Data Usage Control Enforcement in Distributed Systems

Florian Kelbert
Technische Universität München
Garching bei München, Germany
kelbert@cs.tum.edu

Alexander Pretschner
Technische Universität München
Garching bei München, Germany
pretschner@cs.tum.edu

ABSTRACT

Distributed usage control is concerned with how data may or may not be used in distributed system environments after initial access has been granted. If data flows through a distributed system, there exist multiple copies of the data on different client machines. Usage constraints then have to be enforced for all these clients. We extend a generic model for intra-system data flow tracking—that has been designed and used to track the existence of copies of data on single clients—to the cross-system case. When transferring, i.e., copying, data from one machine to another, our model makes it possible to (1) transfer usage control policies along with the data to the end of local enforcement at the receiving end, and (2) to be aware of the existence of copies of the data in the distributed system. As one example, we concretize "transfer of data" to the Transmission Control Protocol (TCP). Based on this concretized model, we develop a distributed usage control enforcement infrastructure that generically and application-independently extends the scope of usage control enforcement to any system receiving usage-controlled data. We instantiate and implement our work for OpenBSD and evaluate its security and performance.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Information flow controls; D.4.6 [**Security and Protection**]: Access controls

General Terms

Security

Keywords

Distributed Usage Control; Policy Enforcement; Security and Privacy; Sticky Policies; Data Flow Tracking

1. INTRODUCTION

Distributed usage control [26, 30] has been proposed with the goal of overcoming one shortcoming of access control models: loss of control over data once it has been released. Usage control is therefore concerned with what must or must not happen to data *after* access to it has been granted and is particularly interesting in distributed system environments [16, 18]. While traditional access control mechanisms are deployed at the data provider's site, *distributed* usage control requirements must be enforced at the data consumer's site.

Such requirements are expressed in usage control policies by the data provider [11, 17, 34]. Example policies include "only process my data with application X," "do not redistribute my data to company Y," and "delete my data after thirty days." In a distributed setting, these policies must be enforced at *all systems* that store, process, and distribute data. Since data can easily be redistributed in today's internet, it is particularly challenging to make sure that a policy is enforced even after data has been transferred to another system. This paper's subject is the problem of building a usage control infrastructure that ensures that if data is transferred, respective policies are transferred along with the data, and that they will be enforced at the receiving end.

1.1 Motivating Example

Consider a company in which confidential digital business reports are repeatedly handled by several cooperating employees of the finance department for means of creation, modification, approval, and reading. Although the company deployed state-of-the-art security mechanisms such as encrypted and access controlled shared file servers, Public-Key Infrastructures, and secured web services, a data breach happened recently: an employee sent business reports to a server outside the company. While the employees, including Alice and Bob, should be able to do their work as usual, the CEO does not want such incidents to happen ever again.

The CEO thus decides to equip all servers and client machines with a usage control infrastructure and to deploy a policy stating that "business reports may not leave the financial department." Now, once Alice tries to access a business report on the shared file sever, the file server's usage control infrastructure determines whether also Alice has such an infrastructure in place and, if so, allows the access. Our infrastructure will then also transfer the usage policy and enforce it on Alice's machine. Similarly, Alice would then only be able to share the report with Bob if he has the corresponding infrastructure in place. This way, the CEO can be sure that the business reports are not leaked—be the attempt intentional or inadvertent. Note that in general the CEO may also provide more complex policies consisting of additional rules such as "only use with application X," "delete after 30 days," or "do not modify."

CODASPY'13, February 18–20, 2013, San Antonio, Texas, USA.

1.2 Realization of Usage Control

Within one computer system, data exists in different representations (e.g., file, pixmap, Java object) at different system layers (operating system, window manager, Java virtual machine). Therefore, the flow of data must be tracked at and across different system layers. It has been shown how usage control requirements can be enforced at each of these layers [28]. To this end, a generic data flow model [10, 29] as well as a generic enforcement infrastructure have been proposed and instantiated at several system layers [33]. By combining the instantiations of the data flow model and the enforcement infrastructures of each single layer, policies can be enforced at and across different layers *of one system* [33].

These instantiations of the data flow model do not take into account the fundamentally distributed nature of data usage control enforcement [10, 29, 33]. Other proposed models and implementations do not consider data flow tracking and/or fix data provider and data consumer beforehand [2, 12, 18, 20, 21]. Despite being the natural way of data dissemination in today's internet, generic and application-independent data dissemination in the context of usage control has, to our knowledge, not yet been investigated.

We address this shortcoming by extending a generic data flow model for intra-system data flow tracking to the case of cross-system data flow tracking. We use this model to develop a Data Distribution Infrastructure (DDI). As one example, we concretize the data flow model for internet communication using the Transmission Control Protocol (TCP) and instantiate both the concretized model and the DDI at the operating system layer. This allows for generic, transparent, and application-independent cross-system data flow tracking, transferring usage control policies along with to-be controlled data, and extending the scope of usage control enforcement to any system receiving data.

Big Picture. If usage control requirements are to be enforced within a single system, a *Local Enforcement Infrastructure* (LEI) must be deployed on that system. The task of the LEI is to (i) track the flow of data within and across several system layers, (ii) take usage control policy decisions, and (iii) enforce these decisions. Several LEIs have been described and implemented [7, 10, 18, 29, 33, 44]. For distributed systems, an additional *Data Distribution Infrastructure* (DDI) is needed. Its task is to (i) track the flow of data across different connected systems, (ii) transfer the corresponding policies along with the data, and (iii) trigger the receiving LEI to take care of local enforcement. The DDI thus provides additional functionalities for usage control enforcement in distributed system environments. The subject of this paper is the development of a generic DDI and its integration with generic LEIs as depicted in Fig. 1. Because the DDI transfers policies but is not responsible for local enforcement, the content of any specific usage control policy is irrelevant from the DDI's perspective. As a consequence, in this paper, we do not provide any concrete examples of policies (see [7, 10, 18, 29, 33, 44] for various LEIs).

1.3 Organization

We organize our work along six steps as follows:

Step A: Generic data flow model, §2.1. In order to enforce usage control policies on all representations of a particular data item, the flow of that data must be tracked both within one system and across systems. We recap a generic data flow model, a transition system that has been designed to allow for data flow tracking within single systems.

