State-based Usage Control Enforcement with Data Flow Tracking using System Call Interposition

Matúš Harvan
Information Security
ETH Zurich
Zurich, Switzerland
email: mharvan@inf.ethz.ch

Alexander Pretschner
Fraunhofer IESE and TU Kaiserslautern
Kaiserslautern, Germany
email: Alexander.Pretschner@iese.fraunhofer.de

Abstract

Usage control generalizes access control to what happens to data in the future. We contribute to the enforcement of usage control requirements at the level of system calls by also taking into account data flow: Restrictions on the dissemination of data, for instance, as stipulated by data protection regulations, of course relate not to just one file containing the data, but likely to all copies of that file as well. In order to enforce the dissemination restrictions on all copies of the sensitive data item, we introduce a data flow model that tracks how the content of a file flows through the system (files, network sockets, main memory). By using this model, the existence of potential copies of the data is reflected in the state of the data flow model. This allows us to enforce the dissemination restrictions by relating to the state rather than all sequences of events that possibly yield copies. Generalizing this idea, we describe how usage control policies can be expressed in a related state-based manner. Finally, we present an implementation of the data flow model and state-based policy enforcement as well as first encouraging performance measurements.

1. Introduction

Usage control is an extension of access control [1], [2]. In addition to specifying and enforcing who may access given data, one also specifies and enforces restrictions on its usage, such as what may, may not or has to be done with it.

In general, usage control requirements can be enforced at different levels of granularity, including the levels of CPU, operating system, runtime system, infrastructure applications such as window managers or the X11 system, user applications and services, middleware such as enterprise service buses, database systems, or at the process workflow level. Enforcement can be performed both intrusively, i.e., by modifying and re-compiling the source code, and transparently, by adding technology to a running system. Both variants give rise to monitors that observe actions and either react in case of a policy violation, or prevent the policy violation from happening. Different enforcement mechanisms not only differ in which technology is deployed but also in terms of the guarantees that can be given. For instance, if a usage control mechanism can block deletion requests for a service, this does not mean that a user cannot bypass this interface of the system and directly delete the file.

Our work focuses on the level of system calls. System calls are the interface between user space processes and the operating system kernel. They allow processes to use resources provided by the operating system, such as communication with other processes, accessing files and using the network and various devices. System call interposition refers to monitoring system calls, possibly changing them or their return values, and denying their execution.

Inititally, a policy with dissemination or deletion requirements—or any other usage control requirements—is attached to a data item that may be contained in a file. Whenever copies of the data item are created, the policy must be copied as well, attached to these fresh data items, and also enforced for these copied items. Moreover, deletion or dissemination can happen in many ways: a file can be copied by using a dedicated "cp f1 f2" command, its content can be read and written into another file by executing "cat f1 $>$ f2," a process can read every single byte of one file and write it to another file, etc. Expressing all possible permutations and interleavings of events that lead to data copies, and then expressing policies on all these event sequences, seems like a rather ambitious goal. Our approach hence is to rather reflect the existence of potential copies by means of abstract states, and formulate usage control policies in terms of the state rather than event sequences.

Problem statement. In this paper we address the questions of what usage control policies can be monitored with system call interposition and how an enforcement framework can be realized. To this end, we study two main problems.

1) A usage control policy typically restricts the usage of data, independent of the *representation* of the data.

This work was supported by a research collaboration with DOCOMO EuroLabs. The second author was also supported by the FhG internal programs under grant no. Attract 692166 and by the EU project MASTER.

However, in a system we can only observe processes and specific representations of the data. The monitoring of such a policy hence requires knowing into which representations (files, main memory, etc.—in the following referred to as *containers*) within the system the data was propagated. In other words, which copies do exist?

2) Usage control policies mention high level actions including *print*, *distribute*, and *delete*. On the system call level, only calls such as `read`, `write` and `unlink` can be observed. These correspond to rather low level actions. Therefore, a connection between the high level actions and system calls has to be established.

Question 1 is addressed by means of a data flow model associating processes and containers with data. Question 2 is addressed by providing the semantics of selected high-level actions in terms of states in our data flow model.

Contributions. The contributions of our work are:

1) We provide a formal and operational data flow model for system calls that allows us to overapproximate the existence of copies of a data item.
2) We show how policies can be described in a state-based manner by leveraging the data flow model. In this way, we can express policies that cannot be conveniently captured in an event-based way.
3) We show how high-level policies can be enforced in terms of low-level system calls.
4) We provide proof-of-concept implementation and do a preliminary performance analysis.

Limitations. We have focused on system calls available in the OpenBSD 4.4 system and not requiring superuser privileges. Furthermore, our analysis of system calls excludes implicit flows and covert channels. These are mechanisms not explicitly designed for communication that may be used to surreptitiously communicate information between processes. For example file locking or modulation of shared resources use can be used for covert communication [3]. Our approach is also limited to safety properties.

