullet S' \subset S \wedge \exists\, t' \in \mathit{Trace}\, \forall\, m \in \mathbb{N} \bullet \\
&\qquad\qquad t'(n-j) = S' \wedge (n \geq m > n-j \Rightarrow t'(m) = t(m)) \wedge (t', n-j) \models_i^- \psi\}\big| \leq y \\
&\vee\ \exists\, l, u \in \mathbb{N};\ \psi \in \Psi;\ \chi \in \Phi^- \bullet \varphi = \underline{\mathit{repsince}}^-(l, \chi, \psi) \\
&\qquad \wedge\ \big(u \leq n \wedge (t, n-u) \models_i^- \chi \wedge \forall\, v \in \mathbb{N} \bullet u < v \leq n \Rightarrow \neg((t, n-v) \models_i^- \chi) \\
&\qquad\qquad \wedge\ \textstyle\sum_{j=0}^{u} \big|\{S \subseteq \mathit{Event} \mid S \subseteq t(n-j) \wedge \exists\, t' \in \mathit{Trace}\, \forall\, m \in \mathbb{N} \bullet \\
&\qquad\qquad\qquad t'(n-j) = S \wedge (n \geq m > n-j \Rightarrow t'(m) = t(m)) \wedge (t', n-j) \models_i^- \psi \\
&\qquad\qquad\qquad \wedge \nexists\, S' \subseteq \mathit{Event} \bullet S' \subset S \wedge \exists\, t' \in \mathit{Trace}\, \forall\, m \in \mathbb{N} \bullet \\
&\qquad\qquad\qquad t'(n-j) = S' \wedge (n \geq m > n-j \Rightarrow t'(m) = t(m)) \wedge (t', n-j) \models_i^- \psi\}\big| \leq l\big) \\
&\qquad \vee\ \textstyle\sum_{j=0}^{n} \big|\{S \subseteq \mathit{Event} \mid S \subseteq t(n-j) \wedge \exists\, t' \in \mathit{Trace}\, \forall\, m \in \mathbb{N} \bullet \\
&\qquad\qquad t'(n-j) = S \wedge (n \geq m > n-j \Rightarrow t'(m) = t(m)) \wedge (t', n-j) \models_i^- \psi \\
&\qquad\qquad \wedge \nexists\, S' \subseteq \mathit{Event} \bullet S' \subset S \wedge \exists\, t' \in \mathit{Trace}\, \forall\, m \in \mathbb{N} \bullet \\
&\qquad\qquad t'(n+j) = S' \wedge (n \geq m > n-j \Rightarrow t'(m) = t(m)) \wedge (t', n-j) \models_i^- \psi\}\big| \leq l \\
&\vee\ \exists\, \psi \in \Phi_i^- \bullet \varphi = \underline{\mathit{always}}^-(\psi) \wedge (t,n) \models_i^- \underline{\mathit{since}}^-(\underline{\mathit{false}}, \psi) \\
&\vee\ \exists\, i \in \mathbb{N};\ \psi \in \Phi_i^- \bullet \varphi = \underline{\mathit{within}}^-(i, \psi) \wedge i < n \wedge (t,n) \models_i^- \textstyle\bigvee_{x=1}^{i} \underline{\mathit{before}}^-(i, \varphi) \\
&\vee\ \exists\, i \in \mathbb{N};\ \psi \in \Phi_i^- \bullet \varphi = \underline{\mathit{during}}^-(i, \psi) \wedge i < n \wedge (t,n) \models_i^- \textstyle\bigwedge_{x=1}^{i} \underline{\mathit{before}}^-(x, \varphi) \\
&\vee\ \exists\, l \in \mathbb{N};\ \psi \in \Psi \bullet \varphi = \underline{\mathit{repmax}}^-(l, \psi) \wedge (t,n) \models_i^- \underline{\mathit{repsince}}^-(l, \underline{\mathit{false}}, \psi)
\end{aligned}
$$

Fig. 2. Semantics of Φ_i^-

concurrently executing mechanisms). Inhibitors are formulas $(T(e) \wedge (E(e) \Rightarrow \varphi)) \Rightarrow \neg E(e)$ that simply prohibit the desired event $T(e)$ by requiring $\neg E(e)$ in case $E(e)$ would make φ true. Finally, executors are expressed as $(T(e) \wedge (E(e) \Rightarrow \varphi)) \Rightarrow T(e') \wedge E(e)$ for some event e' to be executed; again, since there may be multiple mechanisms in place, e' can only be attempted at this stage. The formal semantics of a set of combined mechanisms as well as conflict detection has been described elsewhere [6, 24].

