Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2011,23
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Representation-Independent Data Usage Control

Alexander Pretschner, Enrico Lovat, Matthias Büchler

2011

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 cross-layer enforcement.

1 Introduction

If usage control enforcement is to be enforced on data, then one must take into account the fact that this data exists in multiple representations. For instance, there can be multiple copies of a file, or multiple clones of an object. Moreover, an image can exist as network packet, Java object, window pixmap, data base record, or file. The representations hence potentially reside at different layers of abstraction, including operating system, runtime system, window manager, and DBMS. High-level usage control requirements such as "don't copy" tend to have different meanings at these 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 processor and memory, it turns out to be hard to identify, *in general*, precisely those CPU 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 layers of abstraction. This, however, makes it necessary to follow the flow of data from one representation to another within and across abstraction layers.

In this paper, we present a model and an implementation of a framework that combines 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 at first layer of abstraction) but not copy cache files (second representation, second layer) or take screenshots (third representation, third layer) [1]. This paper describes the model and its prototypical implementation; detailed security and performance analyses are not in the scope. We organize our paper along the following steps.

Step 1: Specification-level usage control policies based on events We start with a trace-based semantic model. Traces are infinite streams of sets of events, $\mathit{Trace} = \mathbb{N} \to \mathbb{P}(\mathit{Event})$. In this model, usage control policies are interpreted as sets of allowed sequences of sets of events. We call these policies *specification-level policies.* The semantics of a policy, expressed in language Φ^+ (here: a future time temporal logic, hence the $+$) is the set of traces that makes the corresponding formula true. This is captured by a relation $\models^+ \subseteq \mathit{Trace} \times \Phi^+$. This policy language is part of the literature [2].

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. The data state changes with every step (set of events) of the system via a transition function $\varrho : \Sigma \times \mathbb{P}(\mathit{Event}) \to \Sigma$. Using ϱ, the current data state can, for each trace t and each moment in time n, be recursively computed from the past by the function $\mathit{states}(t, n)$. This data flow model (which in fact is slightly more complex, as we will see below) has been described and instantiated to various levels of abstraction before [3–5]. In this paper, we embody the data flow model in a usage control policy language and an integrated semantic model; this constitutes the core contribution of this paper.

Step 3: Specification-level usage control policies based on data In Φ^+, we can only express container usages, i.e., usage events that pertain to *one specific representation.* As argued above, we deem it natural to express *data usages* as well, which pertain to all representations of the same data. We hence augment the policy language Φ^+ by (1) data usages and (2) special operators that operate on data rather than containers—e.g., some data may not flow into a specific container such as a network socket. This new language is called Φ_i^+. The semantics of Φ_i^+, $\models_i^+ \subseteq \mathit{Trace} \times \Phi_i^+$, is defined on the grounds of the data state function *states*. At each moment in time, we compute the current data state via *states* and use it for defining the semantics of data usages and the special operators. Essentially, if a *data usage* is specified, we evaluate the respective formula w.r.t. *all the containers* that, according to the current data state, contain the respective data item. In this paper, we provide the language and its formal semantics.

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 happend and the respective condition has evaluated to true. The action can be to inhibit the trigger event (which requires the distinction between desired and actual events), to modify the trigger event (which also requires this distinction), 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. This language Φ^- and the semantics $\models^-$ are the natural past duals of Φ^+ and $\models^+$ [6]. In this paper, similar to Φ_i^+, we augment Φ^- by data usages and special state-based operators and obtain Φ_i^- with semantics $\models_i^- \subseteq \mathit{Trace} \times \Phi_i^-$. The distinction between specification-level and implementation-level policies in the context of usage control based on events has been described in the literature [6, 7]. In this paper, we add the data dimension also to implementation-level policies.

Step 5: Runtime monitors for events and data The semantic relations $\models^+, \models^-, \models_i^+, \models_i^-$ are of a declarative nature. Since Φ^+ and Φ^- essentially boil down to future and past LTL, we can leverage the huge body of work in the area of runtime verification to synthesize efficient monitors both for monitoring 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 verification technology is available [8].

It is straightforward to implement the evolution of the data state. At each moment in time, we intercept the current events 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 *states* 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 levels of abstraction, there hence is one data state tracker per abstraction layer, and one runtime monitor per layer per policy. While pure usage control monitors [9] 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 in this paper.

Step 6: Cross-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 abstraction, but also across layers of abstraction. Conceptually, this turns out to be rather simple, and we provide a respective coarse model in this paper. The implementation is, however, far more difficult if a *general* solution is sought (how does the operating system know that a specific window content corresponds to a file?). In this paper, we provide a non-generic ad-hoc implementation of this combined model as a proof of concept.

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

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 abstraction. We use the formalism in this paper to clarify concepts only; in a second step, we plan to extend existing work on analysis technology [7] to the combined model. Secondly, as a proof of concept, we show how to implement such a system. Practical security issues are not the focus of this paper which is why we do not present a security analysis, and we do not claim that our implementation cannot be circumvented (respective protection technology is described elsewhere [10]). We do not discuss performance nor policy management either.

Contribution Data flow tracking at specific layers of abstraction has been done in a multitude of ways [3, 4, 11–20], also in the form of information flow analyses where implicit flows are also taken into account [21, 22]. 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 [23, 8], 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 Section 2 recapitulates a semantic model and a policy language for usage control as well as a simple semantic model for data flow from the literature. Section 3 presents our combined model. Section 4 describes three different instantiations as well as an ad-hoc cross-layer enforcement implementation. Section 5 puts our work in context, and Section 6 concludes with a discussion.