Structure. The remainder of this paper is organized as follows. The data flow model that we use to track the existence of potential copies is described in Section 2. Section 3 shows how usage control policies can be expressed in terms of states of the data flow model, and subsequently enforced on the level of system calls. An evaluation on the basis of a proof-of-concept implementation is presented in Section 4. Related work is discussed in Section 5 before the paper concludes in Section 6.

2. Data Flow Model

In this section we describe the data flow model. As discussed in Section 3, this will allow us to conveniently express several usage control policies in a state-based manner. For brevity's sake, the described model leaves out some aspects of system calls. The model is limited to explicit information flow. Implicit flows are not covered. The simplifications and their consequences are summarized in Section 4.

In a nutshell, the data flow model is a transition system that captures the flow of data through a system. State transitions are initiated by system calls. A state captures (1) which data is in which container, (2) if there are alias relations between containers—which, among other things, can happen if processes share memory—and (3) under which names containers can currently be accessed. More precisely and formally, the model is a tuple $(D, C, F, \Sigma, I, P, A, R)$.

D is the set of data whose usage is restricted by a usage policy. C is the set of all possible data containers in the system. Containers are possible locations for data. They include files, pipes, message queues and the network. We do not distinguish between various network connections but treat the whole network as a single container c_{net}. Principals, $P \subset C$, are the set of all possible processes in the system. When compared to other containers, principals can invoke actions which other containers, e.g., files, cannot. Processes are containers because the process state, CPU registers and the memory image of the process are possible locations for data. F is the set of container identifiers. It consists of the disjoint sets of file names ($F_{fn} \subseteq F$), descriptors and sockets ($F_{dsc} \subseteq F$), and a special *nil* value for inactive and closed descriptors not pointing to any containers.

$\Sigma = (C \to 2^D) \times (C \to 2^C) \times (P \times F \to C)$ is the set of all possible states. States consist of three mappings:

1) A *storage function* s capturing which data is stored in which container.
2) An *alias function* l capturing 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. Aliases are needed to correctly model memory-mapped file access and shared memory. To treat chains of aliases we will also use the reflexive transitive closure of the alias function, denoted l^*.
3) A *naming function* f mapping identifiers to containers. Since some identifiers are process-specific, e.g., file descriptors, the naming function maps pairs of process and container 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. The alias and naming functions are system specific. A, actions, are system calls. We limit the model to system calls available on an OpenBSD 4.4 system that do not require superuser privileges and that do affect the data flow state.

Actions A, performed by processes, change the state of the system. These changes are described by a (deterministic)

relation $R \subseteq \Sigma \times P \times A \times \Sigma$. R is the smallest relation satisfying equations 1 to 16 described below.

Since states are modeled as triples of functions, we need some additional notation for specifying state changes. For any 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

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

Multiple updates for disjoint sets can be combined by function composition $\circ$. We will use the semicolon as syntactic sugar:

$$m[x_1 \leftarrow expr_{x_1};\ \ldots;\ x_n \leftarrow expr_{x_n}]_{x_1 \in X_1, \ldots, x_n \in X_n} = m[x_n \leftarrow expr_{x_n}]_{x_n \in X_n} \circ \ldots \circ m[x_1 \leftarrow expr_{x_1}]_{x_1 \in X_1}.$$

The semantics is that all replacements are done simultaneously and atomically. Later on, we will also use simultaneous function updates, i.e., not only simultaneous updates at different points of the domain, but for different functions. We will use the same notation however and implicitly assume that if two function updates occur within one parenthesis, then they are executed simultaneously.

We will denote the return value of a system call with *rv*. Moreover, we assume that invoked system calls succeed, and therefore we do not include in our model conditions on the return value or error codes relevant to the succeeding or failing of a system call.

We now describe the effects of the various system calls as requirements on the relation R. In particular, we model the handling of descriptors, reading from and writing into containers, renaming and deleting containers, process management, inter-process communication and the creation and removal of aliases.

Descriptors. Containers are usually not accessed directly but rather via descriptors. For example, reading from a file requires opening a descriptor to the file, reading via the descriptor and closing the descriptor afterwards. Descriptors are specific to a single process, rather than known system-wide. In our model, descriptors are elements of the set F and are mapped to containers by function f. Descriptors can be opened, duplicated and closed. Opening a file descriptor is accomplished with an *open* system call as formalized in eq. (1).

$$\begin{aligned} &\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [P \times F \to C], \\ &\forall p \in P, \forall n \in F_{fn}, \forall rv \in F_{dsc} : \\ &\quad \big((s,l,f), p, open(n, rv), \\ &\quad\ (s,l,f[(p,rv) \leftarrow f(p,n)])\big) \in R. \end{aligned} \tag{1}$$

The intuition here is that a successful *open(n,rv)* system call takes place: process p opens a file with name n, and the operating system returns a descriptor *rv*. Consequently, the (naming component of the) state is modified by adding a pointer from (p, rv) to the container that is named by $f(p,n)$: the process-specific descriptor *rv* now points to that container $f(p,n)$.

