# Generic Semantics Specification and Processing for Inter-System Information Flow Tracking

Pascal Birnstill*, Christoph Bier*, Paul Wagner† and Jürgen Beyerer*
*Fraunhofer Institute of Optronics, System Technologies and Image Exploitation IOSB, Karlsruhe, Germany
Email: {pascal.birnstill|christoph.bier|juergen.beyerer}@iosb.fraunhofer.de
†Karlsruhe Institute of Technology, Karlsruhe, Germany
Email: paul.wagner@student.kit.edu

***Abstract*—Data usually takes different shapes and appears as files, windows, processes' memory, network connections, etc. Information flow tracking technology keeps an eye on these different representations of a data item. Integrated with a usage control (UC) infrastructure, this allows us to enforce UC requirements on each representation of a protected data item. To enable UC enforcement in distributed settings, we need to be able to track information flows across system boundaries. In this paper we introduce a state-based information flow model for tracking explicit flows between systems equipped with UC technology. We demonstrate the applicability of our approach by means of an instantiation in the field of video surveillance, where systems are increasingly accessed via insecure mobile applications. Based on usage control and inter-system information flow tracking, we show how video data transmitted from a video surveillance server to mobile clients can be protected against illegitimate duplication and redistribution after receipt.**

***Index Terms*—Information flow tracking, explicit flows, information flow semantics specification, distributed usage control, policy enforcement**

## I. Introduction

*Distributed usage control (DUC)* is a generalization of access control that also addresses obligations regarding the future usage of data, particularly in distributed settings [1]. UC policies are typically specified via events. Events are intercepted or observed by so-called *policy enforcement points (PEP)* as illustrated in Fig. 1. PEPs forward events to a *policy decision point (PDP)*, which evaluates them against policies. The PDP replies with an *authorization action*, such as *allow, modify, inhibit*, and *delay*, and triggers *obligations*.

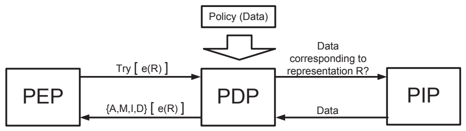

Fig. 1. Generic UC Architecture with Information Flow Tracking

Because data usually comes in different representations – an image can be a pixmap, a file, a leaf in the DOM tree of a website, a Java object, etc.– UC mechanisms have been augmented with information flow tracking technology [2]. One can then specify policies not only for specific fixed representations of a data item, but also on *all* representations of that data item. Policies then do not need to rely on events, but can forbid specific representations to be created, also in a distributed setting [3]. In other words, information flow tracking answers the question into which representations within the (distributed) system monitored data has been propagated.

In order to perform information flow tracking across different applications, different layers of abstraction of a system or across different systems, a multitude of PEPs, each observing an individual set of information flow-relevant events, has to be integrated into the information flow tracking system. The so-called *policy information point (PIP)* interprets the information flow semantics of events and accordingly keeps track of new representations of data being created and of information flows between representations. By this means, when evaluating an event concerning a container (such as a file, process, or window), the PDP can ask the PIP whether this container is a representation of a protected data item, for which a policy must be enforced (cf. Fig. 1).

This work is also explicitly motivated by the increasing number of mobile apps for accessing video surveillance cameras and systems on the market and by the observation that meanwhile video surveillance is entering highly sensitive areas such as hospitals and nursing facilities. While these apps facilitate the cooperation of control rooms and security personnel on-site, we observe that in comparison to heavily secured control rooms the mobile devices being used fall critically short in terms of security mechanisms for protecting the sensitive data captured by surveillance systems. Obviously, the appearance of leaked surveillance footage showing a patient in an emergency situation on a video sharing portal on the Internet is in the interest of neither the patient nor the hospital. We thus instantiate our approach for protecting video data provided by a video surveillance server against illegitimate duplication and redistribution by mobile clients after receipt.

We address the following problems: We generalize an approach to inter-layer information flow tracking introduced by Lovat [4] to additionally cover inter-system flows so as to enable monitoring of flows of protected data between systems equipped with UC enforcement mechanisms. This approach is suitable for proof-of-concept implementation since it is lightweight. Yet it is prone to over-approximations requiring an extension with monitoring technology of higher precision in future work (cf. VI). Plugging new PEPs into existing

UC infrastructures requires information flow semantics of the intercepted events to be deployed at the PIP. We introduce a generic set of primitives for specifying information flow semantics in a uniform syntax to be used by developers of monitors (PEPs). These primitives are derived from analyses of various scenarios in which information flow tracking has been instantiated for UC, such as [2], [5], [6]. Across system boundaries, information flows have to be handled asynchronously, triggered by different events on the particular machines. For this, we specify a protocol for processing inter-layer and inter-system flows based on our semantics description primitives. We thus facilitate UC enforcement on the granularity of representations in distributed settings.

This work is structured as follows. After explaining the formal information flow model of Harvan and Pretschner [2] in Sec. II, we introduce our information flow semantics primitives in Sec. III. In Sec. IV we extend the model so as to allow uniform processing extension of inter-layer and inter-system information flows. We present an instantiation of our approach for protecting video data streamed to a client on behalf of the originating video surveillance server in Sec. V. Eventually we discuss related work in Sec. VI and conclude in Sec. VII.