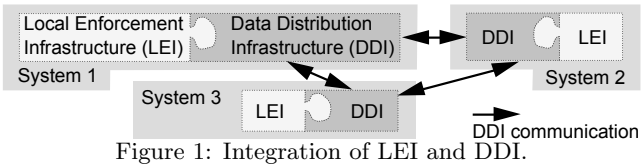

Figure 1: Integration of LEI and DDI.

Step B: Cross-system data flows, §2.2. We extend the data flow model from step A to the cross-system case.

Step C: Concretization, §3. As one example, we concretize the model of step B for TCP; we identify TCP-related actions and describe how they change the data flow state.

Step D: DDI, §4. We develop a Data Distribution Infrastructure (DDI) that allows for transparent and application-independent cross-system data flow tracking and that transfers policies along with data to be controlled. Using this infrastructure, the scope of policy enforcement can be extended to any system receiving usage controlled data.

Step E: Integration with LEI, §4. We integrate our DDI with a Local Enforcement Infrastructure (LEI) for single independent systems, thus combining cross-system and intra-system data flow tracking and policy enforcement.

Step F: Instantiation, §5. We instantiate and implement both the concretized model and the integrated infrastructure for the OpenBSD operating system.

We evaluate our work in §6. In §7, we discuss the limitations of our work and point to future work. §8 puts our work in context. §9 concludes.

1.4 Contribution and Assumptions

In sum, we tackle the following **research problem**. If data usage is to be controlled across system boundaries, then policies need to be (1) transferred together with data when data is transmitted and (2) enforced at the receiving end. Current instantiations and implementations of usage control models do not consider generic data flows in-between different connected systems but rather focus on one single machine. Other solutions fix data provider and data consumer beforehand, therefore not catering to the complexity of today's internet environment, where data may be arbitrarily redistributed by any data possessor.

Our **solution** is a model for cross-system data flow tracking and its concretization for TCP. We develop and deploy a DDI, realizing application-independent cross-system data flow tracking and the sticky policy paradigm for usage control. We instantiate our concepts for the OpenBSD operating system and evaluate our work. A demo video is available at `http://www22.cs.tum.edu/index.php?id=64`.

We see our **contribution** in the development of a generic usage control enforcement model for distributed systems. We are not aware of usage control solutions that generically, transparently, and application-independently (1) track the flow of data both within one system and in-between different connected systems, and (2) extend the scope of policy enforcement to any system receiving usage-controlled data.

An implementation of the system connected to a smart metering system has been published as a short demo paper before [13]. In contrast, the present paper describes the underlying theory and evaluates and discusses the approach.

Assumptions. To reduce complexity, we assume a static network structure: the same IP address may not be assigned to more than one host over time. Policy considerations such

as policy specification, policy translation, and policy evolution are out of the scope of this work; they are discussed in [17,34,35]. We assume an initial policy to be deployed and assume that the receiving end of a data transfer is equipped with a usage control enforcement infrastructure. Policies are assumed to be formulated in terms of system calls as described in [17,33]. We discuss these assumptions in §6-§8.

Attacker Model. As motivated in §1.1, we consider non-privileged end users on both local and remote systems to be a threat w.r.t. sensitive data. Attempts to use usage controlled data without respecting the corresponding policy may be either intentional or inadvertent.

2. APPLICATION-INDEPENDENT CROSS-SYSTEM DATA FLOW TRACKING

To extend the scope of usage control to other systems over the network, data flow between the communicating systems must be tracked. We recap a generic data flow model in §2.1 and provide an extension in §2.2 that allows for tracking both cross-system and intra-system data flows.

2.1 Generic Data Flow Model

Step A. The data flow model presented in [10, 29] allows to overapproximate the existence of data item copies in a system by capturing the flow of data within this system. To this end, the distinction between abstract data (e.g., picture) and data representations (e.g., file or database entry), so-called containers, is made: containers are entities that may contain data. The data flow model is a transition system: states capture which data is stored in which container; state transitions are initiated by actions related to data flow and change the mapping between data and containers (i.e., which containers potentially contain which data).

Formally, the model is a tuple $(D, C, F, \Sigma, \sigma_i, P, A, \varrho)$; D is the set of data items whose usage is restricted by a policy, C is the set of containers, and F is the set of identifiers that are used to identify containers. $P \subseteq C$ are all possible principals in the system; principals may have read sensitive data (which is why they are considered a subset of C) and they can, as opposed to other containers, invoke actions from the set of all possible actions A. $\Sigma = (C \to \mathbb{P}(D)) \times (C \to \mathbb{P}(C)) \times (P \times F \to C)$ are all possible states of the system; σ_i is the initial state. A state therefore consists of three mappings: (1) A storage function $s : C \to \mathbb{P}(D)$ capturing which data is potentially stored in which container. (2) An alias function $l : C \to \mathbb{P}(C)$ capturing that some containers may implicitly get updated whenever other containers do: If $c_2 \in l(c_1)$ for $c_1, c_2 \in C$, then any data written into c_1 is immediately propagated to c_2. (3) A naming function $f : P \times F \to C$ capturing the mapping from principal-relative identifiers to containers. $f(p, n)$ thus returns the container that can be accessed by principal $p \in P$ via identifier $n \in F$.

Actions A change the system state; these changes are described by relation $\varrho \subseteq \Sigma \times P \times A \times \Sigma$. Additional notation for specifying state changes is needed. For any mapping $m : S \to T$ and an element $x \in X \subseteq S$, define $m[x \leftarrow expr]_{x \in X} = m'$ with $m' : S \to T$ such that $m'(y) = expr$ if $y \in X$ and $m'(y) = m(y)$ if $y \notin X$.

Multiple updates for disjoint sets are combined by function composition $\circ$. The replacements are done simultaneously and atomically; the semicolon is syntactic sugar:

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

Function f^- returns the set of all names for a given container: $\forall\, c \in C : f^-(c) = \{(p, n) \in P \times F \mid f(p, n) = c\}$

The described model was originally designed to model the flow of data within one system. In §2.2 we provide an extended model to allow for both intra-system and cross-system data flow tracking from a global point of view.