Creating a pipe c_p with the corresponding descriptors is accomplished with a *pipe* system call (eq. (2)).

$$\begin{aligned} &\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [P \times F \to C], \\ &\forall p \in P, \forall c_p \in C, \forall e_1, e_2 \in F_{dsc} : \\ &\quad \big((s,l,f), p, pipe(e_1, e_2, c_p), \\ &\quad\ (s,l,f[(p,e_1) \leftarrow c_p;\ (p,e_2) \leftarrow c_p])\big) \in R. \end{aligned} \tag{2}$$

Similar to file access, network communication takes place via a network socket - a descriptor representing the network connection. As explained earlier, we do not distinguish between different communication partners and treat the whole network as the single container c_{net}. The descriptor is created with the *socket* or *accept* system calls (eq. (3)). For brevity we only show *socket*; *accept* is analogous.

$$\begin{aligned} &\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [P \times F \to C], \\ &\forall p \in P, \forall rv \in F_{dsc} : \\ &\quad \big((s,l,f), p, socket(rv), \\ &\quad\ (s,l,f[(p,rv) \leftarrow c_{net}])\big) \in R. \end{aligned} \tag{3}$$

Descriptors can be duplicated within a process with the *dup*, *dup2* and *fcntl* with the `F_DUPFD` flag (eq. (4)). We do not model the sending of descriptors among processes via unix sockets, and we omit the definition of *dup2* and *fcntl*.

$$\begin{aligned} &\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [P \times F \to C], \\ &\forall p \in P, \forall e_1, e_2 \in F_{dsc}, \forall rv \in F_{dsc} : \\ &\quad \big((s,l,f), p, dup(e_1, rv), (s,l,f[(p,rv) \leftarrow f(p,e_1)])\big) \in R \end{aligned} \tag{4}$$

A descriptor can explicitly be closed with the *close* system call (eq. (5)). A closed descriptor can no longer be used for accessing the container.

$$\begin{aligned} &\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [P \times F \to C], \\ &\forall p \in P, \forall e \in F_{dsc} : \\ &\quad \big((s,l,f), p, close(e), (s,l,f[(p,e) \leftarrow nil])\big) \in R. \end{aligned} \tag{5}$$

Reading and Writing. Once a descriptor has been opened, data can be read via the descriptor from the corresponding container into a process by invoking one of the following system calls: *read*, *readv*, *pread*, *preadv*, *recv*, *recvfrom*, *recvmsg*. For brevity, we show the respective definition of relation R only for *read* (eq. (6)) as it is analogous for the other system calls.

$$\begin{aligned} &\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [P \times F \to C], \\ &\forall p \in P, \forall e \in F_{dsc} : \\ &\big((s,l,f), p, read(e), (s[t \leftarrow s(t) \cup s(f(p,e))]_{t \in l^*(p)}, l, f)\big) \in R. \end{aligned} \tag{6}$$

$f(p,e)$ yields the container for the descriptor e. Since our model allows for aliases, reading data from the container into process p immediately propagates the data to other containers aliased by p (recall that processes are containers). These are computed via $l^*(p)$. This is the reason for parametrizing the update of the storage function.

Similarly, writing from a process into a container via an open descriptor can be achieved with one of the following

system calls: *write*, *writev*, *pwrite*, *pwritev*, *send*, *sendto*, *sendmsg* supplying the descriptor identifier as a parameter to the system call. For brevity eq. (7) shows only *write*. The relation is analogous for the other system calls.

$$\begin{aligned}&\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [P \times F \to C],\\ &\forall p \in P, \forall e \in F_{dsc} :\\ &\quad \big((s,l,f), p, write(e),\\ &\quad\ (s[t \leftarrow s(t) \cup s(p)]_{t \in l^*(f(p,e))}, l, f)\big) \in R.\end{aligned} \tag{7}$$

As with reading, the update of s is parametrized over the transitive closure of aliases to the destination container.

