# Usage Control Enforcement with Data Flow Tracking for X11

A. Pretschner*,[1] M. Büchler+, M. Harvan+
C. Schaefer†, T. Walter†

* *Fraunhofer IESE and TU Kaiserslautern, Germany*
*alexander.pretschner@iese.fraunhofer.de*
+ *ETH Zurich, Switzerland*
{*buechlermatthias@student, mharvan@inf*}*.ethz.ch*
† *DOCOMO Euro-Labs, Munich, Germany*
{*schaefer, walter*}*@docomolab-euro.com*

**Abstract**

Usage control requirements specify restrictions and compulsory actions that relate to the future handling of data. The enforcement of such requirements can and must happen at different levels of abstraction. This is because propositions in policies such as "delete," "copy," "play," etc. can be assigned different semantics, depending on the level of abstraction that is chosen. We provide a formal model for data flow tracking at one such level, namely that of the X11 system, implement monitors that check different copy&paste policies, and use these monitors to ensure that, for instance, screenshots of a window cannot be taken if it contains sensitive data.

*Keywords:* usage control, access control, enforcement, X11, data flow, information flow.

## 1 Introduction

Usage control [18,11] generalizes access control to the future handling of data. Requirements include both restrictions ("movie may not be played more than once," "do not delete within five years," "do not copy") and duties ("notify owner whenever data is accessed", "delete after thirty days") and hence encompass the domains of data protection, compliance, the management of intellectual property and data in distributed (possibly service-oriented) contexts, and also digital rights management.

Requirements are specified in so-called policies. High-level policies like those sketched above immediately raise the question about the semantics of events such as deletion or copying. Deletion can refer to removing a FAT entry, to overwriting the space occupied on a hard disk twenty times, to also deleting all possible copies, including archived copies, etc. Copying can refer to literally copying a file, copying

[1] Work partially supported by the FhG Internal Programs under Grant No. Attract 692166 as well as by the EU, project FP7-IST-IP-MASTER.

the content from a file in a text editor and pasting it into another file, displaying the content and taking a screenshot, etc. In order to enforce policies, it is necessary to define the semantics at different levels of abstraction, including the operating system, runtime system, infrastructure applications such as X11 or enterprise service buses, applications such as data bases or word processors, and business processes. It is subsequently also necessary to enforce policies at different levels of abstraction, and to connect the different levels whenever data flows from one level (a file) to another (a process) and yet another one (a window on a screen).

**Problem**

We tackle the problem of enforcing usage control policies at the X11 level. X11 is a distributed system for rendering windows. In the X11 context, data marked as sensitive is propagated to many different containers, including network packets, windows, properties of these windows such as their names, etc. To enforce such policies, we must track the flow of data through the X11 system. Moreover, we must take care of "copying" in different varieties, including marking text with the mouse and taking screenshots.

**Solution**

We start by instantiating a formal data flow model to the concepts of the X11 system. We then use this model to implement a monitor that tracks data flows and that makes it possible to prohibit and modify specific actions. For instance, if a data item is not supposed to be copied, then taking a screenshot from a window that contains that data will result in a black rectangle.

**Contribution**

We are not aware of any systems that have implemented data flow tracking at the X11 level; nor do we know of systems that implement copy&paste policies at this level (for a discussion of related work see §5). More importantly, we see our contribution in the machine-level instantiation of earlier more conceptual work on usage control and information flow tracking.

**Structure**

We introduce the X11 system and a general model for data flow tracking in §2. After instantiating the model to the X11 context in §3, we describe our implementation and evaluate it in §4. We put our work in context in §5, and conclude in §6.

# 2 Background

## *2.1 X11*

The X Window system is a distributed system and a protocol for building a graphical user interface environment on Unix-like systems. It provides a basic framework for drawing and moving windows on a screen as well as for communicating with devices attached to the system, including mouse and keyboard. The X system is designed to

be used over network connections. Consequently, it consists of clients and servers. Servers manage and render windows on one particular machine. Servers also manage the communication between different clients which, in turn, access the information that is managed by the server. Client applications use the X server to provide a graphical user interface and communicate with each other via the X server. The XWindows client/server terminology is slightly confusing because each machine that wants to render windows runs its own X11 server: there is not one server and many clients, as one might expect, but many servers and many clients, and the client can run on the same or a different machine as the server. Clients use the *Xlib* library for communicating with the server, either directly or by other widget toolkits libraries that use *Xlib* which offers a high-level API for using the X11 protocol.