## II. Information Flow Model

Our approach to information flow modeling is based on works of Harvan and Pretschner [2], [5]. An information flow model is a transition system that captures the flow of data throughout a system. Transitions of the state are triggered by events that are observed by monitors, such as PEPs of a UC infrastructure. A system's information flow tracking component, the PIP, interprets events given information flow semantics provided by monitors when being deployed.

The state of the information flow model comprises three aspects. It reflects which data units are in which container, where a container may be a file, a window in the graphical user interface, an object in a Java virtual machine, a network connection, etc. The state also captures *alias relations* between containers, which express that a container is implicitly updated whenever some other container is updated. This happens, for instance, when processes share memory. Finally, the state comprises different *names* that identify a container, e.g., a file may not only be accessible by its file name, but also by a file handle.

### A. Formal Model.

As introduced by Pretschner and Harvan in [2], [5] the formal information flow model is a tuple $(D, C, F, \Sigma, E, R)$. $D$ is the set of data for which UC policies exist. $C$ is the set of containers in the system. $F$ is the set of names. $\Sigma = (C \to 2^D) \times (C \to 2^C) \times (F \to C)$ is the set of possible states, which consists of the *storage function* $s : C \to 2^D$, the *alias function* $l : C \to 2^C$, and the *naming function* $f : F \to C$. Chains of aliases are addressed using the reflexive transitive closure $l^*$ of the alias function. The initial state of the system is denoted as $\sigma_I \in \Sigma$, where the state of the storage function $s$ is given by the initial representation of a data item a UC policy refers to. *Events* $E$ are observed actions that trigger changes of the storage function $s$, the alias function $l$, or the naming function $f$. These changes are described in a (deterministic) transition relation $R \subseteq \Sigma \times E \times \Sigma$. We describe updates to the functions $s$, $l$, and $f$ using a notation introduced in [2].

We describe updates to the functions $s$, $l$, and $f$ using a notation introduced in [2]: Let $m : S \to T$ be any mapping and $x \in X \subseteq S$ a variable. Then $m[x \leftarrow expr]_{x \in X} = m'$ with $m' : S \to T$ is defined as

$$m'(y) = \begin{cases} expr & if\ y \in X \\ m(y) & otherwise. \end{cases}$$

## III. Generic Primitives for Information Flow Semantics

For any PEP, $R$ is specified in an *information flow semantics*, which the PEP deploys on the PIP when being added to a UC infrastructure. For each event intercepted by a PEP, an information flow semantics specifies the state changes of the functions $s$, $l$, and $f$ using generic primitives that we introduce in the following. When processing an event according to an information flow semantics (e.g., Listing 2), the PIP picks the action description for the event, converts event parameters in order to match the signatures of the contained semantics primitives (i.e., it implicitly applies $f$ or $s$ on a given parameter: $F \xrightarrow{f} C \xrightarrow{s} D$), and finally modifies its state according to the given primitives.

### A. Primitives for Updating the Storage Function

The storage function keeps track of representations, i.e., mappings between data units and containers. We employ it for modeling the actual information flows.

$$\begin{aligned} &flow(container\ c,\ data\ \{d_i\}_{1 \le i \le n \in \mathbb{N}}) : \\ &\qquad s[c \leftarrow s(c) \cup \{d_i\}] \end{aligned} \tag{1}$$

The $flow$ primitive (cf. Eq. 1) indicates an information flow of a set of data units $\{d_i\}_{1 \le i \le n \in \mathbb{N}}$ into the container $c$. This primitive is used to model that a process creates a new file, a child process, or that a file is copied. Data will then also flow into containers of processes that have a read handle on this file.

$$\begin{aligned} &flow\_to\_rtc(container\ c,\ data\ \{d_i\}_{1 \le i \le n \in \mathbb{N}}) : \\ &\qquad \forall t \in l^*(c) : s[t \leftarrow\ s(t) \cup \{d_i\}] \end{aligned} \tag{2}$$

The $flow\_to\_rtc$ primitive (cf. Eq. 2) models a flow into containers of the reflexive transitive closure $l^*(c)$ of container $c$. It is used for processes reading from a file, writing to a file, or getting data from the system clipboard.

$$\begin{aligned} &clear(container\ c) : \\ &\qquad s[c \leftarrow \varnothing] \end{aligned} \tag{3}$$

We employ the $clear$ (cf. Eq. 3) primitive whenever a container is deleted, such as when deleting a file, closing a window, killing a process, etc.