Renaming Containers. Files can be renamed (eq. (8)). Let the file with name n_1 be renamed to n_2. Then $f(p,n_1) = c_1, f(p,n_2) = c_2$ for $\forall p \in P$ before the renaming. After the renaming $f(p,n_2) = c_1$ and the contents of c_2 can no longer be accessed. However, a new file $f(p,n_1) = c_3$ could be created with *open*. Since we do not add or remove containers to the set C, we reuse c_2 for c_3. To ensure there is no connection between c_2 before and after the rename, we discard the contents of $c2$, i.e., set $s(c2) = \varnothing$. Although the data may still be in blocks on the disc, directly accessing them would require privileged system calls which are excluded from our model.

$$\begin{aligned}&\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [P \times F \to C],\\ &\forall p \in P, \forall n_1, n_2 \in F_{fn} :\\ &\big((s,l,f), p, rename(n_1, n_2), (s[f(p,n_2) \leftarrow \varnothing], l,\\ &\ f[(q,n_2) \leftarrow f(q,n_1);\ (q,n_1) \leftarrow f(q,n_2)]_{q \in P})\big) \in R.\end{aligned} \tag{8}$$

Note that the renaming makes the container accessible to all processes under the new name and hence the naming function f has to be updated for all processes. Having $f(q,n_1)$ point to $f(q,n_2)$ prevents a dangling pointer; the respective container is emptied, as mentioned above. The fact alone that the file was renamed can convey information. However, we consider this a covert channel and hence do not treat it in our model.

Deleting Containers. Some containers, files, can explicitly be deleted using the *unlink* system call (eq. (9)). Upon deletion, the contents of the container are lost and all alias relations involving the container are removed. The data stored in aliased containers is not affected. Note that we do not add or remove containers from the system. As with renaming, the data of the deleted file may remain on the disc, but accessing it requires privileged system calls. These are not treated in our model.

$$\begin{aligned}&\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [P \times F \to C],\\ &\forall p \in P, \forall n \in F_{fn} :\\ &\quad \big((s,l,f), p, unlink(n), (s[f(p,n) \leftarrow \varnothing], l[f(p,n) \leftarrow \varnothing;\\ &\qquad\qquad t \leftarrow l(t) \setminus \{f(p,n)\}]_{t \in C}, f)\big) \in R.\end{aligned} \tag{9}$$

Process Management. Processes can duplicate themselves, replace their memory image with a file to execute, and cease to exit.

Duplication takes place by using the fork system call (eq. (10)). The duplicate, or child process, will be in the same state as the parent, i.e., contain the same data, have the same alias relations and the same descriptors as the parent. Since memory-mapping and shared memory attaching are the only mechanisms capable of creating aliases we can simply copy all alias relations involving the parent process.

$$\begin{aligned}&\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [P \times F \to C],\\ &\forall p, rv \in P :\\ &\quad \big((s,l,f), p, fork(rv), (s[rv \leftarrow s(p)], l[rv \leftarrow l(p);\\ &\quad\ t \leftarrow l(t) \cup \{rv\}]_{t \in \{t' | p \in l(t')\}}, f[(rv,e) \leftarrow f(p,e)]_{e \in F_{dsc}})\big) \in R.\end{aligned} \tag{10}$$

Using the *execve* system call (eq. (11)) a process can replace its memory image with the contents of a file. As a simplification we assume that alias relations involving the process are removed, but descriptors are not modified.

$$\begin{aligned}&\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [P \times F \to C],\\ &\forall p \in P, \forall n \in F_{fn} :\\ &\quad \big((s,l,f), p, execve(n), (s[p \leftarrow s(f(p,n))], l, f)\big) \in R.\end{aligned} \tag{11}$$

After executing the *exit* system call (eq. (12)) a process ceases to exist. The memory image of the process is discarded, alias relations are removed and open descriptors are closed.

$$\begin{aligned}&\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [P \times F \to C],\\ &\forall p \in P :\\ &\big((s,l,f), p, exit, (s[p \leftarrow \varnothing], l[p \leftarrow \varnothing;\ t \leftarrow l(t) \setminus \{p\}]_{t \in C},\\ &\qquad\qquad f[(p,e) \leftarrow nil]_{e \in F_{dsc}})\big) \in R.\end{aligned} \tag{12}$$

Inter-Process Communication. Processes can communicate with each other directly or indirectly via containers. The direct way is by sending signals with the *kill* system call, using the *ptrace* debugging interface. The indirect way involves reading (eq. (6)) and writing (eq. (7)) into containers or setting up aliases. Aliases are discussed in the next section. Note that for brevity some IPC-specific containers like message queues or semaphores are omitted from our model.

The semantics of *kill* is that the process invoking the *kill* system call communicates data to another process (eq. (13)). Hence the formula is similar to the formula that models writing (eq. (6)).

$$\begin{aligned}&\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [P \times F \to C],\\ &\forall p, q \in P :\\ &\big((s,l,f), p, kill(q), (s[t \leftarrow s(t) \cup s(p)]_{t \in l^*(q)}, l, f)\big) \in R.\end{aligned} \tag{13}$$

For brevity we omit details of the ptrace interface. With a read flag it behaves analogically to eq. (13) while with a write flag p and q would be swapped in eq. (13).