The structure of Φ^- reflects that of Φ^+. Observe that the semantics of Φ_i, $\models_i$, "does not look into the future" and makes use of the *states* function *that already is defined solely in terms of the past*. As a consequence, we use

$$
\begin{aligned}
\Phi^- ::= \Psi \mid\ & \underline{\mathit{not}}^-(\Phi^-) \mid \underline{\mathit{and}}^-(\Phi^-, \Phi^-) \mid \underline{\mathit{or}}^-(\Phi^-, \Phi^-) \mid \underline{\mathit{implies}}^-(\Phi^-, \Phi^-) \mid \underline{\mathit{since}}^-(\Phi^-, \Phi^-) \mid \\
& \underline{\mathit{before}}^-(\mathbb{N}, \Phi^-) \mid \underline{\mathit{within}}^-(\mathbb{N}, \Phi^-) \mid \underline{\mathit{during}}^-(\mathbb{N}, \Phi^-) \mid \\
& \underline{\mathit{always}}^-(\Phi^-) \mid \underline{\mathit{repmax}}^-(\mathbb{N}, \Psi) \mid \underline{\mathit{replim}}^-(\mathbb{N}, \mathbb{N}, \mathbb{N}, \Psi) \mid \underline{\mathit{repsince}}^-(\mathbb{N}, \Psi, \Phi^-);
\end{aligned}
$$

let $\Phi_i^- ::= \Phi_i \mid \Phi^-$, verbatim reuse the definition of $\models_i$, and get the combined semantics of Φ_i^-, $\models_i^-$, in Figure 2, where we omit the definition of the propositional operators. Because of space limitations, we do not provide the semantics of entire mechanisms (that is: entire ECA rules, not just conditions) here; this straightforwardly generalizes the case without data flow tracking [6].

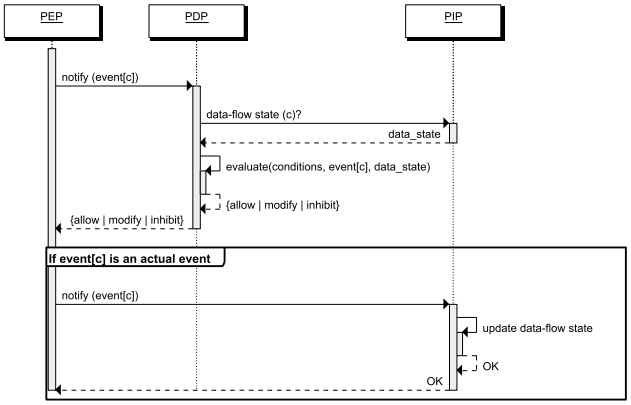

Fig. 3. Interplay of PEP, PDP, PIP

Step 5: Architecture Our generic architecture is the same for each concrete system layer at which the infrastructure is instantiated. We distinguish three main components: a *Policy Enforcement Point* (PEP), able to observe, intercept and 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 state $\sigma \in \Sigma$ to the PDP.

The role of the PEP is to implement the mechanisms of step 4. PEPs intercept desired and actual events, signal them to the PDP and, according to the response, allow, inhibit or modify them. Using the events signaled by the PEP, the PDP evaluates the policies, more specifically, the condition of the ECA rules. While we implemented one specific algorithm [25] for the PDP, any runtime verification algorithm can be used [7]. Due to its generic nature, the same implementation can be reused at different system layers: only the binding of events in the system to events specified in the policies has to be performed. In order to take a decision, the PDP may need additional information (e.g., in case of state-based formulae or data usages) concerning the distribution of data among the different representations. For this reason the PDP queries the PIP. The PIP represents a (layer-specific) implementation of the data-flow tracking model presented in step 3. In order to properly model the evolution of the data-flow state, the PEP notifies the PIP about every actual event that happens in the system, and the PIP then updates its data state $\sigma \in \Sigma$ according to ϱ.