## B. Primitives for Updating the Alias Function

The alias function maintains relations between containers that lead to implicit flows. Whenever data items flow to container $c_{from}$, they also flow into the aliased container $c_{to}$.

$$\begin{aligned} &create\_alias(container\ c_{from},\ container\ c_{to}) : \\ &\qquad l[c_{from} \leftarrow l(c_{from}) \cup c_{to}] \end{aligned} \tag{4}$$

The primitive $create\_alias$ shown in Eq. 4 adds an unidirectional alias from container $c_{from}$ to container $c_{to}$ to the alias function of $c_{from}$. We use unidirectional aliases for memory-mapped file I/O, if a process has read-only access to the file (cf. `mmap` system call on POSIX-compliant UNIX and Linux systems).

$$\begin{aligned} &create\_bidir\_alias(container\ c_{from},\ container\ c_{to}) : \\ &\qquad l[c_{from} \leftarrow l(c_{from}) \cup c_{to}], \\ &\qquad l[c_{to} \leftarrow l(c_{to}) \cup c_{from}] \end{aligned} \tag{5}$$

We add bidirectional aliases using the primitive $create\_bidir\_alias$ (cf. Eq. 5). Examples to be modeled with bidirectional aliases include creating a new window, or a process having read and write access to a file.

$$\begin{aligned} &rm\_alias\_locally(container\ c_{from},\ container\ c_{to}) : \\ &\qquad l[c_{from} \leftarrow l(c_{from}) \setminus c_{to}] \end{aligned} \tag{6}$$

The primitive $rm\_alias\_locally$ removes an unidirectional alias from $c_{from}$ to $c_{to}$, e.g., aliases added using the primitive $create\_alias$ (cf. Eq. 4).

$$\begin{aligned} &rm\_alias\_globally(container\ c_{to}) : \\ &\qquad \forall c \in C : l[c \leftarrow l(c) \setminus c_{to}] \end{aligned} \tag{7}$$

In some cases we also need to remove an unidirectional alias from all containers in $C$, e.g., in case $c$ is a file, which is deleted. For this, we employ the primitive $rm\_alias\_globally$ as shown in Eq. 7.

$$\begin{aligned} &rm\_bidir\_alias\_locally(container\ c_{from},\ container\ c_{to}) : \\ &\qquad l[c_{from} \leftarrow l(c_{from}) \setminus c_{to}], \\ &\qquad l[c_{to} \leftarrow l(c_{to}) \setminus c_{from}] \end{aligned} \tag{8}$$

Bidirectional aliases as added using the primitive $create\_bidir\_alias$ (cf. Eq. 5) are removed using the primitive $rm\_bidir\_alias\_locally$ as shown in Eq. 8.

$$\begin{aligned} &clear\_aliases(container\ c) : \\ &\qquad l[c \leftarrow \varnothing] \end{aligned} \tag{9}$$

$clear\_aliases$ removes all aliases with the given container as source from the state of the alias function (cf. Eq. 9), e.g., to clean up if a container is deleted.

## C. Primitives for Updating the Naming Function

The naming function maps different names to the same container, e.g., files can be addressed via file names and also via file handles or hard links; in the Windows operating system, we can identify a window via a window handle and also via a window name.

$$\begin{aligned} &add\_naming(naming\ n,\ container\ c) : \\ &\qquad f[n \leftarrow c] \end{aligned} \tag{10}$$

A new name $n$ for a container $c$ is added using the primitive $add\_naming$ (cf. Eq. 10) and removed via $rm\_naming$:

$$\begin{aligned} &rm\_naming(naming\ n) : \\ &\qquad f[n \leftarrow nil] \end{aligned} \tag{11}$$

## IV. Inter-Layer and Inter-System Flows

So far, our primitives do not capture *inter-layer* and *inter-system* information flows. When using the term *inter-layer*, we refer to flows between different layers of abstraction, e.g., between an application and the operating system. *Inter-system* flows take place whenever data is exchanged between systems over a network connection. We introduce an information flow model extension for monitoring such flows, which requires that an event indicating an *incoming* flow is matched to a preceding *outgoing* event on another system or layer of abstraction.

### A. Extended Information Flow Model

As an example, consider the transfer of video data from a streaming server to a client. Assume further that both, server and client, are equipped with PEPs that are capable of intercepting outgoing respectively incoming events as well as with local UC infrastructures. The server side PEP observes an outgoing event indicating a flow from a local container to another container representing the network connection to the client. When receiving data of the video stream via this connection, the client side PEP observes a related incoming event. Finally, when either the client disconnects from the video stream or the server closes the connection, a third event is observed, which terminates the flow. Initially, these events are independent from the perspective of both PIPs. Detecting an inter-system flow requires that both events are interpreted at both PIPs requiring according *remote* information flow semantics, which are provided by the respective PEPs and exchanged between PIPs.

Within an information flow semantics a so-called *scope* specification indicates that an event is related to an event on another system (or layer of abstraction). The particular events are matched to a scope by means of a *scope name* parameter, which is a label for a flow mutually known by two systems (or layers of abstraction). We thus extend the information flow model with a set of scopes $SCOPE$, and the state with the following two mappings: The *intermediate container function* $\iota : SCOPE \rightarrow C$ maps each scope to an intermediate container $c_\iota \in C$. The *scope state function* $\varsigma : SCOPE \rightarrow \{\text{ACTIVATED}, \text{DEACTIVATED}\}$ indicates currently open scopes. Intermediate containers of different systems are distinct containers, which are mapped on each other by means of scopes and virtually represent the connection. Each event belongs to at most one inter-layer (XLAYER) or inter-system (XSYSTEM) scope. In the initial state $\sigma_I$ of the system there is one intermediate container $c_\iota$ for each scope $\iota$ and $\varsigma(sc)$ is DEACTIVATED for all $sc \in SCOPE$. Three attributes of a scope define how the model state is modified when processing an according event:

$$\begin{aligned}
&X_{SCOPE} : \Sigma \times E \rightarrow SCOPE \times BEHAVIOR \\
&\times DELIMITER \times INTER \\
&DELIMITER = \{\text{OPEN}, \text{CLOSE}, \text{NONE}\} \\
&BEHAVIOR = \{\text{IN}, \text{OUT}, \text{INTRA}\} \\
&INTER = \{\text{XLAYER}, \text{XSYSTEM}\}
\end{aligned}$$