Aliases. Aliases can be created in two ways, by memory-mapping containers and by attaching a shared memory region to the memory image of a process. We will first discuss memory-mapping.

Containers can be mapped to memory unidirectionally (read) or bidirectionally (read/write). In the unidirectional case (eq. (14)), a process p mapping the container c to its memory can read from c but not write into c. This corresponds to an alias relation $(\mathrm{c},\mathrm{p}) \in l$. The uni-directional case also constitutes a reading from the file, so s has to be updated as in eq. (6).

In the bidirectional case (eq. (15)) the process can both read and write. Hence, aliases for both directions have to be added. This also constitutes reading and writing, so s has to be updated as in eq. (6) and eq. (7).

$$\begin{aligned}&\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [P \times F \to C],\\ &\forall p \in P, \forall e \in F_{dsc} :\\ &\big(\quad (s,l,f), p, mmap(e, \texttt{PROT\_READ}), (\\ &\quad s[t \leftarrow s(t) \cup s(f(p,e))]_{t \in l^*(p)},\\ &\quad l[f(p,e) \leftarrow l(f(p,e)) \cup \{p\}], f)\big) \in R.\end{aligned} \tag{14}$$

$$\begin{aligned}&\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [P \times F \to C],\\ &\forall p \in P, \forall e \in F_{dsc} :\\ &\big(\quad (s,l,f), p, mmap(e), (\\ &\quad s[t \leftarrow s(t) \cup s(f(p,e));\ t' \leftarrow s(t') \cup s(p)]_{t \in l^*(p), t' \in l^*(f(p,e))},\\ &\quad l[p \leftarrow l(p) \cup \{f(p,e)\};\ f(p,e) \leftarrow l(f(p,e)) \cup \{p\}],\\ &\quad f)\big) \in R.\end{aligned} \tag{15}$$

The memory mapping-based alias is removed with the *munmap* system call (eq. (16)).

$$\begin{aligned}&\forall s \in [C \to 2^D], \forall l \in [C \to 2^C], \forall f \in [P \times F \to C],\\ &\forall p \in P, \forall c \in C :\\ &\quad \big((s,l,f), p, munmap(c),\\ &\quad (s, l[p \leftarrow l(p) \setminus \{c\};\ c \leftarrow l(c) \setminus \{p\}], f)\big) \in R.\end{aligned} \tag{16}$$

Shared memory is similar to memory-mapping containers, but the containers are dedicated memory regions. For brevity we omit a detailed discussion.

Our model does not explicitly include the possibility of multiple instances of the same alias, i.e., memory mapping the same file several times. This could be handled by attaching identifiers to aliases (process ID and memory address of the mapping), but is omitted in the model.

3. State-based Policies

The data flow model allows us to describe policies in a state-based way. This allows us 1) to conveniently express some policies that are hard to express by explicitly listing sequences of events, and 2) to map high-level policies to low-level policies at the level of system calls. In the following, we will use the terms event-based and state-based policies. Both abstractions are equally expressive as histories of events can be encoded in the system state. However, when it comes to keeping track of copies, and *having policies that implicitly also apply to all these copies* we will argue that the latter are more convenient for describing usage control policies. As a motivating example for state-based policies consider an event-based policy that is supposed to restrict the dissemination of data: the medical record of a particular patient may not be disseminated over the network. One can rather easily describe the sequence of system calls for disseminating a particular container's content (the file storing the medical record) over the network. Capturing the fact that the data may have been propagated in various ways into other containers and disseminated from these, however, quickly becomes inconvenient if not infeasible. This is because it is no challenge to construct arbitrarily long sequences of system calls by simply adding intermediate files via which data is copied before finally sending it from the last file in the chain.

In contrast, the dissemination policy can conveniently be described in a state-based way: any state in which the data item (potentially) is in the c_{net} container is not allowed. The expression of usage control requirements at the level of *states* rather than *events* does not seem to be mainstream as indicated by the relevant literature (note that our notion is different from state-based policies as found, for instance, in ODRL—here the *policy* maintains a state). Many usage control-relevant policies, however, can be described in a state-based way using our data flow model. These policies seem to be out of reach of an approach that is based on explicitly listing sequences of events. For example,

1) *Data d must not be distributed over the network.* – The state of the system shall always satisfy $d \notin s(c_{net})$.
2) *Data d may not be copied.* – This is hard to express in event-based policies because there are many sequences of events that potentially lead to copies of a data item. Let c_d be the only container where d is initially stored. Then the state of the system shall always satisfy $\forall c' \in C \setminus P : d \in s(c') \Rightarrow c_d = c'$.
3) *Data d must be deleted at time t.* – This is hard to express in event-based policies because the deletion requirement extends to all copies of d rather than just the initial copy of d. Expressed in a state-based way, the state of the system at time t must satisfy $\forall c \in C : d \notin s(c)$. Note that this does not specify how data is removed from containers, only that they must not be present at time t. Also note that if one simply deletes all containers that potentially contain the item, the overapproximation inherent in our model is likely to lead to many unnecessary deletions. Finally, note that a dual retention requirement cannot easily be expressed for the same reason of overapproximation: the fact that a data item d is stored in a container via the s-mapping does not necessarily mean that d is in the container or can be reconstructed from its content.
4) *Data d_1 and d_2 must not be combined.* – This is hard

to express in event-based policies because again many different sequences of events lead to "combinations." We may say, however, that the state of the system shall never satisfy $\exists c \in C : \{d_1, d_2\} \subseteq s(c)$.