Using the X11 protocol, interaction between clients and server takes place over sockets that we will refer to as *X11 connections*. Communication consists of requests, replies, events and errors. *Requests* are sent by a client to the server to request the execution of an action by the server. In response to some requests the server sends back information in a *reply*. *Events* are sent asynchronously by the server to inform clients about relevant state changes in the server. The protocol allows clients to selectively register for specific events, such as those related to a particular window. An *error* is sent in response to an invalid request.

The X server maintains resources used as basic elements in the interaction with a client. As we will later see, these resources are the containers that potentially carry sensitive information. Containers include windows (the whole screen area can be split into several movable, potentially overlapping subareas, called windows), pixmaps (memory areas in the X server that are valid destinations for a majority of the drawing functions), atoms (unique names that clients may use for accessing resources or for communication between different clients), attributes and properties (variables attached to windows), cut buffers (special named, pre-defined properties used for communication between clients), fonts (structures for representing text), etc.

**Copy&Paste**

Copy&paste is supported by X11 in two ways: selections and cut buffers. Roughly speaking, *selection* of data works as follows: During the "copy" phase, a client obtains the selection (a token) by requesting it from the server. In response, the previous selection owner looses the selection. During the "paste"-phase, a client sends a request that specifies a property where the selected content should be stored, together with the format into which the selected content should be converted. The X server forwards the request to the selection owner. The selection owner converts the selected content to the specified format, copies it to the property specified in the request, and sends back a notification event. The selection owner has to be available and maintain the selected data until copied to the receiving application's property. Otherwise, the selected data is lost. An alternative implementation relies on an additional application, the clipboard manager, which acts as a middle man and keeps a copy of the selected content even if the original application ceases to exist.

In contrast to selections, *cut buffers* act as intermediate containers. Similarly to

the clipboard manager case, availability of the data to be copied does not rely on the availability of the application holding the original data.

**Screenshots**

A second way of copying data is to take a screenshot. This refers to copying the graphical representation of the contents of the whole screen or a part of it, as rendered by the X server. To take a screenshot, a client essentially sends a request to the X server, specifying the area for the screenshot. The server returns the graphical data.

### *2.2 Data Flow*

We have introduced usage control as relating to the handling of data after giving it away. *Handling* relates to the rendering, processing, distribution, management, and execution of data [20]. The definition of *data* is slightly less evident because the same data is usually represented in many different forms. The requirement for a data item sent by email not to be printed relates to many different representations: among others, this data comes as network packets, files, memory areas of an email client, and window content managed by a window manager. The requirement most likely relates to all these representations (which precisely is the main motivation behind our approach to enforce usage control requirements at different levels of abstraction). This leads to the notion of *containers* that store a data item. Usage control policies for a data item will then relate to all containers that (potentially) contain the item.

We now introduce a data flow model that boils down to a transition system capturing the flow of data through a system. State transitions are initiated by X11 messages. A state of the model captures (a) which data is in which container, (b) if there are alias relations between containers—which happens, among other things, whenever two windows overlap, and (c) under which names containers can currently be accessed. The general idea is the following. We monitor X11 messages and update the state of the data flow model accordingly. Enforcement mechanisms for usage control policies then refer to the information stored in the data model and grant or deny the execution of specific actions, including copy&paste of sensitive text or taking screenshots of sensitive data in X11 windows. The data flow model hence is used as an enhancement to an execution monitor at the X11 level.

The generic data flow model, which we will instantiate to the X11 system in §3, is a tuple

$$(D, C, F, \Sigma, I, P, A, R). \tag{1}$$

$D$ is the set of data items whose usage is restricted by a policy. $C$ is the set of data containers in the system and $F$ the set of constructs that are used to identify data containers. Containers include windows, properties, messages, etc. Principals, $P \subset C$, are those containers that store data and can invoke actions with other containers, e.g., applications.

$$\Sigma = (C \to 2^D) \times (C \to 2^C) \times (F \to C) \tag{2}$$

is the set of states. States consist of three mappings. The *storage function*, $s$,

captures which data is stored in which containers. The *alias function*, $l$, captures the fact that some containers may implicitly get updated whenever other containers do. Intuitively, if $c_2 \in l(c_1)$ for $c_1, c_2 \in C$, then whenever something is written into $c_1$, it is immediately propagated into $c_2$. For example, when windows are moved in XWindows, the background of the moved window may be replaced by that of a window that is overlapped—which we will model as an alias in §3. To treat chains of aliases we will also use the reflexive transitive closure of the alias function, denoted $l^*$. Finally, the *naming function*, $f$, maps identifiers to containers. $I \in \Sigma$ is the initial state of the system. The storage function of the initial state is given by the usage policy. That is, we assume that the usage policy specifies which restrictions apply to which data, and where that data is initially stored. $A$, actions, initiate state changes, X11 messages in our context, described by a (deterministic) relation