The $DELIMITER$ of a scope describes whether an event indicates a new XLAYER or XSYSTEM flow. The delimiter OPEN changes the state of the scope within which the event is processed to ACTIVATED. The $BEHAVIOR$ describes whether the event indicates an outgoing flow to (OUT), or an incoming flow (IN) from another system or layer of abstraction. The $BEHAVIOR$ of a scope affects the processing of semantics primitives when handling XLAYER/XSYSTEM flows as will be described in Sec. IV-C (INTRA is the default behavior, i.e., a flow within a layer of abstraction, which does not affect the interpretation of primitives). $INTER$ differentiates between XSYSTEM and XLAYER flows.

### B. Selecting the Appropriate Scope Semantics for an Event

For each event type a PEP's information flow semantics contains *action descriptions*, which specify its interpretation in terms of information flow using semantics primitives (cf. Sec. III). An action description also includes an ordered list of all scope specifications that possibly apply for this event type.The event notification only contains the scope (as a name-value pair, where the value is the scope itself). When processing an event, the PIP needs to check, in the given order of the action description, which scope specification is applicable. For each scope specification, the PIP evaluates the following three conditions:

1) Does the scope name in the scope specification match the name of a parameter provided in the parameter list of the event notification?
2) If $DELIMITER$ = OPEN in the scope specification: scope deactivated?
3) If $DELIMITER$ = NONE or $DELIMITER$ = CLOSE: scope activated?

If only one of the conditions is not fulfilled, the respective scope is skipped. The ordered list is processed until the matching scope specification $X_{SCOPE}$ is found.

### C. Scope Processing

The transition relation $R$ is modified when processing a scope specification. Algorithm 1 describes how $R$ is modified to obtain $R_{mod}$, i.e., the transition relation for XLAYER or XSYSTEM flows. $R[left \overset{\text{subst.}}{\Longleftarrow} right]$ denotes that the term of $R$ on the left is substituted with the term on the right in $R_{mod}$. If the delimiter of the scope equals OPEN, the scope is activated (cf. line 6); if the delimiter equals CLOSE, the scope is deactivated after handling the event (cf. line 17). In between (cf. line 8 ff.), depending on the scope's behavior, either the left argument (target) (cf. line 12 ff.) or the right argument (source) of the storage function primitives $flow$ or $flow\_to\_rtc$ is substituted with the scope's intermediate container. $R_{mod}$ is then applied on the state $\sigma$ (cf. line 16). In case of an XSYSTEM flow, the PIP needs to enable its remote counterpart to process the given event. As described in Sec. V-A this is achieved by forwarding a remote information flow semantics to the remote PIP.

**Algorithm 1** Processing an XSYSTEM scope

```
1:  procedure $R_{inter}(\sigma, e)$
2:      $(scope, behav, delim, inter) \longleftarrow X_{SCOPE}(\sigma, e)$
3:      if $scope \neq \emptyset$ then
4:          $ic \longleftarrow \iota(scope)$
5:          $R_{mod} \longleftarrow R$
6:          if $delim = \text{OPEN}$ then
7:              $\sigma \longleftarrow \varsigma[scope \leftarrow \text{ACTIVATED}]$
8:          if $behav = \text{OUT}$ then
9:              $R_{mod} \longleftarrow R_{mod}\big[s[c \leftarrow s(c) \cup \{d_i\}]$
                $\overset{\text{subst.}}{\Longleftarrow} s[ic \leftarrow s(ic) \cup \{d_i\}]\big]$
10:             $R_{mod} \longleftarrow R_{mod}\big[\forall t \in l(c) : s[t \leftarrow s(t) \cup \{d_i\}]$
                $\overset{\text{subst.}}{\Longleftarrow} \forall t \in l(ic) : s[t \leftarrow s(t) \cup \{d_i\}]\big]$
11:             $R_{mod} \longleftarrow R_{mod}\big[\forall t \in l^*(c) : s[t \leftarrow s(t) \cup \{d_i\}]$
                $\overset{\text{subst.}}{\Longleftarrow} \forall t \in l^*(ic) : s[t \leftarrow s(t) \cup \{d_i\}]\big]$
12:         if $behav = \text{IN}$ then
13:             $R_{mod} \longleftarrow R_{mod}\big[s[c \leftarrow s(c) \cup \{d_i\}]$
                $\overset{\text{subst.}}{\Longleftarrow} s[c \leftarrow s(c) \cup s(ic)]\big]$
14:             $R_{mod} \longleftarrow R_{mod}\big[\forall t \in l(c) : s[t \leftarrow s(t) \cup \{d_i\}]$
                $\overset{\text{subst.}}{\Longleftarrow} \forall t \in l(c) : s[t \leftarrow s(t) \cup s(ic)]\big]$
15:             $R_{mod} \longleftarrow R_{mod}\big[\forall t \in l^*(c) : s[t \leftarrow s(t) \cup \{d_i\}]$
                $\overset{\text{subst.}}{\Longleftarrow} \forall t \in l^*(c) : s[t \leftarrow s(t) \cup s(ic)]\big]$
16:         $\sigma \longleftarrow R_{mod}(\sigma, e)$
17:         if $delim = \text{CLOSE}$ then
18:             $\sigma \longleftarrow \varsigma[scope \leftarrow \text{DEACTIVATED}]$
19:             $\sigma \longleftarrow s[ic \leftarrow \varnothing]$
20:     else
21:         $\sigma \longleftarrow R(\sigma, e)$
22:     return $\sigma$
```