Special purpose actions like playing a song using the audio device or printing can also be expressed using a state-based policy. Similarly to the dissemination over the network with c_{net}, we can use additional containers. As an example, for the CUPS printing system the sockets of the cups server could be defined as c_{print}. For the audio device, `/dev/audio*` and `/dev/sound*` device nodes could be defined as c_{audio}, and the *ioctl* system calls with the corresponding flags added similarly to *write* in eq. (7) and eq. (14). Playing a song would then correspond to the song data appearing in $s(c_{\text{audio}})$.

Although policies tend to be stated in terms of high-level actions, such as disseminate, copy or delete, a monitor typically observes only low-level actions like system calls. By expressing the high-level actions in terms of states in our data flow model we map the high-level policies to low-level actions without explicitly naming these low-level actions. We simply leverage the work described above that was done to set up the R relation for system calls.

In order to conveniently describe policies, we define the following policy atoms:

- *deny_c*(d, C') – Data $d \in D$ may not enter containers in $C' \subseteq C$. Formally, every state must satisfy $\forall c \in C' : d \notin s(c)$.
- *limit_c*(d, C') – Data $d \in D$ may only enter containers in $C' \subseteq C$. Formally, every state must satisfy $\forall c \in C \setminus C' : d \notin s(c)$.
- *limit_c_file*(d, C') – *limit_c* ignoring non-file containers. Formally, the state must satisfy $\forall c \in C : d \in s(c) \Rightarrow (c \in C' \vee c \text{ is not a file})$.
- *deny_comb*(d_1, d_2) – Data $d_1, d_2 \in D$ may not be combined. Formally, every state must satisfy $\nexists c \in C : \{d_1, d_2\} \subseteq s(c)$.

These atoms allow us to describe the following high-level policy examples,

1) *Data d must not be distributed over the network.* – *deny_c*$(d, \{c_{net}\})$
2) *Data d may not be copied.* – *limit_c_file*$(d, \{c_d\})$ where c_d is the container where d is initially stored.
3) *Data d must be deleted.* – *limit_c*$(d, \varnothing)$ at some specific moment in time.
4) *Data* d_1 *and* d_2 *must not be combined.* – *deny_comb*(d_1, d_2)

We have seen that state-based policies are useful whenever a policy is stated for one data item but is actually meant for all copies of the data item as well. Unfortunately, in addition to the overapproximation issues discussed in the context of deletion and retention requirements, not all policies can directly be expressed in a pure data flow state-based way, which suggests of course a combination of state-based and event-based policies that we do not discuss in this paper:

- Counting – Counting the number of times an event occurs, such as in the policy "data d may be used at most 3 times [4]." A possible remedy would be to describe sequences of events in addition to states.
- Duration – Measuring the duration of events or states, such as "data d may be used for a total time of at most 2 hours [4]." However, the usage can be captured as the time while the system state satisfies $\exists p \in P : d \in s(p)$.
- Actions inside of processes - Only interactions of a process with the rest of the operating system can be observed. Peeking into the process to observe internal actions such as anonymization of data is not possible.

Note that the monitored information flow-related policies are restricted to explicit information flow, hence are properties of a trace and EM-enforceable[5].

4. Evaluation and Discussion

To evaluate our approach and measure the imposed overhead, we have implemented a proof-of-concept framework that keeps track of the state as described in Section 2 and enforces the policy atoms described in Section 3. We now discuss the implementation, its performance and the restrictions of our approach.

The basic idea of the implementation is to observe invoked system calls before they are executed; calculate the state resulting from the system call's execution; check if that state does not violate the policy; and if it does, then deny the system call's execution. To calculate the state of the data flow model, the invoked system calls have to be observed. The standard system call interaction consists of 1) a user-space process invoking a system call, 2) the kernel executing the invoked system call, and 3) delivering a return value back to the invoking process. The systrace framework [6] allows us to step in the interaction between 1) and 2) – the invoked system call can be inspected before it is executed by the kernel, its parameters can be modified or its execution can be denied; and also between 2) and 3) – the return value delivered to the invoking process can be observed and modified. Upon inspection of the system call, we compute the state change it would entail. If this state change comes with a policy violation, system call execution is denied, and the value `EPERM` is returned to the invoking process. Otherwise, execution of the system call is permitted. System calls for creating descriptors and forking processes are also inspected between 2) and 3). This is because their return value is needed to correctly calculate the new state. Note that this does not interfere with enforcing the policy atoms described in Section 3. Also note that only a subset of system calls is monitored, executing unmonitored system calls does not incur any overhead.