The interplay of PEP, PDP, and PIP is shown in Figure 3. Whenever the PDP checks an actual (container) event e against a data usage event u in a policy, the PIP is consulted to check if the data item referred to by u is contained in the container referred to by e.

Step 6: Multi-layer data flow detection and usage control enforcement

In the example of the social network application in Section 1 we have three monitors: one at the level of the operating system, one at the level of the web browser, and one at the level of the X11 system. Now, some events, together with the data that they operate on, at one layer imply related events at a different

layer. For instance, saving a page in the web browser (event *save*) implies a *write()* system call at the operating system layer. As a consequence, data flows from one layer to another one.

We introduce a set of layers, L, that includes layers such as X11, the operating system, a browser, etc. For each event, we assume that there is precisely one layer at which this event happens (if there is more than one layer, then this is captured by the following function π). This motivates the definition of a function $\lambda : \mathit{Event} \to L$ that partitions the set of events. Note that neither our definition of the transition relation ϱ nor the definition of the data state σ nor the definition of languages Φ_i^+ and Φ_i^- require events, containers, and data to reside at one level of abstraction. As a consequence, we may assume that our system is specified globally, i.e. encompassing all levels of abstraction. We can then use function λ to separate the different layers: $\mathit{Event}_\ell = \{e \in \mathit{Event} : \lambda(e) = \ell\}$ contains the events relevant at layer ℓ, and, using graph notation, $\varrho_\ell = \{(\sigma, E, \sigma') : E \subseteq \mathit{Event}_\ell \wedge \varrho(\sigma, E) = \sigma'\}$ projects the data flow transition relation to layer ℓ (remember that in step 2 of Section 2, we required independence of events in the definition of ϱ for simplicity's sake). With ϱ_ℓ and Event_ℓ, we can implement the data flow monitor for layer ℓ as described in step 5. The usage control monitor part is synthesized from a policy; the only layer-specific part is Event_ℓ. In the implementation, the set of ϱ_ℓ and Event_ℓ hence defines the set of independent enforcement mechanisms for all layers $\ell \in L$.

We now consider the flow of data in-between different layers. To this end, we introduce a relation $\pi : \mathit{Event} \to 2^{\mathit{Event}}$. With this relation, it is possible to specify, *at the model level*, that whenever an event happens at one layer ℓ_1, a set of simultaneous events at another layer ℓ_2 necessarily take place. Formally, we can capture this intuition by a constraint on the set of traces of a system: $\forall\, s \in \mathit{Trace}\, \forall\, t \in \mathbb{N}\, \forall\, e \in \mathit{Event} : e \in s(t) \Rightarrow \pi(e) \subseteq s(t)$. In other words, via π we require *in the semantic model* that, for instance, there must be a write() system call whenever there is a save action in the web browser, thus capturing the data flow from browser to operating system.

In this way, cross-layer data flow tracking and data-driven usage control enforcement can be *specified* in a conceptually very simple way. However, in terms of the *implementation*, this is far more challenging. While every layer-specific infrastructure instantiates the general model, our current cross-layer enforcement solution is an ad-hoc implementation that relates an event at one layer to an event at another layer in a hard-coded way (Section 4).

4 Instantiations

Operating System: OpenBSD At the operating system level, system calls are the events that can change the state of the system. The complete description of the data-flow tracking model can be found in [3]. Here, we show how to extend this implementation with a usage control monitor, thus providing an instance of the combined model of this paper. Events are system calls, and they are invoked by processes on data containers. Containers include files, pipes, message queues and the network. A process itself is also considered as a data container

because the process state, CPU registers and the memory image of the process are possible locations for data. Data containers are identified by a set of names, which includes file names, descriptors and sockets. Each state consists of the three mappings presented in Section 3: *storage, alias* and *naming*. As an example, aliases are created if memory is mapped to a file system (mmap() system call). The transition relation ϱ is described in [3].