## V. Instantiation

We implemented XSYSTEM information flow tracking for a scenario concerning video data provided by a streaming server. It enables us to enforce the UC requirement of preventing redistribution of the video data after receipt on client systems, such as mobile apps for video surveillance systems as mentioned earlier. For this, we need to (i) deploy an according policy at the UC infrastructure of the client, (ii) track the flow of video data from the server to the client, and (iii) monitor the video data at the client so as to inhibit further representations of the data to be created. We achieve (i) and (ii) in the following protocol steps:

1) Intercept an event signaling the *outgoing* data at the server side PEP
2) Evaluate the event against an according policy at the server side PDP
3) Deploy a policy for the data at the client side PDP
4) Process the event at the server side PIP

5) Create a new representation of the video data at the client side PIP
6) Process the outgoing event also at the client side PIP
7) Intercept an event signaling the *incoming* data at the client side PEP
8) Evaluate the event at the client side PDP
9) Process the event at the client side PIP
10) Intercept an event signaling *close* of the connection at the server side PEP
11) Process the event at the server side PIP
12) Process the close event also at the client side PIP

We explain the details of the protocol steps by means of Fig. 2, where an additional component, a system's *policy management point (PMP)*, takes care of policy shipment and deployment.

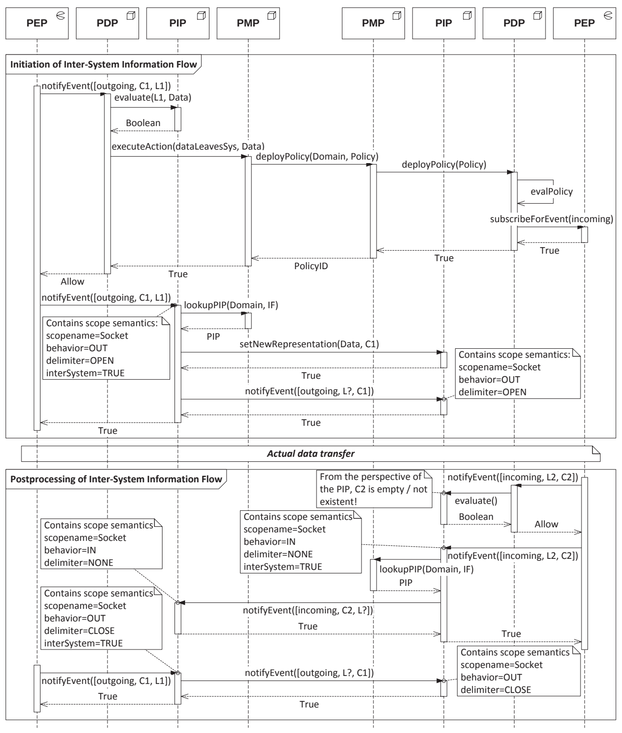
PEP
PDP
PIP
PMP
PMP
PIP
PDP
PEP
Initiation of Inter-System Information Flow
notifyEvent([outgoing, C1, L1])
evaluate(L1, Data)
Boolean
executeAction(dataLeavesSys, Data)
deployPolicy(Domain, Policy)
deployPolicy(Policy)
evalPolicy
subscribeForEvent(incoming)
True
True
PolicyID
True
Allow
notifyEvent([outgoing, C1, L1])
lookupPIP(Domain, IF)
Contains scope semantics: scopename=Socket behavior=OUT delimiter=OPEN interSystem=TRUE
PIP
setNewRepresentation(Data, C1)
True
Contains scope semantics: scopename=Socket behavior=OUT delimiter=OPEN
notifyEvent([outgoing, L?, C1])
True
True
Actual data transfer
Postprocessing of Inter-System Information Flow
From the perspective of the PIP, C2 is empty / not existent!
notifyEvent([incoming, L2, C2])
evaluate()
Boolean
Allow
Contains scope semantics scopename=Socket behavior=IN delimiter=NONE
Contains scope semantics scopename=Socket behavior=IN delimiter=NONE interSystem=TRUE
notifyEvent([incoming, L2, C2])
lookupPIP(Domain, IF)
PIP
Contains scope semantics scopename=Socket behavior=OUT delimiter=CLOSE interSystem=TRUE
notifyEvent([incoming, C2, L?])
True
True
Contains scope semantics scopename=Socket behavior=OUT delimiter=CLOSE
notifyEvent([outgoing, C1, L1])
notifyEvent([outgoing, L?, C1])
True
True