The monitored policy can be any conjunction of the atoms from Section 3. Such a policy also has to describe the initial distribution of data among the containers, *s*. This distribution is used as the initial state. Enforcement of the policy is limited to denying system calls. Injecting additional system calls, as needed for the enforcement of some policies, such as forcing the deletion of all data copies, is not the subject of this paper.

We evaluated the implementation with unmodified standard unix tools like *cp*, *mv* and *rm*, and the following policy including typical usage control requirements: disallowing the network dissemination of data 1 (/tmp/a) – *deny_c*$(1, \{c_{net}\})$, the copying of data 2 out of /tmp/b – *limit_c_file*$(2, \{\text{file:/tmp/b}\})$, and the combination of data 1 (/tmp/a) with data 3 (/tmp/c) – *deny_comb*$(1, 3)$; initial state $s(\text{file:/tmp/a}) = \{1\}$, $s(\text{file:/tmp/b}) = \{2\}$, $s(\text{file:/tmp/c}) = \{3\}$. In a sample scenario we propagated data 1 with *cp /tmp/a /tmp/m*, *mv /tmp/m /tmp/n*, *cat /tmp/n* $>$ */tmp/o* and reading /tmp/o into the *vi* editor and writing it into /tmp/p. This resulted in /tmp/p containing data 1 – $s(\text{file:/tmp/p}) = \{1\}$. Examples of detected policy violations were uploading /tmp/p to an ftp server using the ftp client *lftp*, *cp /tmp/b /tmp/x* and *cat /tmp/c* $>>$ */tmp/a*. The execution of the offending system calls was denied and the returned EPERM value was correctly interpreted as insufficient permissions.

We measured the overhead of data flow tracking and policy monitoring by comparing the runtimes of untraced programs, of programs subject to data flow tracking with disabled policy monitoring, and of programs subject to data flow tracking and monitoring of the above described policy. To ensure comparable runs with identical system call sequences, we chose programs that did not violate the policy. We have considered the runtime of 1) compiling C++ source code (namely, the implemented framework) as an example of a CPU-intensive task, and as i/o intensive tasks 2) copying a 10KB file using *cp*, 3) copying a 10KB file using *dd*, 4) copying a 10MB file using *cp*, 5) copying a 10MB file using *dd*. *dd* used a block size of 1KB. The runtime was measured on a system with a Pentium M 1.6GHz CPU using the *real* output of the *time* program. The traced programs were run multiple times, 6 for 1), 6000 for 2) and 3), and 30 for 4) and 5). Instead of the average we have considered the aggregate time over the multiple runs. The aggregate runtimes are depicted in Figure 1. For compiling C++ code tracing incurred an overhead of 8% while policy monitoring raised it up to 33%. Copying files incurred less overhead with *cp* than with *dd* as well as with large files instead of small files. For 10KB files, the overhead without policy monitoring was 61% for *cp* and 478% for *dd* with policy monitoring. For copying 10MB files, the overhead with *cp* was rather low at 5% without and 17% with policy monitoring. With *dd*, it was 12% without and 272% with policy monitoring.

The better performance of *cp* and on large files is likely

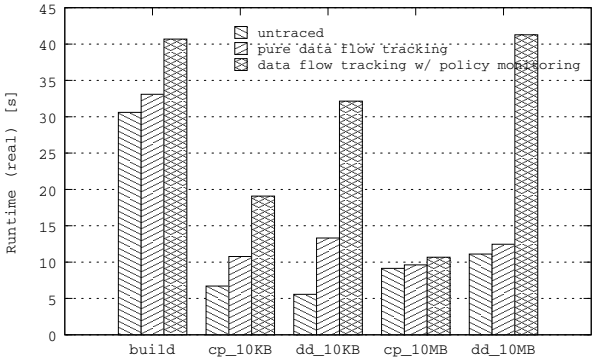

Figure 1. Comparison of runtimes between untraced executions and traced with and without policy monitoring for 6 times compiling the implemented framework, 6000 times copying a 10KB file and 30 times a 10MB file with cp and dd with a block size of 1KB.

due to the fact that *cp* copies small files using *read* and *write* system calls while larger files are copied using the *mmap* mechanism resulting in fewer system call invocations. The larger overhead for policy monitoring is due to the naive proof-of-concept implementation: for each update a copy of the state is stored in case the execution of the system call would be denied. This could be improved by a rollback capability of the state. Although the overhead depends on the programs used, it seems generally acceptable and monitored programs remain usable.