The combined usage control and data flow tracking system is implemented using *Systrace*, a policy enforcement tool for monitoring, intercepting and modifying system calls in OpenBSD. In contrast to our earlier work [3], the combined implementation of this paper can enforce advanced usage control policies that address all the instances of the same data at the same time.

One example policy is from the DRM world: the content (dataUsage, lines 11 and 16) of a file, song.mp3, can be used, i.e. opened, (lines 10 and 15) at most 4 further times and within 30 seconds (1 timestep = 1 second) after the first use (lines 7-19); further attempts of opening the file will result in opening a predefined error message (lines 21-23). We provide it to demonstrate the use of complex conditions. Further examples are available [23].

```
1  <controlMechanism>
2   <id>OS_DRM_example</id>
3   <triggerEvent> <id>open</id>
4                  <param name="obj" value="song.mp3" type="dataUsage"/>
5                  <param name="isTry" value="true"/>
6   </triggerEvent>
7   <condition>
8   <or>
9     <not><before timeInterval="30"><always><not>
10      <event>  <id>open</id>
11               <param name="obj" value="song.mp3" type="dataUsage"/>
12      </event>
13    </not></always></before></not>
14    <not><repmax limit="5">
15      <event>  <id>open</id>
16               <param name="obj" value="song.mp3" type="dataUsage"/>
17      </event>
18    </repmax></not></or>
19  </condition>
20  <actions> <allow> <modify>
21     <param name="obj" value="/etc/UCmon/expired.msg" />
22  </modify> </allow> </actions>
23 </controlMechanism>
```

The effect of our implementation can be seen by executing the following sequence of commands:

```
> vlc song.mp3 && cp song.mp3 song2.mp3 && mv song2.mp3 song3.mp3 &&
  cat song3.mp3 > song4.mp3 &&
  ... (after more than 30 seconds) ... && vlc song4.mp3   --> ERROR!
```

When trying to play (command *vlc*) the file *song4.mp3* (a copy of the original *song.mp3*) more than 30 seconds after the first play, an error message is played instead of the song. The same error is generated when trying to open whatever instance of the song after the fifth time.

Windowing System: X11 X11 is a distributed system and a protocol for GUI environments on Unix-like systems. In X11, events that change the state of the

system are network packets exchanged between clients and servers. The model for data-flow tracking and primitive usage control is described elsewhere [4]. Events are requests, replies, events and errors, invoked on specific X11 resources by principals that, because of the distributed setting, are identified by IP address and port. Resources form the containers that potentially carry sensitive information, like windows, pixmaps (memory areas that are valid destinations for drawing functions), atoms (unique names for accessing resources or for communication between different clients), attributes and properties (variables attached to windows), etc. States consists of the three mappings presented in Section 3: *storage, alias* and *naming*. Among others, aliases are created whenever windows overlap translucently. The transition relation ϱ is described in [4].

The combined usage control and data flow tracking system is implemented using *Xmon*, an X11 debugging tool for monitoring, intercepting and modifying network packets from/to an X server. As opposed to [4], thanks to the usage control runtime monitor, it is able to enforce advanced usage control policies, with temporal and cardinality operators, addressing all instances of the same data at the same time. One example policy is the following.

```
1  <controlMechanism>
2   <id>X11_Screenshot</id>
3   <triggerEvent> <id>GetImage</id>
4                  <param name="obj" value="0x1a00005" type="dataUsage"/>
5                  <param name="isTry" value"true"/>
6   </triggerEvent>
7   <condition> <true /> </condition>
8   <actions> <allow>
9    <modify> <param name="planeMask" value="0x0" /> </modify>
10  </allow> </actions>
11 </controlMechanism>
```

In this example, the enforcement mechanism prevents the X client from taking a screenshot (X11 action *GetImage*, line 3) of the content of window 0x1a00005 (line 4; in the multi-layer example, this data is filled in by the web browser PEP). If a client requests a screenshot of that window, the action is permitted (line 8), but the parameter *planeMask* is modified to the value 0x0 (line 9). *planeMask* represents which set of drawable objects should be included in the screenshot: a *planeMask* of 0xffff means that every plane is contained in the screenshot, whereas invoking *GetImage* with *planeMask* equal to 0x0 returns a black image because no plane is included. Further examples are available [23].