Fig. 2. Inter-System Information Flow

## A. Inter-System Information Flow Tracking

Video streaming is triggered by a request from the client, which is intercepted on the server side. The according *outgoing* event triggers the steps 1 to 6 (cf. Listing 1). It indicates

```
<event action="outgoing" timestamp="2015−05−30T09:30:10">
  <parameter name="network" value="192.168.0.2:80;192.168.0.1:49152"> <!−− C1 −−>
  <parameter name="process" value="2a26af9d−f565−4775−87b5−8eb1fb987ad5"> <!−− L1 −−>
  <parameter name="currentscope" value="192.168.0.2:80;192.168.0.1:49152">
</event>
```

Listing 1. Outgoing Event at Server Side

an outgoing flow from the local container *L1*, i.e., the actual server process providing the video stream, to a container *C1* representing the network connection to the client from the perspective of the server.

In step 2, a policy deployed at the server side PDP grants access to the video stream under the condition that a policy for protecting the requested video data is deployed at the client side (step 3): Both policies refer to the video data by means of a unique dataID $data$, which represents the video data within PIPs. Thus, when evaluating the *outgoing* event concerning the local container *L1* against the local policy, the server side PDP queries the local PIP whether *L1* contains $data$ (cf. *evaluate*-call to the server side PIP in Fig. 2). As the PIP returns *true*, the policy matches the outgoing event and evaluates to *allow* under the condition that the policy deployment at the client is successful. This policy demands that no further representations of the protected video data must be created, which includes that it must not be saved and that the screen must not be captured while the video data is accessed.

In step 4 the outgoing event is handled by the server side PIP. The PIP holds a *local semantics* and a *remote semantics* for this event type. The local semantics for the outgoing event is shown in Listing 2. The scope attribute

```
<ifsemantics>
  <params>
    <param name="network" type="CONTAINER"/>
    <param name="process" type="CONTAINER"/>
  </params>
  <actions>
    <action name="outgoing">
      <scope behavior="OUT" delimiter="OPEN" interSystem="TRUE">currentscope</scope>
      <operation name="SF_FLOW">
        <left>
          <operand>network</operand> <!−− C1 −−>
        </left>
        <right>
          <operand>process</operand> <!−− L1 −−>
        </right>
      </operation>
    </action>
    <action name="close">
      <scope behavior="OUT" delimiter="CLOSE" interSystem="TRUE">currentscope</scope>
      <operation name="SF_CLEAR">
        <left>
          <operand>network</operand> <!−− C1 −−>
        </left>
        <right>
        </right>
      </operation>
    </action>
  </actions>
</ifsemantics>
```

Listing 2. Local Semantics of Outgoing and Close Event

$interSystem = \text{TRUE}$ in the *outgoing* action description is equivalent to $inter = \text{XSYSTEM}$ in the formal model and activates an XSYSTEM scope. The action description indicates a flow from the local container *L1* into the network container *C1*. Due to $behavior = \text{OUT}$ of the scope, *C1* is substituted by the scope's *intermediate container* at the server side PIP: As the PIP knows that *L1* contains $data$, it models this flow by mapping $data$ to the intermediate container.

The scope specification in the semantics also triggers the server side PIP to signal the upcoming data transfer to the client side PIP. So far, the client side PIP neither knows that this data exists nor that the client requested it. Step 5 takes care of the first part: The server side PIP creates a new representation of the data at the client side PIP, i.e., we add an initial mapping between the dataID $data$ of the video data and the remote network container *C1* to the client side information flow model. The server side PIP then forwards the event to the client side PIP. In case the remote semantics for this event has not yet been deployed at the client side PIP, it is attached to this notification (cf. Listing 3).

```
<ifsemantics>
  <params>
    <param name="network" type="CONTAINER"/>
    <param name="process" type="CONTAINER"/>
  </params>
  <actions>
    <action name="outgoing">
      <scope behavior="OUT" delimiter="OPEN">currentscope</scope>
      <operation name="SF_FLOW">
        <left>
          <operand>process</operand> <!−−L?−−>
        </left>
        <right>
          <operand>network</operand> <!−−C1−−>
        </right>
      </operation>
    </action>
    <action name="close">
      <scope behavior="OUT" delimiter="CLOSE">currentscope</scope>
      <operation name="SF_CLEAR">
        <left>
          <operand>network</operand> <!−− C1 −−>
        </left>
        <right>
        </right>
      </operation>
    </action>
  </actions>
</ifsemantics>
```

Listing 3. Remote Semantics of Outgoing and Close Event

In step 6, the client side PIP processes the outgoing event from the server side given the remote semantics (cf. Sec. IV-B). Due to $delimiter = \text{OPEN}$ the client side PIP also creates a new scope. The semantics indicates an information flow from the network container *C1* into the container *L?*, which is a wildcard for an unknown container that receives the flow at the client (the local container at the server included in the event is ignored at the client). According to $behavior = \text{OUT}$ of the scope, the client PIP replaces *L?* with the scope's *intermediate container* (cf. Sec. IV-C). Together with the fact that *C1* contains $data$, we obtain a flow of $data$ from *C1* into the *intermediate container* at the client side. After this step, the server starts sending video data to the client.