2.2 Cross-system Data Flow Tracking for IP

All major application-level protocols (e.g., HTTP, FTP, SMB, SSH, DNS) in today's internet build on the Internet Protocol (IP) which transfers data packets between internet hosts in a best-effort manner. Protocols at the transport layer bridge the gap between IP (host-to-host communication) and application-layer protocols (end-to-end application communication) by delivering the data packets addressed to a particular host to the correct process running on that host.

Step B. On the basis of the model of step A we provide a model that allows for both intra-system and cross-system data flow tracking from a global point of view. Our model supports any protocol building on IP and is applicable to all unicast internet-based communication.

Hosts. Since we investigate cross-system data flows, we need to introduce the concept of hosts. In real-world systems, multiple IP addresses may be assigned to the same host, which is why we define a host as a set of IP addresses. We consider all IP addresses to be globally unique: no IP address may be assigned to more than one host over time. Exceptionally, each host can refer to itself by using several reserved IP addresses. For IPv4, these are all IP addresses starting with "127.", while in IPv6 the single address "::1" is reserved. We refer to any of these addresses as *localhost* (lo). We consider lo to be a reserved value within $IPAddr$ where $IPAddr$ is the set of all IP addresses. The set of hosts $H \subseteq \mathbb{P}(IPAddr \setminus \{lo\})$ is defined such that

$$\forall\, h_x, h_y \in H : h_x \neq \varnothing \wedge h_y \neq \varnothing \wedge (h_x \cap h_y \neq \varnothing \Rightarrow h_x = h_y)$$

Therefore, each host $h_z \in H$ is identified by its set of globally unique IP addresses. Note that $\forall\, h_z \in H : lo \notin h_z$. Additionally, we define $Port$ as the set of network ports.

Principals. Principals P are processes. They are also containers ($P \subseteq C$) because their memory is a possible location for data. Each process runs on exactly one host, while a host can run multiple processes at the same time. Each process $p \in P$ is assigned a host-relative process ID (PID). Thus, the set of principals P is defined as $P = H \times PID$. Function $h : P \to H$ returns for each process $p \in P$ its host $h_p \in H$. In order to model the fact that a network communication endpoint (i.e., a network socket) bound to IP address $lo \in IPAddr$ is only able to communicate with processes running on the same host, we define $\forall\, p \in P, \forall\, a \in h(p) \setminus \{lo\} : scope(p, lo) = \{q \in P \mid h(p) = h(q)\}$ and $scope(p, a) = P$ as the set of all processes that can communicate with process p via p's network socket that is bound to IP address a.

Containers. We consider both network sockets (C_S) and the runtime memory of each process (i.e., the processes themselves) as containers: $C_S \cup P \subseteq C$.

Identifiers. Network socket containers are identified by process-relative file descriptors $e \in F_S$, which are only valid for the process that created the socket. For network communication, processes refer to other processes' sockets using an IP address and a port. Yet, this is not sufficient to uniquely identify a socket, since a process may use the same IP address and port for different communications. For this reason, a socket is identified by the IP address and port of the

caller and the IP address and port of the receiver, called *local socket name* and *remote socket name*, respectively. Note that the remote socket is not necessarily on a different machine. Hence, the set of identifiers F is $F = F_S \cup (F_N \times F_N)$ with $F_N = (IPAddr \times Port)$.

Actions. We consider all system calls related to IP networking and writing to and reading from file descriptors as actions. System calls are provided by the operating system kernel and may be invoked by user-space processes to access resources, communicate, retrieve system information, and the like. Actions (system calls) are performed by principals (processes) and may change the system's data flow state as defined by ϱ (cf. §3).

The presented model allows for intra-system and cross-system data flow tracking for any unicast communication method building upon IP. As one example, §3 concretizes this extended generic data flow model for TCP.G

3. CROSS-SYSTEM DATA FLOW TRACKING FOR TCP

Since IP does not provide means for end-to-end application communication, different protocols at the transport layer bridge the gap between IP and the corresponding application-layer protocol. The most well-known protocols at this layer are the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP). While UDP is a connectionless protocol (no dedicated connection is established between the communication partners and no delivery guarantees are provided), TCP is a connection-based protocol providing reliable data delivery. Using TCP, a client and a server process establish a full-duplex connection before exchanging data. Two network sockets exist, each allowing for sending and receiving data on one end of the connection.

We model and realize cross-system data flow tracking at the level of TCP. Before we model the protocol in detail in terms of ϱ in §3.1-§3.3, let us provide a high-level overview of TCP-related system calls (cf. Fig. 2): First, both client and server create a communication endpoint, called socket, (system call *socket*) and bind a name, i.e. an IP address and a port, to it (*bind*). The server then marks its socket as passive (*listen*) and waits for incoming connections (*accept*). The client then initiates a connection to the server's passive and listening socket (*connect*). Once *accept* and *connect* return, the TCP communication channel has transparently been set up by the underlying operating systems. The processes may then exchange any kind of information by writing to and reading from the network sockets using a variety of system calls (e.g., *write*, *read*). Finally, the communication channel is torn down (*shutdown*, *close*, *exit*).

Step C. We concretize the cross-system data flow model for TCP networking at the layer of the operating system. At this layer, TCP-relevant actions are system calls [10, 21]. Since most application layer protocols rely on TCP, our concretization supports a variety of internet protocols, including web browsing, e-mail, and file transfer. In order to model the data flow according to our model (cf. §2.2), we need to define the transitions ϱ for TCP-related system calls.

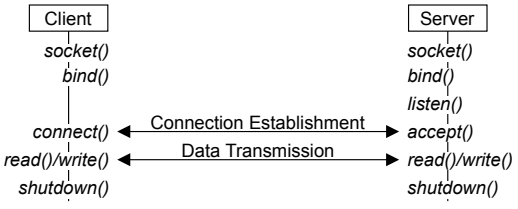

Figure 2: Sequence of TCP-related system calls.