Web Browser: Firefox A third instance of our model at the browser level extends an existing usage control extension for the Firefox web browser [26]. In this scenario, we want to protect sensitive web page content from malicious usage by the user of the browser. Here, we show how to instantiate the data-flow tracking model to objects of the browser domain, in order to extend the existing implementation to another instance of the combined model presented so far.

Events include the user actions “copy”, “paste”, “print”, “save as”, etc., and are performed by a user on web page content. Content is stored in two types of containers: read-only (the non-editable part of a web-page) and read-write (text fields where it is possible to type); in addition, there is the clipboard. The only principal in this scenario is the user of the browser. The browser-level

instantiation does not require the alias function, because no alias relations are created among containers. Similarly, the naming function is constant. Therefore, a state of the system is given only by the state of the storage function $\Sigma = (Container \rightarrow 2^{Data})$. Due to space constraints, we do not present the definition of the transition relation ϱ here. The resulting system can enforce advanced policies that address all the representations of the same data. Our example is from the social network scenario: a user is allowed to print a profile picture (lines 3-6) only once (lines 8 and 14). More examples are available [23].

```
1  <controlMechanism>
2   <id>Browser_Print</id>
3    <triggerEvent> <id>print</id>
4                   <param name="obj" value="imgprofile" type="dataUsage"/>
5                   <param name="isTry" value"true"/>
6   </triggerEvent>
7   <condition>
8    <not> <repmax limit="1">
9      <event> <id>print</id>
10              <parameter name="obj" value="imgprofile" type="dataUsage"/>
11     </event>
12    </repmax> </not>
13  </condition>
14  <actions> <inhibit/> </actions>
15 </controlMechanism>
```

Multi-Layer Enforcement We also implemented multi-layer usage control by combining the three implementations presented above [1]. To do so, we deployed the three monitors, each consisting of PEP, PDP, and PIP, on the same physical system and made them communicate with each other. A general protocol for such a communication among arbitrary parties is the subject of current work, so we hard-coded a solution tailored for this specific scenario: we made the Firefox monitor able to instruct the OS and X11 monitors about new policies and data flows from the browser layer to the operating system and the windowing system.

We consider a social network [26] where a user watches a picture on someone else's profile page. Since the picture is considered sensitive, its usage is restricted. In particular, no local usage is allowed after download, except for printing, and whenever the picture is printed, a notification must be sent to the owner. The respective specification policy is *"This picture cannot be copied to the clipboard (not even as a screenshot) nor saved to disk and its cached version can be used only by Firefox. No printing of the picture without notification of the owner."* We do not show the implementation-level policies here. Together with a description of the interplay of the components, they are provided elsewhere [23].

In our implementation, at each layer we distinguish between the business logic and the monitoring component which instantiates the model presented in this paper (PEP, PDP and PIP in step 5 of Section 3). If the user requests the page with picture Pic, the browser downloads the profile page together with a policy that contains a sub-policy related to the figure. Upon reception by the web browser, Pic takes new representations: it is rendered as a set of pixels inside the browser window, it is cached as a file, and it is internally represented by the browser in some memory region referenced by a node in the DOM tree. Each representation must be protected at its layer in the system.

To do so—and this is the ad hoc part of the implementation—the browser monitor instantiates the generic policy it got from the remote server to each layer by adding runtime information including the name of the cache file and the ID of the window. Because this data is created at runtime, it cannot be statically determined by the server a priori. After instantiating and deploying the policies to the OS and X11 layers, the browser monitor allows rendering the picture and creating the cache file. From this point onward, all three instantiations of the policy are enforced at different levels of abstraction.

5 Related Work

The subject of this paper is the combination of data flow detection with usage control, a policy language, and a prototype enforcement infrastructure.