2 Background

In this section, we recap the specification-level policy specification language [2] and the data flow model [3–5] that we will combine in Section 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 $\mathit{Event} \subseteq \mathit{EventName} \times \mathit{Params}$ consists of the event's name and parameters, represented as a partial ($\nrightarrow$) function from names to values: $\mathit{Params} \subseteq \mathit{ParamName} \nrightarrow \mathit{ParamValue}$ for basic types $\mathit{ParamName}$, $\mathit{ParamValue}$, $\mathit{EventName}$. We denote event parameters by their graph, i.e., as $(\mathit{name}, \mathit{value})$ pairs. We assume a reserved parameter name, *obj*, to indicate, in case of a usage event, on which data item the event is performed. An example is the event $(\mathit{show}, \{(\mathit{obj}, x)\})$, where *show* is the name of the event and the parameter *obj* has the value x. Moreover, 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, a policy likely does not contain the time-stamp of an event, but if the event actually happens, the time stamp is present. As a second example, if the event $(\mathit{show}, \{(\mathit{obj}, x)\})$ is prohibited, then the event $(\mathit{show}, \{(\mathit{obj}, x), (\mathit{window}, w)\})$ 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 $\mathit{refinesEv} \subseteq \mathit{Event} \times \mathit{Event}$ with $\forall\, e_1, e_2 \in \mathit{Event} \bullet e_2\ \mathit{refinesEv}\ e_1 \Leftrightarrow e_1.1 = e_2.1 \wedge e_1.2 \subseteq e_2.2$. In the semantic model, we will assume traces to be maximally refined (all parameters carry values; this seems natural in an actually running system): $\mathit{maxRefinedEv} = \{e \in \mathit{Event} : \forall\, e' \in \mathit{Event} \bullet e'\ \mathit{refinesEv}\ e \Rightarrow e' = e\}$. 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: $\mathit{Trace} : \mathbb{N} \to \mathbb{P}(\mathit{maxRefinedEv})$.

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 [7], and where we omit the permission operator for brevity's sake. We distinguish between purely propostional (Ψ) and temporal and cardinality operators (Φ^+).

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

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 $\models_\varepsilon \subseteq \mathit{Event} \times \Phi^+$ that relates events (rather than traces) to formulae of the form $E(\cdot)$ or $T(\cdot)$ as follows: $\forall\, e, e' \in \mathit{Event} \bullet e \models_\varepsilon E(e') \Leftrightarrow e\ \mathit{refinesEv}\ e' \wedge e.2(\mathit{isTry}) = \mathit{false}$ and $\forall\, e, e' \in \mathit{Event} \bullet e \models_\varepsilon T(e') \Leftrightarrow e\ \mathit{refinesEv}\ e' \wedge e.2(\mathit{isTry}) = \mathit{true}$.

$\underline{\mathit{not}}$, $\underline{\mathit{and}}$, $\underline{\mathit{or}}$, $\underline{\mathit{implies}}$ have the usual semantics. The $\underline{\mathit{until}}$ operator is the weak-until operator from LTL. Using $\underline{\mathit{after}}(n)$, which refers to the time after n time steps, we can express concepts like $\underline{\mathit{during}}$ (something must constantly hold during a specified time interval) and $\underline{\mathit{within}}$ (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 $\underline{\mathit{replim}}$ operator specifies lower and upper bounds of times within a fixed time interval in which a given formula holds. The $\underline{\mathit{repuntil}}$ 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 $\underline{\mathit{repuntil}}$, we can also define $\underline{\mathit{repmax}}$, which defines the maximal number of times a formula may hold in the indefinite future. As an example of a cardinality operator, $\underline{\mathit{replim}}(\mathit{100}, \mathit{0}, \mathit{3}, E((\mathit{login}, \{(\mathit{user}, \mathit{Alice})\})))$ 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 *con-*

tainers, to data. Transitions are triggered by principals that perform actions. Formally, we describe systems as tuples $(P, \mathit{Data}, \mathit{Event}, \mathit{Container}, \Sigma, \sigma_i, \varrho)$ 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 $\mathit{Container} \rightarrow \mathbb{P}(\mathit{Data})$, to know which set of data is stored in which container; an *alias function* of type $\mathit{Container} \rightarrow \mathbb{P}(\mathit{Container})$ 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 $F \rightarrow \mathit{Container}$. F is a set of identifiers. We need identifiers to correctly model renaming activities. We thus define $\Sigma = (\mathit{Container} \rightarrow \mathbb{P}(\mathit{Data})) \times (\mathit{Container} \rightarrow \mathbb{P}(\mathit{Container})) \times (F \rightarrow \mathit{Container})$. We define transitions between two states by $\varrho : \Sigma \times \mathbb{P}(\mathit{Event}) \rightarrow \Sigma$. 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.