3.1 Connection Establishment

socket. First, each communication partner must execute the *socket* system call. With parameter SOCK_STREAM *socket* creates a new unconnected socket for connection-based communication (i.e., TCP) on top of IP. *socket* returns a file descriptor $e \in F_S$ that identifies the newly created socket container $c \in C_S$ for the calling process $p \in P$:

$$\forall s \in [C \to \mathbb{P}(D)], \forall l \in [C \to \mathbb{P}(C)], \forall f \in [P \times F \to C],$$
$$\forall p \in P, \forall e \in F_S, \forall c \in C_S :$$
$$((s, l, f), p, socket(e, c), (s, l, f[(p, e) \leftarrow c])) \in \varrho.$$

bind, listen. After creating a socket, each communication partner $p \in P$ must *bind* the local socket name to its socket. *bind* is especially important for the server process, because it will be waiting for incoming connections and its socket name must therefore be fixed and known. The server process then marks its socket as passive using system call *listen*. A listening socket may neither initiate connections nor be part of an actual communication channel. *bind* and *listen* do not change the data flow state.

accept. The server process then performs an *accept* system call on the passive and listening socket. *accept* does not return until an actual connection establishment request to that socket has been made. We discuss the *accept* response and the respective state transition shortly.

connect. The client process then initializes the actual connection establishment by issuing system call *connect*. Parameters are file descriptor $e \in F_S$ of the client's socket, as well as IP address $a_S \in IPAddr$ and port $x_S \in Port$ of the server's listening socket. If the client's socket has not been bound explicitly before, *connect* does an implicit call to *bind*. *connect* returns successfully, once the connection to (a_S, x_S) has been established. Parameters $a_c \in h(p)$ and $x_c \in Port$ correspond to the local socket name of the client's connected socket. This information is retrievable from the operating system via e once the connection has been established:

$$\forall s \in [C \to \mathbb{P}(D)], \forall l \in [C \to \mathbb{P}(C)], \forall f \in [P \times F \to C],$$
$$\forall p \in P, \forall e \in F_S, \forall a_S \in IPAddr, \forall a_C \in h(p),$$
$$\forall x_S, x_C \in Port :$$
$$((s, l, f), p, connect(e, a_S, x_S, a_C, x_C), (s, l,$$
$$f[(q, ((a_C, x_C), (a_S, x_S))) \leftarrow f(p, e)]_{q \in scope(p, a_C)})) \in \varrho.$$

The establishment of the communication channel is modeled along with the *accept* response at the server's side as follows. Note that our model assumes that *connect* always returns before *accept*; we cater to this assumption in §5.1.

***accept* (cont.).** Once the server's *accept* returns, a new communication channel has been established by the operating systems. For this purpose, a new socket has been created and connected to the remote (client's) socket that requested the connection establishment. Output parameters of *accept* are a socket file descriptor $e \in F_S$ referring to the newly created socket container $c \in C_S$ and the local socket name $(a_C, x_C) \in F_N$ of the client's connected socket. $(a_S, x_S) \in h(p) \times Port$ refers to the local socket name that can be retrieved from the operating system via e.

Connection establishment finally is modeled by bidirectionally aliasing the server's and client's socket containers:

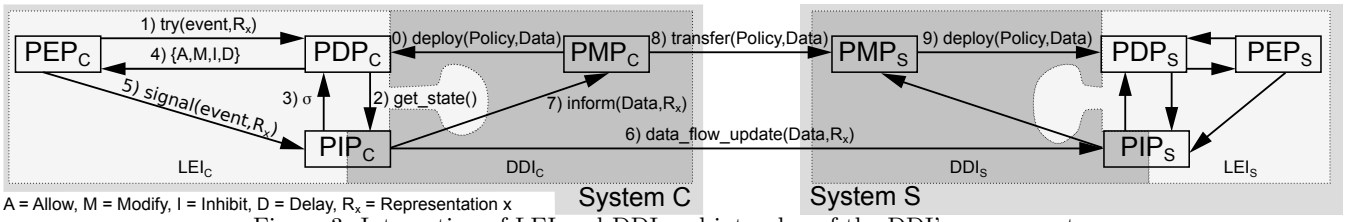

Figure 3: Integration of LEI and DDI and interplay of the DDI's components.

$$\forall\, s \in [C \to \mathbb{P}(D)], \forall\, l \in [C \to \mathbb{P}(C)], \forall\, f \in [P \times F \to C],$$
$$\forall\, p \in P, \forall\, e \in F_S, \forall\, c \in C_S,$$
$$\forall\, a_S \in h(p), \forall\, a_C \in \mathit{IPAddr}, \forall\, x_S, x_C \in \mathit{Port}:$$
$$((s, l, f), p, \mathit{accept}(e, a_C, x_C, a_S, x_S, c), (s,$$
$$l[c \leftarrow f(p, ((a_C, x_C), (a_S, x_S)));$$
$$\quad f(p, ((a_C, x_C), (a_S, x_S))) \leftarrow c],$$
$$f[(p, e) \leftarrow c;$$
$$\quad (q, ((a_S, x_S), (a_C, x_C))) \leftarrow c]_{q \in \mathit{scope}(p, a_S)})) \in \varrho.$$

3.2 Data Transmission

Once the TCP connection has been established, the corresponding processes may write to and read from the communication channel. Modelling sending and receiving of data then corresponds to writing to and reading from any other file descriptor as described in [10]. For completeness, we cite the corresponding state changes for system calls *write* and *read*; l^* denotes the reflexive transitive closure of function l.[1] Other system calls for sending are *sendmsg*, *pwritev*, *pwrite*, *writev*, *send*, *sendto*; system calls for reading are *recvmsg*, *preadv*, *pread*, *readv*, *recv*, *recvfrom*. Their state transitions are analogous: when writing, (potentially) all knowledge of the process flows into the socket container and recursively into all aliased containers. When reading, (potentially) all knowledge from the socket container flows into the reading process and recursively into all aliased containers:

$$\forall\, s \in [C \to \mathbb{P}(D)], \forall\, l \in [C \to \mathbb{P}(C)], \forall\, f \in [P \times F \to C],$$
$$\forall\, p \in P, \forall\, e \in F_S:$$
$$((s, l, f), p, \mathit{write}(e), (s[t \leftarrow s(t) \cup s(p)]_{t \in l^*(f(p,e))}, l, f)) \in \varrho.$$

$$\forall\, s \in [C \to \mathbb{P}(D)], \forall\, l \in [C \to \mathbb{P}(C)], \forall\, f \in [P \times F \to C],$$
$$\forall\, p \in P, \forall\, e \in F_S:$$
$$((s, l, f), p, \mathit{read}(e), ($$
$$s[t \leftarrow s(t) \cup s(f(p, e))]_{t \in l^*(p)}, l, f)) \in \varrho.$$