$$R \subseteq \Sigma \times P \times A \times \Sigma. \tag{3}$$

Since states are modeled as triples of functions, we need some additional notation for specifying state changes. For a mapping $m : S \to T$ and a variable $x$ ranging over $X \subseteq S$, define $m[x \leftarrow expr]_{x \in X} = m'$ with $m' : S \to T$ and $m'(y) = expr$ if $y \in X$ and $m'(y) = m(y)$ otherwise. Multiple updates for disjoint sets are combined by function composition $\circ$. We will use the semicolon as syntactic sugar: $m[x_1 \leftarrow expr_{x_1}; \ldots; x_n \leftarrow expr_{x_n}]_{x_1 \in X_1, \ldots, x_n \in X_n} = m[x_n \leftarrow expr_{x_n}]_{x_n \in X_n} \circ \ldots \circ m[x_1 \leftarrow expr_{x_1}]_{x_1 \in X_1}$. All replacements are done simultaneously and atomically.

# 3 Usage Control and Data Flow for X11

## *3.1 Instantiating the Information Flow Model*

### *3.1.1 Principals*

Processes are natural candidates to act as principals. Because ours is a distributed setting, however, we cannot use process IDs but rather identify principals by IP address and port (that is, one application can possibly be represented by several principals): $P = IPAddress \times Port$ where we assume the carrier sets *IPAddress* and *Port* to be given, and the same port may not be directly shared between different processes without the help of a port management application.

### *3.1.2 Data Containers*

Essentially, the X11 resources sketched in §2 define the set $C$ of X11 data containers. To quote just a few examples, the X11 resources *Attributes* and *Properties* can be perceived as variables in programming languages and are therefore classified as data containers. *Windows* are used for drawing graphic elements like pictures, text, buttons, etc. *Pixmaps* are blocks of off-screen memory in the X server and consist of an array of pixel values. *X11 Connections* are used by principals to send data to the X server. These connections consist of two connected end points (ports). One port of such a connection is considered as a data container in the X11 submodel, namely the port where the X server reads the request from the client applications. The other port of the connection is considered as a data container at the application's side, usually a socket in the Xlib (and this is treated by another instantiation of

our generic data flow model). To send data to the X server, a principal (a process with IP address and port) writes data into one port, and the X server reads it from the other port of the used connection. Therefore, we consider X11 connections, $X11C \subset C$, as data containers. Using the approach of identifying X11 connections as data containers, network packets are considered as data elements that are stored in the ports of a connection. Finally, we consider the (overapproximated) memory content of each principal $p \in P$, denoted by $m_p$.

There are many other containers that we do not have the space to discuss here, including atoms, graphics contexts, bitmaps, colormaps, cursors, and fonts [4]. The set of containers is $C :=$ Windows $\cup$ Pixmaps $\cup$ Colormaps $\cup$ Cursors $\cup$ Fonts $\cup$ Graphics Contexts $\cup$Atoms$\cup$Properties $\cup$Attributes $\cup$X11C$\cup\{m_p : p \in P\}$ where we assume the sets of identifiers on the right hand side to be given.

### *3.1.3 Identifiers*

Resources (window, pixmap, colormap, cursor, font, graphics context) are X server resources and are identifiable by a single ID, whereas properties and attributes are bound to particular windows. We assume a set $F$ of identifiers to be given that caters to both situations.

### *3.1.4 Actions and State Transitions*

Actions are X11 messages. To define the transition relation $R$, we now describe how some example actions affect the state. For reasons of space, we will restrict ourselves to three exemplary actions (see [4] for a more complete treatment). The initial state consists of three empty mappings.

**Swap**

The X server provides a function to swap or rotate the content of several properties with one function call. The content of two data containers is swapped without knowing their content (i.e., pointers are re-directed). In the formula we express that by not directly updating $m_p$. After swapping, the new content of the two swapped properties does not contain information from the previous content anymore. Therefore, for a storage function $s$ (the first component of the state), container $c_1$ is only updated with $s(c_2)$, but not with $s(c_1)$. If a client application has read one of the involved properties prior to the swapping, the new property content is not propagated to the client. However, the request to swap properties itself can convey information. The client that initiated the swapping potentially sends information to all the clients that know at least the content of one participating property. Such a client can re-read one of the containers and compare the content to what it assumes to be the content. Therefore the participating properties potentially contain all data of the initiating client. This is specified by adding $s(m_p)$. We have

$$\begin{aligned} &\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [F \to C], \forall p \in P, \forall c_1, c_2 \in \mathit{Properties} : \\ &( (s, l, f), p, \mathit{Swap}(c_1, c_2), \\ &\quad (s[u \leftarrow s(c_2) \cup s(m_p); v \leftarrow s(c_1) \cup s(m_p)]_{u \in l^*(c1), v \in l^*(c2)}, l, f)) \in R. \end{aligned} \tag{4}$$