3 A Combined Model

In the usage control model of Section 2, data is addressed by referring to specific representations of this data as event parameters. For instance, $\underline{\mathit{after}}(30, \underline{\mathit{always}}(\underline{\mathit{not}}(E((\mathit{play}, \{(\mathit{obj}, \mathit{song1.mp3})\})))))$ 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 of course not store the entire system history but rather use state machines to record the data state of a system 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.

$$\begin{array}{l} \mathit{EventClass} = \{\mathit{dataUsage}, \mathit{containerUsage}\} \\ \mathit{getclass} : \mathit{Event} \rightarrow \mathit{EventClass} \end{array}$$

$$\begin{array}{ll} \mathit{Data} \subseteq \mathit{ParamValue} & \{(\mathit{obj}, d) \mid d \in \mathit{Data}\} \subseteq \mathit{Params} \\ \mathit{Container} \subseteq \mathit{ParamValue} & \{(\mathit{obj}, c) \mid c \in \mathit{Container}\} \subseteq \mathit{Params} \\ \mathit{Container} \cap \mathit{Data} = \varnothing & \end{array}$$

$$\begin{array}{l} \forall\, e : \mathit{Event} \bullet \mathit{getclass}(e) = \mathit{dataUsage} \Leftrightarrow \\ \qquad \exists\, x : \mathit{ParamValue} \bullet ((\mathit{obj}, x) \in e.2) \wedge x \in \mathit{Data} \\ \forall\, e : \mathit{Event} \bullet \mathit{getclass}(e) = \mathit{containerUsage} \Leftrightarrow \\ \qquad \exists\, x : \mathit{ParamValue} \bullet ((\mathit{obj}, x) \in e.2) \wedge x \in \mathit{Container} \end{array}$$

Step 3: Adding Data State In our semantic model, policies are defined on sequences of events. We want to describe certain situations to be avoided or enforced. However, in practice there usually is an almost infinite number of different sequences of events that lead to the same situation, e.g., the 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 types of formulas we introduce a new set of *state-based operators*, Φ_i (inspired from [3] where no precise semantics is provided):

$$\begin{array}{l} \Phi_i ::= \underline{\mathit{isNotIn}}(\mathit{Data}, \mathbb{P}\, \mathit{Container}) \mid \underline{\mathit{isCombinedWith}}(\mathit{Data}, \mathit{Data}) \mid \\ \qquad \underline{\mathit{isOnlyIn}}(\mathit{Data}, \mathbb{P}\, \mathit{Container}) \end{array}$$

and define $\Phi_i^+ ::= \Phi^+ \mid \Phi_i$. Intuitively, $\underline{\mathit{isNotIn}}(d, C)$ 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 $\underline{\mathit{always}}(\underline{\mathit{isNotIn}}(s, \{c_{net}\}))$ for a network container (any socket) c_{net}. The rule $\underline{\mathit{isCombinedWith}}(d_1, d_2)$ states whether data items d_1 and d_2 are combined in one container. This is useful to express Chinese Wall policies. $\underline{\mathit{isOnlyIn}}$ is the dual of $\underline{\mathit{isNotIn}}$ and expresses that data d can only be in containers of set C. This can be used to express concepts like “data d has been deleted”: $\underline{\mathit{isOnlyIn}}(d, \varnothing)$.

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. An event of class dataUsage can be 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 can 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 case (2), 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 $(\mathit{obj}, d) \in e_1.2$; $(\mathit{obj}, c) \in e_2.2$; 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):

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

We use the 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 $n > 0 \Rightarrow \mathit{states}(t, n) = \varrho(\mathit{states}(t, n-1), t(n-1))$. With the help of $\mathit{refinesEv}_i$ and states, we can 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:

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

On these grounds, we can formally define the semantics of the specific data usage operators in Φ_i with semantics $\models_i \subseteq (\mathit{Trace} \times \mathbb{N}) \times \Phi_i$ which leads to the definition of $\models_i^+ \subseteq (\mathit{Trace} \times \mathbb{N}) \times \Phi_i^+$ 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]).

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

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. For instance, the requirement "no non-anonymized data may leave the system without notification" (where the exact meaning of anonymization is not important) can be enforced by overwriting name and birth date fields with blanks (modification), by blocking messages that are not anonymized (inhibition), or by actually sending the notification if such a data item leaves the system (execution). 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

$$
\begin{array}{l}
\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) \\
\quad \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 \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) \\
\quad \wedge\, x \le \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 t'(n+j) = S \wedge (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 (m < n+j \Rightarrow t'(m) = t(m)) \wedge (t', n+j) \models_i^+ \psi \} \big| \le y \\
\vee\, \exists\, l, u \in \mathbb{N};\ \psi \in \Psi;\ \chi \in \Phi^+ \bullet \varphi = \underline{\mathit{repuntil}}(l,\psi,\chi) \\
\quad \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 \wedge\, \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 t'(n+j) = S \wedge (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 (m < n+j \Rightarrow t'(m) = t(m)) \wedge (t', n+j) \models_i^+ \psi \} \big| \le l \big) \\
\quad \vee\, \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 t'(n+j) = S \wedge (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 (m < n+j \Rightarrow t'(m) = t(m)) \wedge (t', n+j) \models_i^+ \psi \} \big| \le 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^+ \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^+ \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{array}
$$

Fig. 1. Semantics of Φ_i^+

are conveniently expressed as event-condition-action (ECA) rules [6]; whether or not satisfaction of an implementation-level usage control policy entails enforcement of a specification-level policy can be checked automatically [7]. 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, 7]. 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 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 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, 7].

Now, in order to make mechanisms aware of data-flow, we need to extend Φ^- by Φ_i. 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 can simply let $\Phi_i^- ::= \Phi_i \mid \Phi^-$, verbatim reuse the definition of $\models_i$, and directly get the combined semantics of Φ_i^-, $\models_i^-$ (Appendix A). Because of space limitations, we do not provide a formal semantics of entire mechanisms (that is: entire ECA rules, not just conditions) here; however, this straightforwardly generalizes the case without data flow tracking [6].

Step 5: Architecture Our generic architecture is the same for each concrete level of abstraction 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 additional information to the PDP, namely the state of data dissemination, $\sigma \in \Sigma$.