3.3 Connection Teardown

After data transmission the connection is shut down. System calls *shutdown*, *close*, and *exit* cause a (potentially partial) connection teardown.

shutdown. Using the *shutdown* system call, process $p \in P$ may shut down all or part of the connection constituted by the socket identified by file descriptor $e \in F_S$. Parameter SHUT_RD disallows further receptions, SHUT_WR disallows further transmission, and SHUT_RDWR forbids further receptions and transmissions. In terms of the data flow model, this has the following implications: In case of SHUT_RD, the socket container is emptied and all aliases to it are deleted. In case of SHUT_WR, all aliases from the socket container are deleted. In case of SHUT_RDWR, the socket container is emptied and all aliases to and from it are deleted; additionally, all its identifiers of type $F_N \times F_N$ are deleted. We use the reserved value $\mathit{nil} \in C$ to refer to non-existing containers.

$$\forall\, s \in [C \to \mathbb{P}(D)], \forall\, l \in [C \to \mathbb{P}(C)], \forall\, f \in [P \times F \to C],$$
$$\forall\, p \in P, \forall\, e \in F_S:$$
$$((s, l, f), p, \mathit{shutdown}(e, \mathit{SHUT\_RD}), (s[f(p, e) \leftarrow \varnothing],$$
$$l[c \leftarrow l(c) \backslash \{f(p, e)\}]_{c \in C}, f)) \in \varrho.$$

$$\forall\, s \in [C \to \mathbb{P}(D)], \forall\, l \in [C \to \mathbb{P}(C)], \forall\, f \in [P \times F \to C],$$
$$\forall\, p \in P, \forall\, e \in F_S:$$
$$((s, l, f), p, \mathit{shutdown}(e, \mathit{SHUT\_WR}), (s,$$
$$l[f(p, e) \leftarrow \varnothing], f)) \in \varrho.$$

$$\forall\, s \in [C \to \mathbb{P}(D)], \forall\, l \in [C \to \mathbb{P}(C)], \forall\, f \in [P \times F \to C],$$
$$\forall\, p \in P, \forall\, e \in F_S:$$
$$((s, l, f), p, \mathit{shutdown}(e, \mathit{SHUT\_RDWR}), (s[f(p, e) \leftarrow \varnothing],$$
$$l[f(p, e) \leftarrow \varnothing;\ c \leftarrow l(c) \backslash \{f(p, e)\}]_{c \in C},$$
$$f[x \leftarrow \mathit{nil}]_{x \in \{(q,n) \in f^-(f(p,e)) \mid n \in F_N \times F_N\}})) \in \varrho.$$

close, exit. Process $p \in P$ may close a file descriptor $e \in F_S$ using system call *close*. The behaviour of *close* is modeled as described in [10] by mapping identifier (p, e) to *nil*. Yet, if (p, e) is the last remaining file descriptor for socket $c = f(p, e)$ (i.e., $P \times F_S \cap f^-(f(p, e)) = \{(p, e)\}$), the connection is implicitly shut down by the operating system. In this case, we model an implicit *shutdown* with parameter SHUT_RDWR. When a process exits (system call *exit*), all of its file descriptors and TCP connections are closed alike.

We have concretized the cross-system data flow model by defining transition relation ϱ for TCP-related system calls. It enables us to know which data is—due to over-approximations induced by the semantics of the write system call: potentially—stored on which system whenever data has been transferred via TCP. In §4 we develop a usage control enforcement infrastructure; §5 will show how this technical infrastructure uses the presented cross-system data flow model.

4. A DISTRIBUTED ENFORCEMENT INFRASTRUCTURE

Step D. In order to practically extend the scope of usage control enforcement to the system that receives usage-controlled data, we develop a Data Distribution Infrastructure (DDI, Fig. 3) to (1) track cross-system data flows as modeled in §2.2 and §3, (2) transfer policies along with the to-be controlled data, and (3) deploy the policy at the receiving end where the respective local enforcement infrastructure is responsible for its local enforcement. The main components are a distribution-enhanced Policy Information Point (PIP), which implements ϱ as defined in §3, and a

[1] $\forall\, a \in C : l^*(a)$ is the smallest set satisfying $l^*(a) = \{a\} \cup \{b \in C \mid b \in l(a) \vee (\exists\, c \in l(a) \wedge b \in l^*(c))\}$.

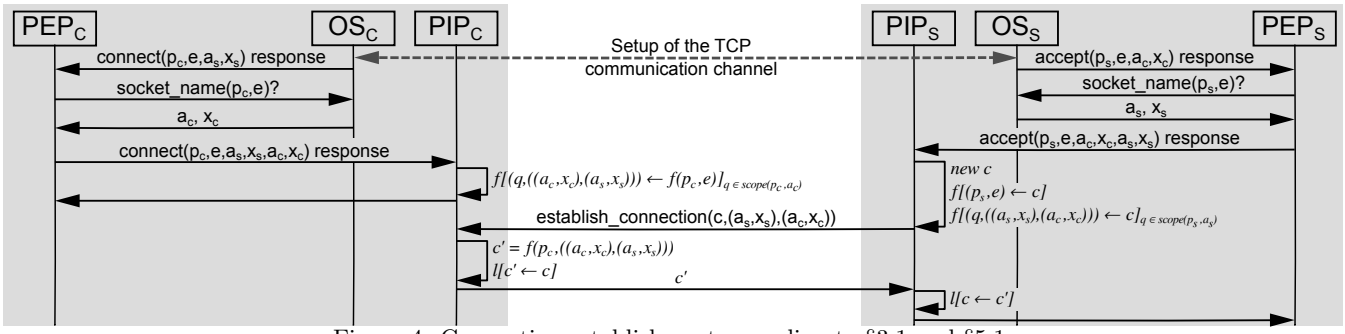

Figure 4: Connection establishment according to §3.1 and §5.1.