**GetImage**

Screenshots are taken by sending a `GetImage` request. Although the screenshot is taken from window $c_2$, the content of a window $c_1$ that overlaps with window $c_2$ is also included in the screenshot. Assuming that $overlap(c_1, c_2)$ is true if window $c_1$ overlaps with $c_2$—information that can be queried from the X server—the state of the system is changed in the following way: The principal that takes a screenshot from window $c$ gains information from all containers that overlap $c$. We express that by iterating over all containers $t'$ that satisfy $overlap(t', c)$ and add $s(t')$ to the updating of $m_p$:

$$\begin{aligned}&\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [F \to C], \forall p \in P, \forall c \in \mathit{Windows}: \\ &((s, l, f), p, \mathit{GetImage}(c), (s[m_p \leftarrow s(m_p) \cup s(t)]_{t \in \{t' | overlap(t', c)\}}, l, f)) \in R.\end{aligned} \tag{5}$$

**MoveWindow (MvW)**

Windows may be re-positioned on the screen. For the sake of our presentation and due to the fact there is no simple request to re-position a window, we hence introduce an abstract *MvW* request that only reflects the re-positioning aspect of the more complex `ConfigureWindow` request. Define $overlap'$ like $overlap$ with the additional requirement that the background of $c_2$ is referenced by $c_1$. Moving a window can then lead to an information flow from another window because its background might be used for the moving window. The intuition is that every window $t$ that is at least partially covered by a window $c$ refers to that window $c$. We model that by adding $c$ to the alias mapping of the window $t$. Similarly $c$ is removed from the alias mapping of a window $v$ as soon as $v$ is not covered by window $c$ anymore. We get

$$\begin{aligned}&\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [F \to C], \forall p \in P, \forall c \in \mathit{Windows}: \\ &(\ (s, l, f), p, \mathit{MvW}(c), (s, \\ &\quad l[t \leftarrow l(t) \cup \{c\}; v \leftarrow l(v) \backslash \{c\}]_{t \in \{i | overlap'(c,i)\}, v \in \mathit{Windows} \backslash \{i | overlap'(c,i)\}}, f)) \in R.\end{aligned} \tag{6}$$

The above actions all modify the $s$ and $l$ mappings but none of them modifies the $f$ mapping. Due to space limitations we omit an action that also modifies the $f$ mapping. One example is the `InternAtom` request that is used to register an atom-string pair at the X server.

### *3.2 Usage Control Policies*

We have implemented a monitor that allows to intercept and possibly block, add, and modify X11 messages. The decision on whether or not such an action takes place also depends on where sensitive data has flown: we need to know that sensitive information is displayed in a window in order to prohibit screenshots only when necessary. To do so, we implemented the data flow model and update its state by tracking the data flow induced by the X11 messages. We consider it convenient to configure—rather than re-compile—an X11 monitor for usage control by means of policies. Such policies specify rules that consist of conditions and actions. For simplicity's sake, we assume rules to be non-conflicting in this paper. Conditions can

relate to the mere presence of particular messages (e.g., forbid taking screenshots altogether) and also to a complex state. This complex state may contain information about tracked data flows (e.g., forbid taking screenshots from windows with sensitive data). At present, we do not specify the initial pieces of data that have to be tracked—and that are hence considered sensitive—within the policies. Instead, the human user uses a helper application that allows him to click on a data container on the screen (e.g. a textfield in a window). This click tells the system that the data stored in this data container is sensitive and has, from now on, to be traced.

**Conditions**

Conditions are based on requests, events, or replies. A condition specifies the kind of request, event, or reply and specific values of the action that have to be matched. This kind of specification is static and not appropriate in all situations. Conditions can therefore be extended to also access a more complex state of the system than just the matching of certain values in request, event, or reply packets. Examples of criteria that are currently implemented include the following. Will data be stored in a cut buffer? Is the classification (§4) of the data stored in the intercepted network packet higher than a given threshold value (the implementation of the monitor maintains a mapping from containers to classifications that is bootstrapped by explicitly clicking on sensitive elements, as discussed above)? Is the classification of the data to be copied higher than the classification of the window where the data should be pasted to?