Steps 7 to 9 are triggered by an *incoming* event intercepted by the client side PEP when receiving data over the network connection with the server (cf. Listing 4). The *incoming* event refers to the same *scope* as the *outgoing* event. It indicates a flow from a network container *C2* representing the network connection from the perspective of the client side PEP into a local container *L2*, i.e., the process accessing the video stream. The client side PDP evaluates this event against the policy that has been deployed in step 3. This requires the PDP to query the PIP whether this flow involves a representation of the protected video data (step 8, cf. *evaluate*-call to the client side PIP in Fig. 2). As *C2* is either empty, i.e., it has been created during a prior connection to the server, or does not yet exist, the PIP returns *false*, and the PDP will *allow* the incoming event.

```
<event action="incoming" timestamp="2015−05−30T09:30:11">
  <parameter name="process" value="16820cec−18c7−49a2−a443−cd94f0fec3e0"> <!−−L2−−>
  <parameter name="network" value="192.168.0.1:49152;192.168.0.2:80"/> <!−− C2 −−>
  <parameter name="currentscope" value="192.168.0.2:80;192.168.0.1:49152">
</event>
```

Listing 4. Incoming Event at Client Side

In step 9, the *incoming* event is processed at the client side PIP, which holds a local semantics for this event type. The semantics is shown in Listing 5. It contains a scope specification with $behavior = \text{IN}$ and $delimiter = \text{NONE}$. It further signals a flow from the network container *C2* into the local container *L2*. The $delimiter = \text{NONE}$ indicates that the event belongs to an already activated inter-system

```
<ifsemantics>
  <params>
    <param name="process" type="CONTAINER"/>
    <param name="network" type="CONTAINER"/>
    <param name="process_id" type="CONTAINER_NAME"/>
  </params>
  <actions>
    <action name="incoming">
      <scope behavior="IN" delimiter="NONE">currentscope</scope>
      <operation name="SF_FLOW">
        <left>
          <operand>process</operand> <!−−L2−−>
        </left>
        <right>
          <operand>network</operand> <!−−C2−−>
        </right>
      </operation>
      <operation name="NF_ADD_NAMING">
        <left>
          <operand>process_id</operand>
        </left>
        <right>
          <operand>process</operand> <!−−L2−−>
        </right>
      </operation>
    </action>
  </actions>
</ifsemantics>
```

Listing 5. Local Semantics of Incoming Event

scope. Due to $behavior = \text{IN}$, *C2* is replaced by the scope's *intermediate container* within the client side PIP. Together with the state after steps 5 and 6, the client side PIP observes a flow of $data$ from the remote container *C1* via the *intermediate container* into *L2*, i.e., as of now, the PIP knows that *L2* contains the video data $data$, which is protected by our policy. Furthermore, a naming is added to the state of the naming function in order to make *L2* accessible via the PID of the process receiving the video data.

Once the client disconnects from the video stream, the established inter-system state is no longer needed, i.e., we deactivate the scopes and delete the intermediate containers at both PIPs. In our example, the termination of the network connection is observed by the server side PEP (step 10). The according event is processed at the server side PIP (step 11) and forwarded to the client side PIP. In line with Algorithm 1, this event is processed with scope delimiter CLOSE at the server and the client according to the scope specification of the local semantics (cf. Listing 2) respectively the remote semantics deployed in step 5 (cf. Listing 3). As *C1* was replaced by the *intermediate container* at the server in step 4, the event has no effect except for closing the scope locally. Tracking of this XSYSTEM flow terminates after the close event is interpreted at the client side (step 12).

## B. Client Side Policy Enforcement

In terms of enforcing our policy to inhibit the redistribution of video data at the client (iii), the PIP is queried each time a user triggers an event indicating an according information flow, e.g., when trying to take a screenshot. The event of taking a screen shot is intercepted by a PEP on the client (Android platform, cf. [7] for further details) and is only allowed if no application in the foreground has access to the video stream protected by our policy. For this, the PIP can be queried using the PIDs of questionable processes (cf. V-A, step 9). For the PID of the application accessing the video stream, the PIP will answer that this container is a representation of the data, for which our policy applies. Accordingly, the event, i.e., the screen shot, is inhibited.

The reliability of distributed UC enforcement and likewise the obtained level of security is based on the following assumptions: The integrity and the correctness of policies and components of the UC infrastructure is ensured. The infrastructure is up and running and not tampered with, i.e., users do not have administrative privileges on their devices.

## VI. Related Work

The subject of this paper is specification and processing of information flow semantics depending on events that are intercepted by UC monitors – including inter-system and inter-layer information flows.

Park and Sandhu [8] introduced the first UC model UCON, which has not been combined with information flow tracking. The distributed usage control (DUC) model proposed by Pretschner et al. in [1] has been extended with information flow tracking in [2], [5] in order to enable the enforcement of policies depending on the state of an information flow model. The aspect of distributed enforcement of UC policies is considered in greater detail in [9], [10], also focusing on efficient PDP-PIP communication.