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 [24] for the PDP, any runtime verification algorithm can be used [8]. Due to its generic nature, the same implementation can be reused at different levels of abstraction: 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 2. 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.

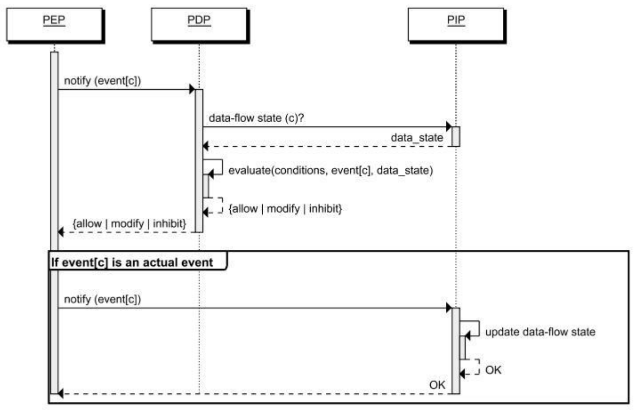
PEP
PDP
PIP
notify (event[c])
data-flow state (c)?
data_state
evaluate(conditions, event[c], data_state)
{allow | modify | inhibit}
{allow | modify | inhibit}
If event[c] is an actual event
notify (event[c])
update data-flow state
OK
OK

Fig. 2. Interplay of PEP, PDP, PIP

Policy management and deployment are out of the scope of this paper. We do not discuss the problem of attaching an initial policy to a data item either; in the implementations discussed below, this is done manually.

Step 6: A coarse model for cross-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 $\lambda : Event \rightarrow L$ 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: $Event_\ell = \{e \in Event : \lambda(e) = \ell\}$ contains the events relevant at layer ℓ, and, using graph notation, $\varrho_\ell = \{(\sigma, E, \sigma') : E \subseteq Event_\ell \wedge \varrho(\sigma, E) = \sigma'\}$ 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 $Event_\ell$, 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 $Event_\ell$. In the implementation, the set of ϱ_ℓ and $Event_\ell$ 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 $\pi : \mathit{Event} \rightarrow 2^{\mathit{Event}}$. 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. For example, we can enforce the policies provided in Appendix B.1.

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]. We recall its main concepts and show how it can be extended with an advanced usage control monitor, providing a second instance of our combined model.

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 applications debugging tool for monitoring, intercepting and modifying network packets from/to an X server. As opposed to [4], by virtue of the usage control runtime monitor, it is able to enforce advanced usage control policies, with temporal and cardinality operators, addressing every instance of the same data at the same time. An example policy we can enforce is presented in Appendix B.2.

Web Browser: Firefox A third instance of our model at the browser level extends an existing usage control extension for the Firefox web browser [25]. 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 are user actions, including “copy”, “paste”, “print”, “save as”, etc., and are performed by a user on web page content. Content can be 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 $\Sigma = (Container \to 2^{Data})$. 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, possibly involving cardinality and temporal operators. As an example, we can enforce the policy provided in Section B.3.

Cross-Layer Enforcement We also implemented cross-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 communication 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 level to the operating system and the windowing system, respectively. For our example, we adopted an existing solution [25] for policy retrieval and hard-coded inter-layer data flow observations.

We consider a social network use case [25] 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 in form of 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."* The implementation-level policies in concrete XML syntax are provided in Appendix B.

The behavior of the combined system is sketched in Figure 3 in Appendix C. In our implementation, at each level 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 (W), it is cached as a file (F), and it is internally represented by the browser in some memory region referenced by a node in the DOM tree (I). Each representation must be protected at its layer of abstraction.

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 level by adding runtime information including the name of the cache file F and the ID of the window W. 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 (Figure 3 in Section C).

From this point onward, all three instantiations of the policy are enforced at different levels of abstraction, as shown in the three following usage attempts. In the first example, the user tries to print the picture; the attempt is intercepted, evaluated against the policy and, after notifying the owner about it, allowed to become an actual usage. In the second example, taking a screenshot of the browser window is intercepted by the X11 monitor. According to the policy, the request is modified; the effect of changing the parameter mask to 0x00 results in returning a black rectangle as screenshot. The last example shows how opening file F, the cached copy of I, is prohibited (i.e., the system call is denied) when the caller process is not Firefox itself (Figure 4 in Section C).

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 at the OS level [26, 27, 3], at the X11 level [4], for Java [11, 12, 28], the .NET CIL [13] and machine languages [14, 15, 29]; at the level of an enterprise service bus [16]; for dedicated applications such as the Internet Explorer [17] and in the context of digital rights management [18–20]. These solutions focus on one of the two aspects of the problem: 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 [29] presents a technique that 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 the formalization and implementation of a generic model and a policy language to express and enforce advanced usage control requirements at arbitrary levels of abstraction. Data flow confinement is also intensely studied at the operating system level [26, 27]; at this level, our work differs in that we aim at enforcing complex usage control policies.

A multitude of policy languages [2, 30–36] 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 analysis which also caters to implicit flows and aims at non-interference assessments [37, 38, 21, 22]: 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 [39], we want to detect these flows at runtime. Implementations of such data-flow tracking system have been realized for the operating system [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 the data flow detection part without full usage control.

In terms of general-purpose usage control models, there are similarities with the models underlying XACML [40], Ponder2 [41] and UCON [42]. 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 [43], and has been used in applications at different level of abstraction, such as the Java Virtual Machine [44] and the Enterprise Service Bus [45]. Data flow is not considered, however.