**Actions**

The action part is executed if the respective condition holds. Network packets, i.e., the X11 messages, can be deleted or modified. Additional network packets can also be inserted. The content of these network packets can be specified in a static and/or dynamic way. The new values of the network packet (content) can either be hard-coded as a constant in the security policy file or be computed from the content of the current network packet that matches the condition.

**Example**

For space reasons, we do not provide schema definitions here but rather resort to an explanation by example. Consider the sample policy in Listing 1 restricting taking a screenshot of information with a classification level of at least 2. For simplicity we assume that information may only flow using the *copyArea* request. This sample policy requires two low-level policies. The first policy (line 4-13) is needed to intercept each *GetImage* request. It states that taking a screenshot is denied (a black retctangle is returned) whenever information is contained in the screenshot that has a classification level of at least 2 (line 6). A screenshot is denied by modifying the attribute $planeMask$ to 0 (line 10). In order to let the monitor know the classification of the data in each container, and therefore also the classification of the screenshot, a second policy (line 1-3) is needed. It states that flow and classification information of every *copyArea* request has to be tracked. This information is tracked if the corresponding `registerFlow` and `registerClassification` directives are set to 1 (line 2). The first policy does not specify additional actions because

for tracking this information neither a *copyArea* request has to be modified, nor do we have to add or delete X11 messages.

```
<Policy>
  <Condition type="CopyArea" registerFlow="1" registerClassification="1"></Condition>
</Policy>
<Policy>
  <Condition type="GetImage" registerFlow="1" registerClassification="1">
    <thresholdValue>2</thresholdValue>
  </Condition>
  <Actions>
    <Modify type="GetImage">
      <planeMask>0</planeMask>
    </Modify>
  </Actions>
</Policy>
```

Listing 1: Screenshot Policy

## 4 Implementation

Clients can be executed on local and remote machines. Remote machines are not necessarily under the same control as the X server. An application connects to the X server by opening a TCP or DECNet connection or a Unix socket. The monitor can be implemented both at the client and the server side of the X11 connections. Implementing it at the client side implies that the Xlib library has to be wrapped or monitored. This approach is convenient when an application is dynamically linked to Xlib and gets more difficult in case of static linking. However, our approach relies on a modification of the Xmon tool (`ftp://ftp.x.org/contrib/devel_tools/`) that we use to wrap the X server rather than all clients. Xmon was conceived for debugging X11 applications at the level of the X core protocol; it essentially allows to monitor X11 network packets. Because our monitor is implemented as a wrapper application it provides the following facilities. Incoming packets from both client and server can be intercepted, i.e., stored in a queue within the X11 monitor. Before such a network packet leaves the queue, the content of the packet can be inspected, analyzed, and modified. A special case of modifying a network packet is to delete it by modifying it into a `NoOperation` request. Insertion of messages is possible by adding elements to the queue.

Several mechanisms can be used to copy data between applications (§2). The idea behind our implementation is to use the fact that these data flow mechanisms boil down to sequences of requests and events at the level of the X protocol (an alternative consists of considering states rather than sequences of messages [9]). To track data flow within the X window system, we maintain a classification mapping, $\mathit{Classification} : C \rightarrow \mathbb{N}_0$, in addition to the three mappings that constitute the state of the data flow model. This mapping assigns classification levels (natural numbers) to data containers. The X11 monitor updates the state if it intercepts a request or event and if the policy specifies that the data flow and classification information for that request or event has to be tracked. It extracts all necessary information from the action (i.e., references to data containers) and uses this information to update the mappings. Monitoring every single data flow of course results in a huge amount of data. Therefore as part of the configuration of the X11 monitor one may specify for each request or event whether or not information flow or classification information should be monitored (§3.2).

### *4.1 Example Application 1*

The first application denies taking a screenshot if the respective window contains sensitive information. Several windows may be visible in a screenshot, and one window usually consists of several subwindows. We assume that a screenshot contains sensitive information if one of the windows visible in the screenshot has a classification of at least $x$, where we assume that a classification of $x$ is more sensitive than a classification of $x-1$. The classification of a window is computed as the maximum of all its subwindow classifications. If the X11 monitor intercepts a `GetImage` request, it checks if a window with classification of at least $x$ is contained in a screenshot (information of another window may be contained in the screenshot because that window may overlap). If so, the X11 monitor modifies the `planeMask` parameter of the corresponding `GetImage` request so that the X server returns a black rectangle which contains no (valuable) information, except for the size of the window.