Policy Management Point (PMP). The infrastructure is distributed in that its components must be deployed on any system that is expected to enforce usage control policies. Each PIP holds the data flow state of the system on which it is deployed. The PMP manages all usage control policies for data entering, leaving, and residing in the system on which it is deployed. Both the PIP and the PMP are able to communicate with their respective counterparts on other systems. Thus they allow for the exchange of information regarding cross-system data flows and usage policies when data flows between systems take place. We provide more details of our infrastructure when presenting its instantiation in §5. At this point we assume that the receiving system has the necessary infrastructure in place; this is discussed in §6.

Step E. We integrate the DDI into a Local Enforcement Infrastructure (LEI) for single independent systems [10, 33]. The main components of the latter are a Policy Enforcement Point (PEP), a Policy Decision Point (PDP), and a local Policy Information Point (PIP). Fig. 3 shows an instantiation of the integrated infrastructure for two operating system instances, C and S.

Initially, the PMP of system C deploys a usage control policy (Fig. 3, step 0) for some specified data. From this point onward, the LEI monitors the corresponding data, tracks its local copies—including any derivations even after operations such as compression or encryption—, and enforces the policy upon every usage of that data as follows: The PEP is tailored to one system layer (in our case the operating system); its task is to intercept attempted and actual events within this layer (system calls). The PEP temporarily blocks the execution of these events and signals them to the PDP (step 1). The PDP decides for each event whether it conforms to the deployed usage control policy. In order to take this decision, the PDP queries the PIP for additional information about the data flow state (steps 2,3) and then decides, on the grounds of this information and the usage control policy, whether to allow, inhibit, delay, or modify the event in question [31]. The PDP returns the decision to the PEP (step 4) which enforces it. If an actual event happened, the PEP signals the event to the PIP (step 5) that then updates the system's data flow state accordingly. [10] describes how the state evolves for intra-system data flow system calls and thus also how any modifications to the data are tracked.

If a TCP-related system call happens on system C (analogous for system S), it is intercepted and evaluated by the components of LEI_C (PEP_C, PDP_C). The PIP_C component of DDI_C, which implements the transition relation ϱ described in §3, then communicates the fact that the system call happened as well as the relevant parameters to PIP_S (step 6), thus realizing cross-system data flow tracking. Upon data transmission, PIP_C additionally informs PMP_C (step 7). PMP_C then transfers the respective usage control policies to PMP_S (step 8), thereby realizing the sticky policy paradigm. PMP_S eventually deploys the policy on system S (step 9). Details for steps 6 through 9 are provided in §5.

5. INSTANTIATION

Step F. To show the usefulness of our approach, we instantiate the concretized cross-system data flow model (§3) and the integrated enforcement infrastructure (§4) for the OpenBSD operating system. As explained in §4, all usage control enforcement components are deployed on each operating system instance. While the PEP must be deployed locally, this is not inevitable for PDP, PIP, and PMP. However, deploying PDP and PIP remotely would lead to communication overhead and thus to higher system response times and lower performance. This is because system calls happen frequently and usage control decisions must be taken for each. Hence, we chose to deploy all components locally. PIPs keep the data flow state of their local system. The work presented here extends the knowledge of local PIPs with information about data that has been communicated to or from other PIPs. The product of the *storage*, *alias*, and *naming* functions (cf. §2.1) of all PIPs corresponds to the system's global data flow state.

Systrace [36] has been used to implement the PEP; it allows to intercept, observe, modify, and prohibit system calls both before and after their execution by the kernel. Using systrace, no modifications to the operating system itself are needed; details are provided in [10, 36]. While LEIs at the operating system layer have been built before [7, 10, 44], our DDI implementation complements this work for TCP-based data flows and policy enforcement on multiple machines. We will now look into the crucial parts of the implementation.

5.1 Connection Establishment

If two communicating processes, p_S and p_C (server and client), run on the same host, the local PIP tracks the data flow through the local TCP connection. In contrast, if p_S and p_C run on two different hosts (cf. Fig. 4), the PIPs of the two DDIs must communicate the fact of connection establishment. As soon as the TCP communication channel has been set up by the underlying operating systems, both the server's *accept* and the client's *connect* return and are intercepted by the corresponding PEPs. On the server side, PEP_S then asks OS_S for the assigned local socket name (a_s, x_s) and notifies PIP_S that the event happened. PIP_S then creates a new socket container c and assigns the corresponding identifiers as described in §3.1 by updating naming function f. PIP_S then communicates the socket's ID (c),

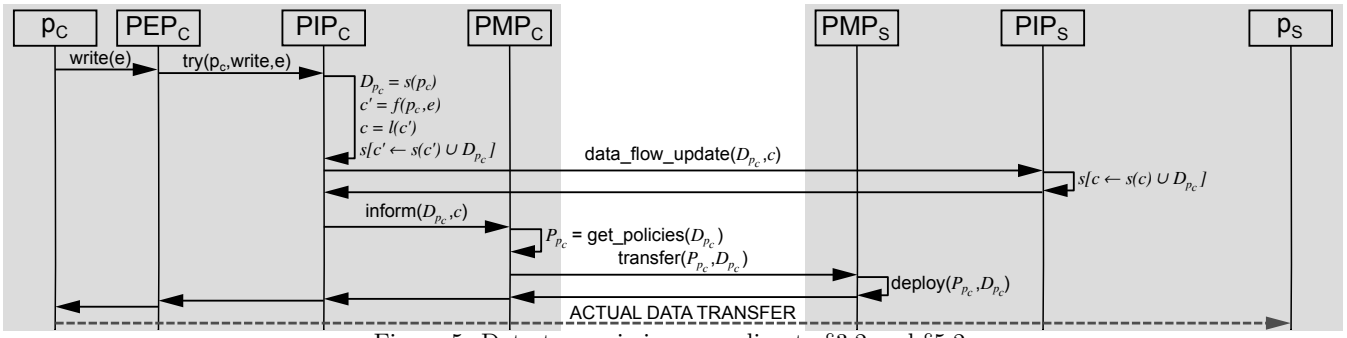

Figure 5: Data transmission according to §3.2 and §5.2.

its local socket name (a_s, x_s), and its remote socket name (a_c, x_c) to PIP_C. If this remote procedure call (*establish_connection*()) is about to be handled by PIP_C before the corresponding client's *connect* response, it is put on hold until the client's *connect* returned. Similarly, upon return of the client's *connect*, PEP_C asks OS_C for the assigned local socket name (a_c, x_c) and notifies PIP_C, which then assigns the corresponding identifiers according to §3.1 by updating naming function f. From the information sent by PIP_S, PIP_C can then identify the client's socket container (c') and create the alias to the server's socket container (c) by updating the alias function l as described in §3.1. PIP_C replies to PIP_S with the ID of the client socket container (c') and PIP_S then creates the second alias from c to c'.