Complex event processing [23] and runtime monitoring [8] 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 (which is usually encoded in events that are usage-controlled), 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 have provided a model, a language, an architecture and a generic implementation for data-centric usage control enforcement that we instantiated to several abstraction 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 cross-layer enforcement infrastructure for these levels. 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 level); nor take a screenshot (enforced at the X11 level); nor access the cache files (enforced at the OS level).

Because of space restrictions, we have not provided security nor performance analyses. While we do not claim that our system cannot be circumvented, we have some confidence that a reasonable level of security can be attained [25, 10]. 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, and 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, and delegation. Finally, we do not consider the problem of 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 present in this paper, the involved overapproximations quickly lead to a label creep in practice. For instance, in the simple OS-level model, 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 [46] as well as dynamic declassification techniques to overcome this problem. The layers of abstraction that 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.

While we believe that many usage control enforcement problems can be solved by instantiating our framework to a few layers (OS, windowing system, data bases, runtime systems, browsers, word processors, mail clients), we have to understand what precisely these layers consist of. In the screenshot example at the windowing system level, for instance, we need to make sure that the system is not run within a virtual machine which would allow one to take a screenshot outside the virtual machine, thus circumventing the enforcement infrastructure; the solution here is to add another layer that runs in a hypervisor.

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 [10].

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 *SACMAT*, 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. Alexander Pretschner, Judith Rüesch, Christian Schaefer, and Thomas Walter. Formal analyses of usage control policies. In *ARES*, pages 98–105, 2009.
8. Martin Leucker and Christian Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.
9. 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.
10. 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`.
11. M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Security monitor inlining for multithreaded java. In *Proc. ECOOP*, pages pp. 546–569, 2009.
12. 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.
13. 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.
14. U. Erlingsson and F. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. New Security Paradigms Workshop*, pages 87–95, 1999.
15. 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.
16. 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.
17. M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic spyware analysis. In *Proceedings of USENIX Annual Technical Conference*, June 2007.
18. Adobe livecycle rights management es. `http://www.adobe.com/products/livecycle/rightsmanagement/indepth.html`, August 2010.
19. Microsoft. Windows Rights Management Services. `http://www.microsoft.com/windowsserver2008/en/us/ad-rms-overview.aspx`, 2010.

20. 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.
21. Heiko Mantel. Possibilistic definitions of security - an assembly kit. *Computer Security Foundations Workshop, IEEE*, 0:185, 2000.
22. 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.
23. 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.
24. Klaus Havelund and Grigore Rosu. Efficient monitoring of safety properties. *Int. J. Softw. Tools Technol. Transf.*, 6, Aug 2004.
25. 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.
26. 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.
27. 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.
28. 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.
29. 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.
30. R. Iannella (ed.). Open Digital Rights Language v1.1, 2008. `http://odrl.net/1.1/ODRL-11.pdf`.
31. Multimedia framework (MPEG-21) – Part 5: Rights Expression Language, 2004. ISO/IEC standard 21000-5:2004.
32. 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/`.
33. Open Mobile Alliance. DRM Rights Expression Language V2.1, 2008. `http://www.openmobilealliance.org/Technical/release_program/drm_v2_1.aspx`.
34. 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.
35. 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.
36. W3C. The Platform for Privacy Preferences 1.1 (P3P1.1) Specification, 2005. `http://www.w3.org/TR/2005/WD-P3P11-20050104/`.

37. J. Rushby. Noninterference, transitivity and channel-control security policies, 1992.
38. 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.
39. 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.
40. E. Rissanen. Extensible access control markup language v3.0. `http://docs.oasis-open.org`, 2010.
41. 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.
42. J. Park and R. Sandhu. The UCON ABC usage control model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174, 2004.
43. 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.
44. 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.
45. 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.
46. Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. In *PLDI*, pages 193–205, 2008.

A Past formulas with data flow: Φ_i^-

Mechanisms, or ECA rules, are specified in a past temporal logic Φ^- [6] that we extend with the data flow semantics as for the future language. This extended past language is called Φ_i^-.

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

Its semantics is defined by $\models_i^- \supset \models^-$ as follows.