As an example, assume three different distinct windows w1,w2, and w3 on the screen. Data can be exchanged between them by sending `CopyArea` requests. In the initial state of this scenario, data stored in window w1 is sensitive, whereas data in w2 and w3 is non-sensitive. In our implementation, this is made explicit by a helper application that allows to classify windows by clicking on them. The policy specifies that a screenshot must not be taken from a window that stores sensitive data. Therefore, in the initial state, a screenshot is denied from w1, but is allowed from w2 and w3. In the next step, the data of w1 is copied to w2, and as a last step, from w2 to w3, via a `CopyArea` request. After this information flow from w1 to w3 over w2 a screenshot must not be taken from any of the three windows because they all store sensitive data now. Our X11 monitor implements this policy by intercepting the `CopyArea` request. It extracts information about the source and destination containers and uses the classification of the source container to possibly update that of the destination container. In addition it intercepts the `GetImage` request and possibly modifies the `planeMask` parameter as discussed above, resulting in a black rectangle.

### *4.2 Example Application 2*

The second example application, which we consider fully independently of the first one, is an enforcement mechanism for copy&paste. Consider an application A that displays sensitive data in its window. Let this sensitive data currently be selected and ready to be pasted to another application by pressing the middle mouse button. Our first policy is as follows. Pasting is denied if the destination application is not monitored by the X11 monitor (because e.g. it directly connects to the X server). This allows to monitor only the participating rather than all applications. If an application is monitored, pasting is allowed if the application already displays sensitive data as well. This prevents sensitive text from being merged with non-sensitive text. The X11 monitor enforces this policy by intercepting `SelectionRequest` events which are forwarded by the X server to application A. The monitor checks if the requesting application is monitored, and whether the window where the data is to be pasted also displays sensitive data. If not, the monitor sends a negative `SelectionNotify` event to the requesting application; it otherwise forwards the

`SelectionRequest` event to application A.

We now consider a Chinese Wall policy, again relying on three windows w1, w2, and w3. Each window can communicate with either of the two remaining windows, but not with both of them. As an example, if w1 first communicates with w2, this communication is allowed. If w1 subsequently starts a communication with w3 as well, this is denied. We again assume that communication in this scenario is implemented by the `CopyArea` request of the X protocol. The X11 monitor extracts information about the source and the destination drawable of the request. Furthermore it stores information about the destination drawable in the source drawable and vice versa. If the X11 monitor recognizes for a source data container that it wants to communicate with one of the two remaining windows but has already communicated with the respective other one, the `CopyArea` request is blocked. In case the source data container has at most communicated with the destination window of the current request so far, the request passes unmodifed.

### *4.3 Performance Overhead*

To measure the monitor's impact we perform several copy-actions in OpenOffice so that data is copied from OpenOffice to a (distinct) clipboard application. As a simple experiment, we execute a macro in OpenOffice that (1) goes to the beginning of the current line, (2) inserts the current time, expressed as seconds and microseconds since 1970/01/01, (3) selects the inserted text, (4) copies it to the external clipboard manager and (5) inserts a new paragraph. After waiting 1s, the macro repeats tasks 1 to 5 fifty times to get a more accurate average value. At the same time we modified the xclipboard application (contained in `https://launchpad.net/ubuntu/hardy/+source/x11-apps/7.3+1/`) so that it compares the times before and after receiving and displaying the copied value.

To measure the overhead, the macro is executed as described above with and without monitoring. In the monitored case, every X11 message has to pass through the (wrapper) monitor that, by virtue of the corresponding policy, intercepts all `SetSelectionOwner, ConvertSelection`, and `ChangeProperty` requests and `SelectionRequest` events. For each intercepted request or event the X11 monitor checks if the condition specified in the policy file evaluates to true. To finally allow the copy action between OpenOffice and the clipboard manager, the policy is configured in such a way that none of the conditions evalute to true and each intercepted network packet is forwarded as usual. The X11 monitor also tracks the classification of the current selection if it intercepts a `SetSelectionOwner` request.

Copy actions take an average of 15ms without and 402 ms with monitoring. The overhead is (402-15)/15=2580%. To put this number in context, monitoring at the Java code level incurs an overhead of up to 680000% [1] even without information flow tracking, at the system call level of up to 270% [9], at the Java bytecode level of up to 7000% [17]. The concrete overhead is context-specific because it is determined, among others, by the number of intercepted requests and events, whether the X11 monitor communicates with other components of the system, and the additional actions performed upon a matching condition. Moreover, the `ChangeProperty` request is not only used for copying but for other actions as well, e.g., write any kind of property.

## 5 Related Work

Usage control has been discussed by several authors [18,3,2,10]. Compared to commonly used information flow models like [6,16] our model is explicitly tailored to dynamic information flow analysis and explicitly includes the notion of aliases. [9] discusses how usage control policies can be cast in terms of information flow policies.