5.2 Data Transmission

After connection establishment, p_S and p_C may cause cross-system data flows by writing to the TCP channel. If p_C (analogous for p_S) executes sytem call *write* (or any equivalent, cf. §3.2) on a file descriptor referring to a TCP channel to p_S, PIP_C and PIP_S realize cross-system data flow tracking, while PMP_C and PMP_S implement the sticky policy paradigm as follows (cf. Fig. 3 steps 6-9 and Fig. 5)[2]:

When p_C's *write* is intercepted and temporarily blocked by PEP_C, PIP_C is informed and updates its storage function s according to §3.2. PIP_C then performs a call to PIP_S conveying the information that the set of abstract data items D_{p_c} (i.e., all data read by p_C) is going to be transmitted to the aliased socket container c (*data_flow_update*()). PIP_S therefore updates its storage function s according to §3.2. PIP_C then informs PMP_C about the transmission of D_{p_c} to socket container c (*inform*()), and PMP_C performs a call to PMP_S, transferring the set of usage control policies P_{p_c} that apply to any of the data items in D_{p_c} (*transfer*()). PMP_S then deploys the policies on system S. On success, p_C's *write* is unblocked and the actual data transfer succeeds. While they are conceptually different, our implementation bundles the two remote procedure calls (*data_flow_update*(), *transfer*()) for performance reasons.

As soon as p_S reads from the aliased socket container c, PDP_S and PIP_S are already aware of the cross-system data flow and the corresponding policies, therefore extending usage control to system S. Note that PIP_C's data flow state may change in-between two *write* system calls of p_C (e.g., if p_C reads additional data). In this case the remote communication (i.e., *data_flow_update*(), *transfer*()) needs to be repeated upon the next *write* system call.

[2]We assume that PDP_C decided to allow the *write*; otherwise no cross-system data flow (tracking) would happen.

5.3 Connection Teardown

If a TCP connection between two processes on different hosts is torn down, this fact is communicated between the corresponding PIPs; parameters are the connection's identifiers (local socket name and remote socket name) and the type of connection teardown. Each PIP is then responsible for updating its state according to §3.3.

Note. By its very nature TCP communication is limited to two communication partners. Thus all considerations in §3-§5 are limited to a client and a server system. For distributed scenarios with a larger number of systems our infrastructure and its features apply transitively.

6. EVALUATION

6.1 Provided Guarantees

If the presented integrated infrastructure (LEI and DDI) is in place and not tampered with, the CEO (cf. §1.1) can now be sure that all copies and derivations of business reports are accompanied with the usage control policy he specified (duty of the DDI), and that employees can use the business reports only in accordance with the policy (duty of the LEI). As described in §2-§5, the CEO's policy is always transferred along with the business reports and enforced on any receiving system—as long as TCP is used as the transport layer. Via intra-system and cross-system data flow tracking at the level of system calls, all copies and derivations of the business reports are detected and protected on both local and remote systems. While our DDI realizes cross-system data flow tracking and the sticky policy paradigm, the LEI is responsible for intra-system data flow tracking, policy decisions and policy enforcement.

Assume that the CEO has defined a weaker policy that permits Alice to share business reports with the PR department after blackening certain parts of the report. Our system would make sure that if Alice sends the report to Charlie, (1) Alice's LEI blackens the corresponding parts and (2) Alice's DDI would send the policy along with the modified report to Charlie. Charlie's usage control infrastructure would work identically. A further recipient outside of the company would then get the report from Charlie only after blackening further details. Different semantics of this kind of requirements are discussed in [35]. In other words, using our system, enforcement of usage control requirements is not restricted to the initial data provider and the initial data consumer, but rather to all (transitive) consumers of a data item, as long as they are equipped with LEI and DDI.

We obviously assume that the receiving system has the necessary distributed usage control infrastructure in place.

Our current instantiation gives some kind of weak assurance for this fact by (1) detecting whether usage controlled data is transmitted to another system and (2) disallowing the data transfer if the receiving system does not follow the infrastructure's protocol. We are aware that this only constitutes an interim solution while more sophisticated solutions are being developed, since attackers could easily circumvent such a solution (cf. §6.2).

Considering the increasing amount of sensitive digital information in various environments (e.g., businesses, governmental institutions, car to car communication, smart meters, cyber-physical systems) as well as the increasing number of embedded and integrated devices (e.g., phones, tablets, cars, smart meters), we are convinced that within (semi-)closed environments such usage control solutions can and will become reality. For this, users must not have administrative privileges on their computing devices and there must not be means to modify underlying hard- and software. In contrast, in completely open environments it is unclear whether such technologies can become reality, since trusted computing technologies would be needed to assure the existence of a usage control infrastructure on a remote machine.

6.2 Attack Vectors

While we are aware that a security analysis cannot be exhausting, the purpose of this section is to understand the attack surface and vulnerabilities in our work. We consider two attacker models: non-privileged end users (root users cannot be controlled but only observed [25]) and a man-in-the-middle between the two communicating systems. Goals of the attackers (w.r.t. usage control in general and our infrastructure in particular) may be to (1) use usage-controlled data without respecting the applicable policies or (2) render the usage control infrastructure useless or unusable.

Assumptions. Since our work focuses on the distributed aspect of usage control, we assume that usage control policies are enforced once they have been deployed locally. By tackling usage control at the operating system layer, we must assume that the operating system and the underlying hardware are not vulnerable and that the user does not have root privileges, since a root user could easily switch off the usage control infrastructure. The latter assumption could be avoided by exploiting trusted computing technologies to assure that the usage control infrastructure, the operating system, as well as underlying hardware are in a proper state and that they cannot be modified in an unwanted way (i.e. by disabling the enforcement infrastructure), cf. §8.