In the following we comment on the limitations of the data flow model. In sum, these are (1) omitted details of the underlying operating system; (2) excluded implicit flows and covert channels; and (3) overapproximation.

(1) The model is limited to the unprivileged system calls in an OpenBSD 4.4 system and some details of system calls have been omitted. Shared memory regions, message queues and semaphores are not included. The data contained in a file is limited to the file content. However, data could also be encoded in the file meta information, such as the file name, permissions or last modified time. The meta information and related system calls are not treated. This issue can be addressed by modelling additional details at the cost of increased complexity of the model. However, the presented selection of details is sufficient to cope with system calls used by standard unix programs like *cp*, *mv*, *rm* and *vi*.

(2) Implicit information flow and covert channels, such as modulation of use of shared resources in order to leak information, have explicitly been excluded from the model. For example, we do not consider the possibility that the mere action of renaming a file could convey information. Treating this issue would result in a significantly more restrictive model where data would be propagated more aggressively or the possible actions of processes would be strongly limited (even with explicit information flow only,

we need to implement declassification techniques). We do not plan to cope with covert channels, but would like to extend our work to implicit information flow in the future. In particular, MAC-policies available in SElinux would be suitable for this purpose.

In terms of issue (3), our model makes an overapproximation showing the potentially worst data flow. Once some data has been written into a file, any subsequent read could potentially read that data. Once a process has that data, every subsequent writing from the process could include that data. This may lead to so-called "label creep" where quickly most data containers are marked as potentially containing almost all data available in the system. We have also observed this phenomenon: *lftp* logged names of transferred files and the history of executed commands into separate files. Both of these log files were marked as potentially containing all the data of the file transmitted by the ftp client. In reality, they only contained the name of the file. Should a policy be monitored, the label creep would lead to false-positive policy violations. The overapproximation problem could be ameliorated by peeking into the processes at a finer level of granularity, such as with an information flow framework at the level of assembler instructions [7].

5. Related Work

We are not aware of other work discussing how usage control policies can be expressed in terms of information flow. Compared to existing information flow models [8], [9] our model additionally caters to aliases. The prototypical operating system Asbestos [10] tracks information flow and enforces policies in the OS kernel. Flume [11] requires applications to use IPC instead of system calls. While our approach works with unmodified binaries, programs have to be modified to run in Asbestos or Flume. [12], [13], [14] use win32 API interposition to monitor data flow and defend against malware, but do not consider more complex usage control policies. The systrace [6] framework neither supports high level policies nor tracks data flow.

6. Conclusion

In this paper, we have studied the suitability of system call interposition for usage control enforcement. To this end, we have set up a data flow model based on system calls, leveraged it to describe policies in a state-based way, and evaluated it in a proof-of-concept implementation. The state-based policy description allows us to conveniently express policies that are hard to capture in an event-based way and to enforce high-level policies at the low-level system calls. The observed overhead on traced processes incurred by our proof-of-concept implementation suggests that our approach is a viable option.

As further work we see combining state-based and event-based policies, extending our work to implicit flows and decreasing the overapproximation of our data flow model by peeking into programs at the level of assembler instructions.

Acknowledgment

The authors would like to thank Thomas Walter and Christian Schaefer from DOCOMO Euro-Labs and Felix Klaedtke from ETH Zurich for comments on an earlier draft of this paper.

References

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

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

[3] B. W. Lampson, "A note on the confinement problem," *Commun. ACM*, vol. 16, no. 10, pp. 613–615, 1973.

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

[5] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, 2000.

[6] N. Provos, "Improving host security with system call policies," in *Proc. SSYM*, 2003, pp. 257–272.

[7] S. McCamant and M. D. Ernst, "Quantitative information flow as network flow capacity," in *Proc. PLDI*, 2008, pp. 193–205.

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

[9] D. E. Denning, "A lattice model of secure information flow," *CACM*, vol. 19, no. 5, pp. 236–243, 1976.

[10] P. Efstathopoulos, 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.

[11] M. N. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard os abstractions," in *SOSP*, 2007, pp. 321–334.

[12] F. Hsu, H. Chen, T. Ristenpart, J. Li, and Z. Su, "Back to the future: A framework for automatic malware removal and system repair," pp. 257–268, 2006.

[13] X. Wang, Z. Li, N. Li, and J. Y. Choi, "Precip: Towards practical and retrofittable confidential information protection," in *NDSS*, 2008.

[14] X. Jiang, A. Walters, D. Xu, E. Spafford, F. Buchholz, and Y.-M. Wang, "Provenance-aware tracing of worm break-in and contaminations: A process coloring approach," in *ICDCS*, 2006, p. 38.