Information flow frameworks have been implemented at various levels of abstraction, including x86 CPU instructions [5], Java [8,17], operating systems [7], etc. These approached do not consider the more general problem of enforcing usage control requirements. An overview of usage control enforcement mechanisms, without considering information flow, is presented in [20]. The Adobe Reader (`http://www.adobe.com/de/products/reader`) can disallow printing or copying via copy-paste the contents of a pdf document based on rights specified within the document. Using the Windows RMS (`http://www.microsoft.com/windowsserver2003/technologies/rightsmgmt/default.mspx`) system, printing and copy-pasting can also be disallowed for newer Microsoft Office products. Compared to our approach, both solutions modify the application that interprets the data while we insert a monitor that is transparent to existing applications.

Security monitors that also act as enforcement mechanisms have been discussed by Schneider et al. [22]. The enforced information flow-related policies are restricted to explicit information flow [6], hence are properties of a trace and EM-enforceable. The kinds of enforcement mechanisms used in this paper – modification, deletion and insertion of x11 network packets – implement the idea of edit automata [14] and have been independently discussed elsewhere [20,19].

[12] describes three approaches to prevent capturing sensitive information on a screen: 1) modifying the API of the windowing system on a Windows system 2) modifying the API and display driver to ensure that content is exchanged in encrypted form between the application and the display driver; and 3) modifying the API and the graphics hardware to ensure that the content is exchanged in encrypted form between the application and the hardware. When compared to our work, these approaches have to modify existing applications as well as parts of the system and hardware. [13] discusses securing the X system with SELinux by placing the X server into a trusted domain and controlling critical X11 objects with SELinux mechanisms. In contrast to our approach, this requires modifying the X server and using a special window manager. XCB [15] is a replacement library for Xlib to secure the X system. When compared to our approch, all application have to be modified to use the new library rather than just have their communication with the X server pass through a monitor.

## 6 Conclusions

Usage control policies relate to data with many different representations. To enforce them, it is necessary to keep track of all those containers that potentially contain the data item. This motivates the introduction of our abstract data flow model.

Because high-level (natural-language) usage control requirements can be interpreted at different levels of abstraction, enforcement necessarily has to be imple-

mented at these different levels. Motivated by different semantics of "copying" a data item, we have instantiated the abstract data flow model to the XWindows system. Our implementation transparently runs on top of an X11 server which hence needs not be modified. The presented work is one step towards our goal of usage control enforcement within and across all the layers of a computer system. As example applications, we have shown how to prohibit doing copy&paste as well as screenshots of sensitive data, and have provided some evidence that the performance overhead is about 2580% in first basic experiments.

Our work has several limitations. To start with, we do not discuss the flow of data *in-between* different levels of abstraction, e.g., from the operating system or a Java virtual machine to an application. Moreover, our model is limited to explicit information flow. Implicit flows are not covered, mainly because the notion of a conditional is not entirely clear at the level of X11. Secondly, our implementation currently intercepts only a subset of all X11 requests and events. In a later step this implementation will be extended to cater to other X window mechanisms as well, e.g., Drag and Drop. Thirdly, the basis for our considerations is the X11 protocol. At the same time the clients usally make use of the xlib library which is not monitored in our approach; this would be another level to be monitored. Monitoring xlib calls is relevant because if, for instance, a human user uses an input device (keyboard, mouse, etc.) this potentially leads to state modifications within the xlib part of the application. Only some of these state modifications trigger a request or event that is observable at the X protocol level. Therefore for a more fine-grained approach, it is important to monitor the xlib part of the application as well. This has an important consequence in terms of distinguishing copy&paste functionality *within* one [21] and *between* different applications, the latter being the subject of this paper. A further related limitation is that the X core protocol can be extended to provide new actions to the clients, e.g., Graphical Tool Kits. An X11 monitor has to handle such extensions as well. Fourthly, because we define the semantics of copy&paste as a sequence of events, it is of course possible that we have missed specific sequences that also allow to do copy&paste (for a state-based rather than event-based approach, see [9]). Fifthly, the problem of how to make sure that the enforcement system actually runs on a system is outside the scope of this paper; we consider trusted computing technology to be a promising candidate in solving this problem. Finally, the instantiation of the general data flow model to the X11 and other levels leads to an overapproximation of the mapping from containers to data: it represents potential, not factual, containment relations. We expect this phenomenon—that is also witnessed by the interest in declassification strategies in many approaches in the information flow community—to become relevant when we start connecting different levels of abstraction.