Enforcement of usage control requirements has been done for the OS layer [27, 28, 3], for the X11 layer [4], for Java [10, 11, 29], the .NET CIL [12] and machine languages [13, 14, 30]; at the level of an enterprise service bus [15]; for dedicated applications such as the Internet Explorer [16] and in the context of digital rights management [17–19]. These solutions focus on either data flow tracking or event-driven usage control. Our model, in contrast, tackles both at the same time and since it is layer-independent, it can be instantiated to each of these layers. At the level of binary files, the Garm tool [30] combines data tracking with an enforcement mechanism for basic usage control. This model focuses on access control, trust and policy management aspects, while our goal is a generic model and a policy language to express and enforce advanced usage control requirements for arbitrary system layers. Data flow confinement is also intensely studied at the operating system level [27, 28]; here, our work differs in that we aim at enforcing complex usage control policies.

A multitude of policy languages [2, 31–37] has been proposed. As far as we know, none of them addresses the data dimension like ours does; they allow for definitions of usage restrictions for specific rather than all representations of data, and their semantic models do not consider data flows.

In terms of data flow tracking, our approach restricts the standard notion of information flow which also caters to implicit flows and aims at non-interference assessments [38, 39, 20, 21]: our system detects only flows from container to container. This explains why we prefer to speak of data flow rather than information flow. Moreover, even if we plan to leverage results of static analyses, like [40], we want to detect these flows at runtime. Implementations of such data-flow tracking system have been realized for OS [3], X11 [4], OpenOffice [5] and Java byte code and can be used as PIP component to instantiate our model. This cited work, however, only addresses data flow detection without full usage control.

In terms of general-purpose usage control models, there are similarities with the models underlying XACML [41], Ponder2 [42] and UCON [43]. The first two, however, do not provide formalized support for cardinality or temporal operators (free text fields exist, but the respective requirements are hard to enforce). UCON supports complex conditions [44], and has been used in applications at different system layers, such as the Java Virtual Machine [45] and the Enterprise Service Bus [46]. Data flow is not considered, however.

Complex event processing [22] and runtime monitoring [7] are suitable for monitoring conditions of usage control policies. As such, they address one aspect of the problem, namely the monitoring part, and do not cater to data flow.

6 Conclusions

The contribution of this paper is a combination of usage control with data flow detection technology. Rather than specifying and enforcing usage control policies on specific representations of a data item (usually encoded in usage-controlled events), our work makes it possible to specify and enforce usage control policies for all representations of a data item (files, windows contents, memory contents, etc.). We provide a model, a language, an architecture and a generic implementation for data-centric usage control enforcement that we instantiate to several system layers. Our implementation consists of combined usage control and data flow monitors for an operating system, a windowing system, and a web browser, together with a multi-layer enforcement infrastructure for them. As an example, this system makes it possible that a user can download a picture on a web page and watch it in the browser but not copy&paste or print the content without notification (enforced at the browser layer); nor take a screenshot (enforced at the X11 layer); nor access the cache files (enforced at the OS layer) [1].

Because of space restrictions, we have not provided security nor performance analyses. While we do not claim that our system cannot be circumvented, we believe that a reasonable level of security can be attained [26, 9]. Performance-wise, we currently are faced with an overhead of one to two orders of magnitude [3, 4]. This, however, heavily depends on the kind of events that happen in our system; moreover, our system is not optimized at all. Security and performance analyses and improvements are the subject of current work. This paper also does not solve the problem of policy deployment, livecycle management, delegation and media breaks (e.g., taking a photograph of the screen).

Our current data flow model is very simple. While it is appropriate for use cases of the kind we presented here, the involved overapproximations quickly lead to a label creep in practice. For instance, at the OS-level, if a process reads a file that contains one tainted bit, then every subsequent output of the process is tainted. We are currently investigating how to adopt McCamant and Ernst's quantitative information flow model [47] as well as dynamic declassification techniques to overcome this problem. The system layers we catered to in this paper do not exhibit indirect information flow caused by control flow; this is, however, the case for runtime systems. We plan to combine static and dynamic analyses at this level to get more precise data flow models for these layers.

In terms of further future work, we need a generic implementation for cross-layer enforcement, a formal model that caters to dependent events at one moment in time, and a way of protecting the enforcement infrastructure that not necessarily inherits the disadvantages of trusted computing technology [9].