$$
\begin{array}{l}
\forall\, t \in \mathit{Trace}, n \in \mathbb{N}, \varphi \in \mathit{\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 \mathit{\Phi}_i \wedge (t,n) \models_i \varphi \\
\vee\, \exists\, \psi \in \mathit{\Phi}_i^- \bullet \varphi \in \{\underline{\mathit{not}}(\psi), \underline{\mathit{not}}^-(\psi)\} \wedge \neg\, ((t,n) \models_i^- \psi) \\
\vee\, \exists\, \psi, \chi \mathit{\Phi}_i^- \bullet \varphi \in \{\underline{\mathit{or}}(\psi,\chi), \underline{\mathit{or}}^-(\psi,\chi)\} \wedge ((t,n) \models_i^- \psi \vee (t,n) \models_i^- \chi) \\
\vee\, \exists\, \psi, \chi \in \mathit{\Phi}_i^- \bullet \varphi = \underline{\mathit{since}}^-(\chi,\psi) \\
\qquad \wedge\, (\exists\, u \in \mathbb{N} \bullet u \le n \wedge (t, n-u) \models_i^- \chi \wedge (\forall\, v \in \mathbb{N} \bullet u < v \le n \Rightarrow (t, n-v) \models_i^- \psi) \\
\qquad\qquad \vee\, \forall\, v \in \mathbb{N} \bullet v \le u \Rightarrow (t, n-v) \models_i^- \psi) \\
\\
\vee\, \exists\, i \in \mathbb{N};\ \psi \in \mathit{\Phi}_i^- \bullet \varphi = \underline{\mathit{before}}^-(i,\psi) \wedge i \le n \wedge (t, n-i) \models_i^- \psi \\
\vee\, \exists\, l, x, y \in \mathbb{N};\ \psi \in \mathit{\Psi} \bullet \varphi = \underline{\mathit{replim}}^-(l,x,y,\psi) \\
\qquad \wedge\, x \le \sum_{j=0}^{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\qquad t'(n-j) = S \wedge (n \ge 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 \ge m > n-j \Rightarrow t'(m) = t(m)) \wedge (t', n-j) \models_i^- \psi \} \big| \le y \\
\vee\, \exists\, l, u \in \mathbb{N};\ \psi \in \mathit{\Psi};\ \chi \in \mathit{\Phi}^- \bullet \varphi = \underline{\mathit{repsince}}^-(l,\chi,\psi) \\
\qquad \wedge\, \big(u \le n \wedge (t, n-u) \models_i^- \chi \wedge \forall\, v \in \mathbb{N} \bullet u < v \le n \Rightarrow \neg((t, n-v) \models_i^- \chi) \\
\qquad\qquad \wedge\, \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 \ge 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 \ge m > n-j \Rightarrow t'(m) = t(m)) \wedge (t', n-j) \models_i^- \psi \} \big| \le l \big) \\
\qquad \vee\, \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\qquad t'(n-j) = S \wedge (n \ge 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 \ge m > n-j \Rightarrow t'(m) = t(m)) \wedge (t', n-j) \models_i^- \psi \} \big| \le l \\
\vee\, \exists\, \psi, \chi \in \mathit{\Phi}_i^- \bullet \varphi \in \{\underline{\mathit{and}}(\psi,\chi), \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 \mathit{\Phi}_i^- \bullet \varphi \in \{\underline{\mathit{implies}}(\psi,\chi), \underline{\mathit{implies}}^-(\psi,\chi)\} \wedge (t,n) \models_i^- \underline{\mathit{or}}^-(\underline{\mathit{not}}^-(\psi), \chi) \\
\vee\, \exists\, \psi \in \mathit{\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 \mathit{\Phi}_i^- \bullet \varphi = \underline{\mathit{within}}^-(i,\psi) \wedge i < n \wedge (t,n) \models_i^- \bigvee_{x=1}^{i} \underline{\mathit{before}}^-(i,\varphi) \\
\vee\, \exists\, i \in \mathbb{N};\ \psi \in \mathit{\Phi}_i^- \bullet \varphi = \underline{\mathit{during}}^-(i,\psi) \wedge i < n \wedge (t,n) \models_i^- \bigwedge_{x=1}^{i} \underline{\mathit{before}}^-(x,\varphi) \\
\vee\, \exists\, l \in \mathbb{N};\ \psi \in \mathit{\Psi} \bullet \varphi = \underline{\mathit{repmax}}^-(l,\psi) \wedge (t,n) \models_i^- \underline{\mathit{repsince}}^-(l, \underline{\mathit{false}}, \psi)
\end{array}
$$

B Implementation-Level Policies

The following policies are expressed in a concrete syntax that reflects the abstract syntax for ECA rules in Section 3, step 4: trigger event, condition, action to be taken in response. Note the use of the *dataUsage* attribute of the *parameter* node. Its purpose is twofold: firstly, the presence of a *type* attribute identifies the node as the compulsory *(obj, object_name)* attribute/value pair. Secondly, it instructs the system on how this value should be interpreted (and possibly transformed), whether as data or as container. Also note the use of the *isTry* parameter to distinguish attempted (value="true") from actual usages (value="false").

B.1 Operating System

The first policy is an example from the DRM world: a file, song.mp3, can be used, i.e. opened, (lines 12-15) at most 4 further times and within 30 seconds (1 timestep = 1 second) after the first use (lines 8-27); further attempts of opening the file will result in opening a predefined error message (lines 28-34). We provide it to demonstrate the use of complex conditions.

```
 1 <controlMechanism>
 2  <id>OS_DRM_example</id>
 3  <triggerEvent>
 4   <id>open</id>
 5    <parameter name="obj" value="song.mp3" type="dataUsage"/>
 6    <parameter name="isTry" value="true"/>
 7  </triggerEvent>
 8  <condition>
 9  <or>
10    <not><before timeInterval="30">
11     <always><not>
12      <event>
13       <id>open</id>
14        <parameter name="obj" value="song.mp3" type="dataUsage"/>
15      </event>
16     </not></always>
17    </before></not>
18    <not>
19     <repmax limit="5">
20      <event>
21       <id>open</id>
22        <parameter name="obj" value="song.mp3" type="dataUsage"/>
23      </event>
24     </repmax>
25    </not>
26   </or>
27  </condition>
28  <actions>
29   <allow>
30    <modify>
31     <parameter name="obj" value="/etc/UCmon/expired.msg" />
32    </modify>
33   </allow>
34  </actions>
35 </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.

As a second example, we present a policy that forbids data-disclosure by using a state-based operator:

```
<controlMechanism>
 <id>OS_Disclosure_example</id>
 <triggerEvent>
  <id>write</id>
   <parameter name=”isTry” value=”true”/>
 </triggerEvent>
 <condition>
  <not>
   <isNotIn data=”song.mp3”>
    <containers>
     <container>Net</container>
    </containers>
   </isNotIn>
  </not>
 </condition>
 <actions>
  <inhibit/>
 </actions>
</controlMechanism>
```

In this example, we forbid every write system call that sends the song (the data) stored in “song.mp3” over the network, i.e. to a socket descriptor. “song.mp3” and “Net” are names for containers and are translated into containers according to the function F at runtime. In particular, “Net” is a reserved name for the container “Cnet” that stands for “the network”. Every container representing a socket descriptor is aliased to “Cnet”, therefore writing data to a socket corresponds to writing it to “Cnet”.

As this policy applies to *dataUsage* events, the value of parameter *data* of operator *isNotIn* is interpreted as “the data stored in the container with this name”.

Finally, as a third example, we present the operating system policy used in the cross-layer enforcement example.