We are aware that full usage control enforcement is an ambitious goal, that there always is a chance that mechanisms are bypassed, and that we will always have to face the problem of media breaks, e.g., when a screen is photographed. However, we believe that in contrast to DRM scenarios, there are many application scenarios where 100% enforcement is not absolutely necessary, e.g., in semi-trusted contexts with document management within one company. We are currently working on implementations at different levels of abstraction, including the level of word pro-

cessors, the Java runtime system, and the operating system. We are also working on connecting these levels. One short-term goal, for instance, is to connect the X11, operating system [9], and word processor [21] levels for seamless enforcement of copying requirements within one document, between several documents, and between the word processor and other applications [4].

# References


[1] Artzi, S., S. Kim and M. D. Ernst, *Recrash: Making software failures reproducible by preserving object states*, in: *Proc. ECOOP*, 2008, pp. 542–565.

[2] Bertino, E., C. Bettini and P. Samarati, *A temporal authorization model*, in: *Proc. CCS*, 1994, pp. 126–135.

[3] Bettini, C., S. Jajodia, X. S. Wang and D. Wijesekera, *Provisions and obligations in policy rule management*, J. Network and System Mgmt. **11** (2003), pp. 351–372.

[4] Büchler, M., *Usage Control Enforcement at the X11 level* (2009), MS Thesis, ETH Zurich.

[5] Clause, J., W. Li and A. Orso, *Dytan: a generic dynamic taint analysis framework*, in: *Proc. ISSTA*, 2007, pp. 196–206.

[6] Denning, D. E., *A lattice model of secure information flow*, CACM **19** (1976), pp. 236–243.

[7] Efstathopoulos, P., M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek and R. Morris, *Labels and event processes in the asbestos operating system*, in: *Proc. SOSP*, 2005, pp. 17–30.

[8] Haldar, V., D. Chandra and M. Franz, *Dynamic taint propagation for java*, in: *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference* (2005), pp. 303–311.

[9] Harvan, M. and A. Pretschner, *State-based usage control enforcement with data flow tracking using system call interposition*, in: *Proc. Network and System Security*, 2009, to appear.

[10] Hilty, M., D. Basin and A. Pretschner, *On obligations*, in: *Proc. ESORICS*, 2005, pp. 98–117.

[11] Hilty, M., A. Pretschner, D. Basin, C. Schaefer and T. Walter, *A policy language for distributed usage control*, in: *Proc. ESORICS*, 2007, pp. 531–546.

[12] Hussain, K., N. Addulla, S. Rajan and G. Moussa, *Preventing the capture of sensitive information*, in: *ACM-SE 43: Proceedings of the 43rd annual Southeast regional conference* (2005), pp. 154–159.

[13] Kilpatrick, D., W. Salamon and C. Vance, *Securing the x window system with selinux*, http://www.nsa.gov/research/_files/selinux/papers/x11/t1.shtml (2003).

[14] Ligatti, J., L. Bauer and D. Walker, *Edit automata: Enforcement mechanisms for run-time security policies*, International Journal of Information Security **4** (2005), pp. 2–16.

[15] Massey, B. and J. Sharp, *Xcb: an x protocol c binding*, in: *ALS '01: Proceedings of the 5th annual Linux Showcase & Conference* (2001), pp. 24–24.

[16] Myers, A. C. and B. Liskov, *A decentralized model for information flow control*, in: *Proc. SOSP*, 1997, pp. 129–142.

[17] Nair, S., P. Simpson, B. Crispo and A. Tanenbaum, *Trishul: A Policy Enforcement Architecture for Java Virtual Machines*, Technical Report IR-CS-045, Department of Computer Science, Vrije Universiteit Amsterdam (2008).

[18] Park, J. and R. Sandhu, *The ucon abc usage control model*, ACM Transactions on Information and Systems Security (2004), pp. 128–174.

[19] Pretschner, A., M. Hilty, D. Basin, C. Schaefer and T. Walter, *Mechanisms for Usage Control*, in: *Proc. ACM Symposium on Information, Computer & Communication Security (ASIACCS)*, 2008, pp. 240–245.

[20] Pretschner, A., M. Hilty, F. Schutz, C. Schaefer and T. Walter, *Usage control enforcement: Present and future*, Security & Privacy, IEEE **6** (2008), pp. 44–53.

[21] Schaefer, C., T. Walter, A. Pretschner and M. Harvan, *Usage Control Policy Enforcement in OpenOffice.org and Information Flow*, in: *Proc. 8th Annual Information Security South Africa Conference*, 2009, to appear.

[22] Schneider, F. B., *Enforceable security policies*, ACM Trans. Inf. Syst. Secur. **3** (2000), pp. 30–50.