Acknowledgments This work was supported by the DFG under grant no. PR 1266/1-1 as part of the priority program SPP 1496, "Reliably Secure Software Systems." Florian Kelbert provided valuable comments.

References

1. Enrico Lovat and Alexander Pretschner. Data-centric multi-layer usage control enforcement: A social network example. In *Proceedings of the 16th ACM symposium on Access control models and technologies*, SACMAT '11, pages 151–152, 2011.
2. M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. A policy language for distributed usage control. In *Proc. ESORICS*, pages 531–546, 2008.
3. M. Harvan and A. Pretschner. State-based Usage Control Enforcement with Data Flow Tracking using System Call Interposition. In *Proc. 3rd Intl. Conf. on Network and System Security*, pages 373–380, 2009.
4. A. Pretschner, M. Buechler, M. Harvan, C. Schaefer, and T. Walter. Usage control enforcement with data flow tracking for x11. In *Proc. 5th Intl. Workshop on Security and Trust Management*, pages 124–137, 2009.
5. C. Schaefer, T. Walter, A. Pretschner, and M. Harvan. Usage control policy enforcement in OpenOffice.org and information flow. *HS Venter, M Coetzee and L Labuschagne*, page 393, 2009.
6. A. Pretschner, M. Hilty, D. Basin, C. Schaefer, and T. Walter. Mechanisms for Usage Control. In *Proc. ACM Symposium on Information, Computer & Communication Security*, pages 240–245, 2008.
7. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.
8. M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. Monitors for usage control. In *Proc. Trust Management*, volume 238, pages 411–414, 2007.
9. Ricardo Neisse, Dominik Holling, and Alexander Pretschner. Implementing trust in cloud infrastructures. In *11th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGrid 2011 (to appear)*, 2011. Available at: `http://zvi.ipd.kit.edu`.
10. M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Security monitor inlining for multithreaded java. In *Proc. ECOOP*, pages pp. 546–569, 2009.
11. I. Ion, B. Dragovic, and B. Crispo. Extending the Java Virtual Machine to Enforce Fine-Grained Security Policies in Mobile Devices. In *Proc. Annual Computer Security Applications Conference*, pages 233–242. IEEE Computer Society, 2007.
12. L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe. The S3MS.NET Run Time Monitor: Tool Demonstration. *ENTCS*, 253(5):153–159, 2009.
13. U. Erlingsson and F. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. New Security Paradigms Workshop*, pages 87–95, 1999.
14. B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proc IEEE Symposium on Security and Privacy*, pages 79–93, 2009.
15. G. Gheorghe, S. Neuhaus, and B. Crispo. xESB: An Enterprise Service Bus for Access and Usage Control Policy Enforcement. In *Proc. Annual IFIP WG 11.11 International Conference on Trust Management*, 2010.
16. M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *Proceedings of USENIX Annual Technical Conference*, June 2007.
17. Adobe livecycle rights management es. `http://www.adobe.com/products/livecycle/rightsmanagement/indepth.html`, August 2010.
18. Microsoft. Windows Rights Management Services. `http://www.microsoft.com/windowsserver2008/en/us/ad-rms-overview.aspx`, 2010.