Our work builds on and extends [2]–[5]. We unify information flow semantics specifications of monitoring components and generalize the information flow model to cope with inter-system flows. Since we were up to a lightweight proof-of-concept implementation we did not yet consider monitoring technology with higher precision such as the following. Lovat et al. [4], [11] proposed approaches to handle implicit flows [12] and to address the issue of over-approximations of such simple taint-based information flow tracking systems, which we do not cover.

Information flows towards operating system resources and in-between processes are addressed by taint-based information flow tracking frameworks such as Panorama [13] and TaintDroid [14]. SeeC [15] also covers inter-system taint propagation. With Neon [16], Zhang et al. provide a virtual machine monitor for tainting and tracking flows on the level of bytes, which does not require the modification of applications and operating systems. Demsky's tool GARM [17] tackles data provenance tracking and policy enforcement across applications and systems via application rewriting.

## VII. Conclusion

We described and implemented a generic, extensible, and application-oriented approach for dynamic information flow modeling and processing of explicit flows, also across the boundaries of systems equipped with usage control technology. By this means we can enforce usage control requirements on representations of protected data items on remote systems after the initial access to the data has been granted. In our proof-of-concept implementation we have shown how video footage from a surveillance system can be protected against duplication and redistribution even if it is accessed by a mobile application as being employed more and more frequently for cooperation between control rooms and security personnel on-site (provided that the mobile device is equipped with UC technology, otherwise it would not be granted access in the first place).

Our generic primitives for specifying information flow semantics enable engineers to develop information flow monitors (PEPs), which can easily be plugged into existing usage control infrastructures, and thus facilitates the deployment of information flow tracking technology in evolving scenarios. By means of eliminating the interdependency between event capturing and information flow tracking at development time, the practical application of state-based usage control enforcement based on information flow tracking is improved.

## References

[1] A. Pretschner, M. Hilty, and D. A. Basin, "Distributed usage control," *Commun. ACM*, vol. 49, no. 9, pp. 39–44, 2006. [Online]. Available: http://doi.acm.org/10.1145/1151053

[2] M. Harvan and A. Pretschner, "State-based usage control enforcement with data flow tracking using system call interposition," in *Proc. NSS*, 2009, pp. 373–380. [Online]. Available: http://dx.doi.org/10.1109/NSS.2009.51

[3] F. Kelbert and A. Pretschner, "Data usage control enforcement in distributed systems," in *Proc. CODASPY*, 2013, pp. 71–82.

[4] E. Lovat, "Cross-layer data-centric usage control," Dissertation, Technische Universität München, München, Germany, 2015.

[5] A. Pretschner, E. Lovat, and M. Büchler, "Representation-independent data usage control," in *Proc. DPM*, 2011, pp. 122–140.

[6] T. Wüchner and A. Pretschner, "Data loss prevention based on data-driven usage control," in *Proc. ISSRE (IEEE)*, 2012, pp. 151–160. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2012.10

[7] D. Feth and A. Pretschner, "Flexible data-driven security for android," in *Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on*. IEEE, 2012, pp. 41–50.

[8] J. Park and R. S. Sandhu, "The ucon$_{abc}$ usage control model," *ACM Trans. Inf. Syst. Secur.*, vol. 7, no. 1, pp. 128–174, 2004. [Online]. Available: http://doi.acm.org/10.1145/984334.984339

[9] D. A. Basin, M. Harvan, F. Klaedtke, and E. Zalinescu, "Monitoring data usage in distributed systems," *IEEE Trans. Software Eng.*, vol. 39, no. 10, pp. 1403–1426, 2013. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TSE.2013.18

[10] F. Kelbert and A. Pretschner, "Decentralized distributed data usage control," in *Proc. CANS*, 2014, pp. 353–369.

[11] E. Lovat and F. Kelbert, "Structure matters - A new approach for data flow tracking," in *Proc. SPW (IEEE)*, 2014, pp. 39–43. [Online]. Available: http://dx.doi.org/10.1109/SPW.2014.15

[12] E. Lovat, J. Oudinet, and A. Pretschner, "On quantitative dynamic data flow tracking," in *Proc. CODASPY*, 2014, pp. 211–222. [Online]. Available: http://doi.acm.org/10.1145/2557547.2557551

[13] H. Yin, D. X. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proc. CCS (ACM)*, 2007, pp. 116–127. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315261

[14] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, p. 5, 2014. [Online]. Available: http://doi.acm.org/10.1145/2619091

[15] H. C. Kim, A. D. Keromytis, M. Covington, and R. Sahita, "Capturing information flow with concatenated dynamic taint analysis," in *Proc. ARES*, 2009, pp. 355–362. [Online]. Available: http://dx.doi.org/10.1109/ARES.2009.56

[16] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage, "Neon: system support for derived data management," in *Proc. VEE*, 2010, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1735997.1736008

[17] B. Demsky, "Cross-application data provenance and policy enforcement," *ACM Trans. Inf. Syst. Secur.*, vol. 14, no. 1, p. 6, 2011. [Online]. Available: http://doi.acm.org/10.1145/1952982.1952988