```
1  <controlMechanism>
2   <id>OS_Restrict_File_Usage</id>
3   <triggerEvent>
4    <id>open</id>
5     <parameter name=”obj” value=”cacheFile” type=”dataUsage”/>
6     <parameter name=”isTry” value=”true”/>
7   </triggerEvent>
8   <condition>
9    <XPathEval>
10    /triggerEvent/parameter[@name='PNAME']/@value!='c:\\Firefox\\firefox.
          exe'
11   </XPathEval>
12  </condition>
13  <actions>
14   <inhibit/>
15  </actions>
16 </controlMechanism>
```

In this example, every attempt of opening file *cacheFile* (lines 3-7) is intercepted and forbidden (lines 13-15) if the caller process is different from Firefox (lines 8-12). Note the use of XPath inside the condition block, in order to test values belonging to the trigger event. In the current implementation we identify the Firefox process by the name and path of the executable, because the process ID is different in every execution. This introduces obviously some security vulnerabilities, and other solutions can be used to overcome this issue.

In the cross-layer example, when such a policy is received from the remote server, values like *cacheFile* and the path of the application are replaced by placeholders and filled in at runtime by the web browser PEP (Appendix C).

B.2 Windowing System

```
 1  <controlMechanism>
 2   <id>X11_Screenshot</id>
 3   <triggerEvent>
 4    <id>GetImage</id>
 5     <parameter name=”obj” value=”0x1a00005” type=”dataUsage”/>
 6     <parameter name=”isTry” value”true”/>
 7   </triggerEvent>
 8   <condition>
 9    <true />
10   </condition>
11   <actions>
12    <allow>
13      <modify>
14       <parameter name=”planeMask” value=”0x0” />
15      </modify>
16    </allow>
17   </actions>
18  </controlMechanism>
```

In this example, the enforcement mechanism prevents the X client from taking a screenshot (X11 action *GetImage*, line 4) of the content of window 0x1a00005 (line 5; in the cross-layer example, this data is filled in by the web browser PEP). If a client sends a request for a screenshot of that window, the action is permitted (line 12), but the parameter *planeMask* is modified to the value 0x0 (line 14). *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.

As mentioned before, if such a policy is received from the server, parameters like the window ID must be replaced by the proper runtime values.

B.3 Web Browser

The first example is from the social network scenario: a user is allowed to print a profile picture (lines 3-7) only once (line 10). Further attempts of printing are forbidden (line 19).

```
 1  <controlMechanism>
 2   <id>Browser_Print</id>
 3   <triggerEvent>
 4    <id>print</id>
 5     <parameter name=”obj” value=”img_profile” type=”dataUsage”/>
 6     <parameter name=”isTry” value”true”/>
 7   </triggerEvent>
 8   <condition>
 9    <not>
10     <repmax limit=”1”>
```

```
     <event>
      <id>print</id>
       <parameter name=”obj” value=”img_profile” type=”dataUsage”/>
     </event>
    </repmax>
   </not>
  </condition>
  <actions>
   <inhibit/>
  </actions>
</controlMechanism>
```

A second example from the same context is the following:

```
1  <controlMechanism>
2   <id>Browser_Submit</id>
3    <triggerEvent>
4     <id>submit</id>
5      <parameter name=”obj” value=”lbl_address” type=”dataUsage”/>
6      <parameter name=”isTry” value”true”/>
7    </triggerEvent>
8   <condition>
9    <true/>
10  </condition>
11  <actions>
12   <allow/>
13   <execute>
14    <action>
15     <id>SendNotification</id>
16      <parameter name=”msg” value=”ErrMsg_1” />
17    </action>
18   </execute>
19  </actions>
20 </controlMechanism>
21
22 <controlMechanism>
23  <id>Browser_Clip</id>
24   <triggerEvent>
25    <id>copy_ext</id>
26     <parameter name=”obj” value=”lbl_address” type=”dataUsage”/>
27     <parameter name=”isTry” value=”true”/>
28   </triggerEvent>
29  <condition>
30   <true />
31  </condition>
32  <actions>
33   <inhibit/>
34  </actions>
35 </controlMechanism>
```

With these two mechanisms, a notification is sent to the (social network) user (lines 14-17) if the content of the address field on its profile page (line 5) is sent as a part of a submission form (lines 3-7) (e.g., when posted on a bulletin board or sent using a webmail client like Gmail) and cannot be copied to the system clipboard (lines 24-28, 33).

Note that in our implementation there is a local clipboard for the Firefox user which is independent from the system clipboard (which, in turn, is managed by the policies that apply to the X11 layer). Every time the user invokes a “copy” command, two events are triggered, “copy_int”, that copies the current selection into our local clipboard, and “copy_ext”, that does the same for the system clipboard. This differentiation allows us to forbid leakage of data outside the browser environment (line 25) without forbidding “copy&paste” actions inside.

Another example, similar to the one mentioned in the cross-layer use case, is the following:

```
 1 <controlMechanism>
 2  <id>Browser_Save_Pic</id>
 3  <triggerEvent>
 4   <id>save</id>
 5    <parameter name=”obj” value=”img_profile” type=”dataUsage”/>
 6    <parameter name=”isTry” value”true”/>
 7  </triggerEvent>
 8  <condition>
 9   <true />
10  </condition>
11  <actions>
12   <inhibit/>
13  </actions>
14 </controlMechanism>
```

It the user tries to save the picture *img_profile* (directly, or as a part of a bigger container, like the frame or the page) (lines 3-7) then the action is forbidden (lines 11-13).

In the cross-layer use case presented in Section 4, policies akin to these three is instantiated by replacing the content of some parameters, like *src*, with the appropriate runtime values.

C Cross-Layer Enforcement

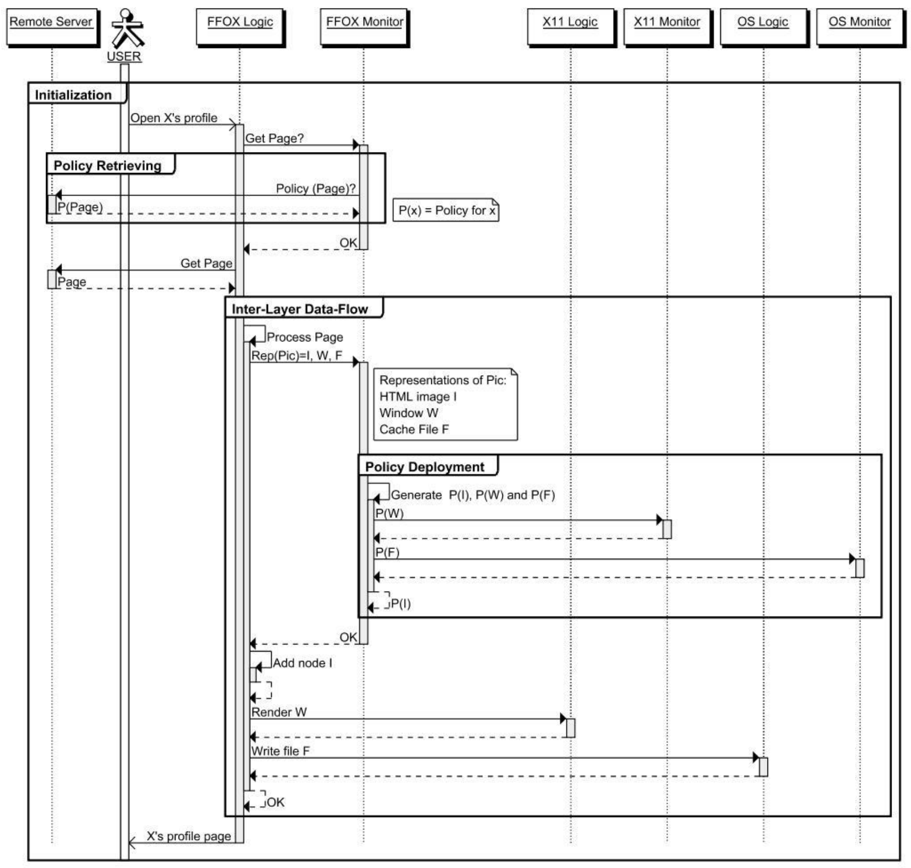

Fig. 3. Policy distribution in the cross-layer setting

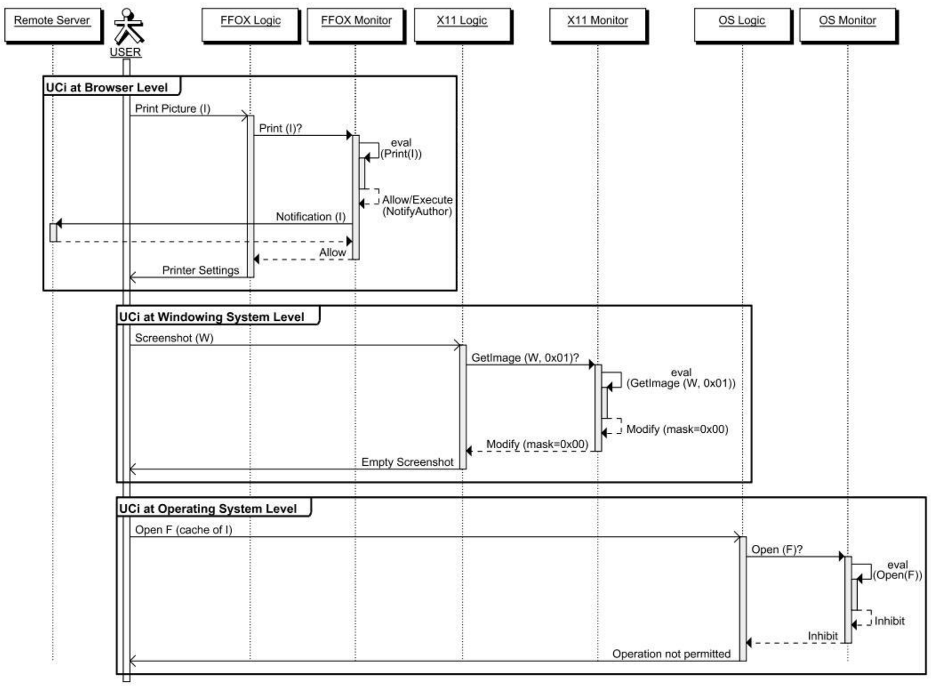

Fig. 4. Policy enforcement in the cross-layer setting