19. A. Pretschner, M. Hilty, F. Schutz, C. Schaefer, and T. Walter. Usage control enforcement: Present and future. *Security & Privacy, IEEE*, 6(4):44–53, 2008.
20. Heiko Mantel. Possibilistic definitions of security - an assembly kit. *Computer Security Foundations Workshop, IEEE*, 0:185, 2000.
21. Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8:399–422, 2009. 10.1007/s10207-009-0086-1.
22. David Luckham. The power of events: An introduction to complex event processing in distributed enterprise systems. In Nick Bassiliades, Guido Governatori, and Adrian Paschke, editors, *Rule Representation, Interchange and Reasoning on the Web*, volume 5321 of *Lecture Notes in Computer Science*, pages 3–3. Springer Berlin / Heidelberg, 2008.
23. Alexander Pretschner, Enrico Lovat, and Matthias Büchler. Representation-Independent Data Usage Control. Technical Report 2011,23, Karlsruhe Institute of Technology, Department of Computer Science, August 2011. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000024005.
24. Alexander Pretschner, Judith Rüesch, Christian Schaefer, and Thomas Walter. Formal analyses of usage control policies. In *ARES*, pages 98–105, 2009.
25. Klaus Havelund and Grigore Rosu. Efficient monitoring of safety properties. *Int. J. Softw. Tools Technol. Transf.*, 6, Aug 2004.
26. Prachi Kumari, Alexander Pretschner, Jonas Peschla, and Jens-Michael Kuhn. Distributed data usage control for web applications: a social network implementation. In *Proceedings of the first ACM conference on Data and application security and privacy*, CODASPY '11, pages 85–96, New York, NY, USA, 2011. ACM.
27. Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *Proc. SOSP*, pages 17–30, 2005.
28. Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 293–308, 2008.
29. William Enck, Peter Gilbert, Byung-Gon Chun, Landon Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010. To appear.
30. Brian Demsky. Garm: cross application data provenance and policy enforcement. In *Proceedings of the 4th USENIX conference on Hot topics in security*, HotSec'09, pages 10–10, Berkeley, CA, USA, 2009. USENIX Association.
31. R. Iannella (ed.). Open Digital Rights Language v1.1, 2008. `http://odrl.net/1.1/ODRL-11.pdf`.
32. Multimedia framework (MPEG-21) – Part 5: Rights Expression Language, 2004. ISO/IEC standard 21000-5:2004.
33. P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter. Enterprise Privacy Authorization Language (EPAL 1.2). IBM Technical Report, 2003. `http://www.zurich.ibm.com/security/enterprise-privacy/epal/Specification/`.
34. Open Mobile Alliance. DRM Rights Expression Language V2.1, 2008. `http://www.openmobilealliance.org/Technical/release_program/drm_v2_1.aspx`.
35. X. Zhang, J. Park, F. Parisi-Presicce, and R. Sandhu. A logical specification for usage control. In *SACMAT '04: Proceedings of the ninth ACM symposium on Access control models and technologies*, pages 1–10. ACM, 2004.

36. N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *Proc. Workshop on Policies for Distributed Systems and Networks*, pages 18–39, 1995.
37. W3C. The Platform for Privacy Preferences 1.1 (P3P1.1) Specification, 2005. `http://www.w3.org/TR/2005/WD-P3P11-20050104/`.
38. J. Rushby. Noninterference, transitivity and channel-control security policies, 1992.
39. J.A. Goguen and J. Meseguer. Security policies and security models. In IEEE Computer Society Press, editor, *Proc of IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.
40. Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Blo Jason A., George A. Reis, Manish Vachharajani, and David I. August. Rifle: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 243–254, Washington, DC, USA, 2004. IEEE Computer Society.
41. E. Rissanen. Extensible access control markup language v3.0. `http://docs.oasis-open.org`, 2010.
42. Kevin Twidle, Emil Lupu, Naranker Dulay, and Morris Sloman. Ponder2 - a policy environment for autonomous pervasive systems. *Policies for Distributed Systems and Networks, IEEE International Workshop on*, 0:245–246, 2008.
43. J. Park and R. Sandhu. The UCON ABC usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, 2004.
44. Xinwen Zhang, Jaehong Park, Francesco Parisi-Presicce, and Ravi Sandhu. A logical specification for usage control. In *Proc. 9th ACM symposium on access control models and technologies*, pages 1–10, 2004.
45. Srijith K. Nair, Andrew S. Tanenbaum, Gabriela Gheorghe, and Bruno Crispo. Enforcing drm policies across applications. In *Proceedings of the 8th ACM workshop on Digital rights management*, DRM '08, pages 87–94, New York, NY, USA, 2008. ACM.
46. Gabriela Gheorghe, Paolo Mori, Bruno Crispo, and Fabio Martinelli. Enforcing ucon policies on the enterprise service bus. In *Proceedings of the 2010 international conference on On the move to meaningful internet systems: Part II*, OTM'10, pages 876–893, Berlin, Heidelberg, 2010. Springer-Verlag.
47. Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. In *PLDI*, pages 193–205, 2008.