Non-TCP Communication. Since our implementation investigates usage control at the TCP-layer, data could unrestrictedly be redistributed using other transport layer protocols, e.g. by tunneling TCP via UDP [23] or by redistributing data via non-TCP application-layer protocols such as DNS. Additionally, users can share usage controlled data using protocols such as USB or Bluetooth. These attacks can be addressed by disallowing any non-TCP data flows. Since this is unlikely to be accepted in real world scenarios, usage control must be extended to support these protocols. Our model for cross-system data flow tracking has been developed for unicast communication using the Internet Protocol (IP); in this paper TCP serves as just one example.

Portable media. In general, users have the possibility to save usage controlled data to portable media, physically removable hard disks, or the like. Such attacks can be prevented by inhibiting all attempts to save usage-controlled data to portable media or by mediating all corresponding actions through encryption tools such as TrueCrypt [42].

Fool Infrastructure. If data is sent, the DDIs of the communicating systems exchange information about cross-system data flows and usage control policies. A malicious user could set up fake usage control components, particularly the distributed parts of PIP and PMP, pretending to have an enforcement infrastructure in place. An honest infrastructure will then be fooled into thinking that the remote enforcement infrastructure is in place while it actually isn't. The malicious user would then receive usage controlled data without having an enforcement infrastructure in place. Certificates cannot eliminate this attack but introduce means for liability. Trusted computing technologies would be needed to defend against such attacks, cf. §8.

Man-in-the-Middle (MitM) Attack. A MitM attack can be performed both on the actual data transfer and on the DDI's communication.

Considering the actual data transfer, a MitM could sniff or modify the usage controlled data, and hence use the sniffed data without having a usage control infrastructure in place. Means to protect the data transfer exist, e.g. TLS [6], IPSec [14], tcpcrypt [3]. While TLS is commonly used for security-critical internet applications, it comes with the drawback that it must be supported by both the application-layer protocol and the corresponding applications. For non-TLS applications, tcpcrypt and IPSec can be integrated, since they are transparent to the application layer while providing similar guarantees in terms of confidentiality.

A MitM could also attack the DDI's communication by (1) inserting, dropping, or changing usage control policies or information regarding cross-system data flow (e.g., dropping messages between the PIPs) or by (2) changing the mapping between policies and data. Similarly, techniques like IPSec or tcpcrypt may be used to transparently secure this communication without interference with our infrastructure.

(Distributed) Denial-of-Service ((D)DoS) Attack. (D)DoS attacks may be mounted both by the local user and by a remote user by causing high amounts of system calls. While a local user could mount such an attack by issuing any kind of system calls, a remote user could do so by sending (potentially from different sources) messages to open ports that then result in system calls for the corresponding process. Such attacks may lead to high system response times and in the worst case render the usage controlled system unusable. Since in this case no data usage is possible at all (all system calls are handled sequentially by the same PEP), soundness of the infrastructure (i.e., no non-controlled usage of data) is not compromised by such an attack. Of course, availability of data can then no longer be guaranteed.

6.3 Performance

Influencing Factors. Hardware as well as system and network load will impact measurement results; in our case network interfaces and bandwidth are particularly important. The amount of system calls plays a significant role, since we intercept the security-relevant ones and take usage control decisions at each. This amount is given by the applications used for performance measurements, i.e. client and server process. Another influencing factor is the time

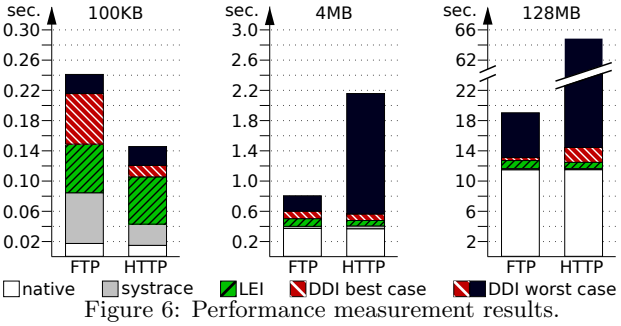

Figure 6: Performance measurement results.

needed for policy evaluation, which heavily depends on the complexity of the policies that must be evaluated[3].

Test Setup. For testing the performance of cross-system data flow tracking, policy transfer, and policy deployment (being the crucial parts of the presented DDI), we transferred files of sizes 100KB, 4MB, and 128MB between a client process and a server process using HTTP and FTP—two of the most popular protocols for transferring data on today's internet. Both server and client run the integrated infrastructure consisting of (1) systrace for intercepting system calls, (2) a LEI for intra-system data flow tracking and policy enforcement (§4, step E, [10]), and (3) our DDI as instantiated in §5.

We ran our tests on two dedicated computers[4] that were linked via a 100Mbit switch. No services other than the enforcement infrastructure and the applications needed for testing run on the machines. For file transfers via HTTP and FTP we used `Apache 2.2.15` and `vsftpd 2.3.2` as server, respectively. In both cases `wget 1.12` was used as client application. The time of data transfer was measured by invoking `wget` with the `time` command. Each test was repeated ten times; the observed standard deviation was negligible.

Since policy evaluations in the PDP are not subject of this paper, we deployed a policy that constantly evaluated to *allow*. Also, our evaluation does not focus on other overhead introduced by the LEI, since the overhead of such infrastructures has been discussed before [7, 10, 44].

Results. Fig. 6 and Table 1 show the results of our performance evaluation. The overhead stems from (1) systrace (□), (2) the LEI for tracking intra-system flows at the level of the operating system (▨), and (3) the overhead induced by this paper's DDI in the best case (▨) and in the worst case (the induced overhead being the sum of ▨ and ■ : ■▨).

Time for file transfer in seconds	native □	systrace ▫	LEI ▨	DDI best case ▨	DDI worst case ■▨
FTP 100KB	0.017	0.084	0.148	0.215	0.240
HTTP 100KB	0.014	0.043	0.105	0.120	0.145
FTP 4MB	0.369	0.397	0.499	0.597	0.800
HTTP 4MB	0.365	0.402	0.474	0.560	2.153
FTP 128MB	11.454	11.601	12.672	13.063	18.987
HTTP 128MB	11.463	11.602	12.426	14.394	64.816

Table 1: Detailed performance measurement results.

We differentiate the overhead of the DDI in a best case and a worst case scenario for the following reason: As described in §5.2, it may happen that the data flow state changes in-between two *write* system calls, e.g. if the corresponding process reads additional sensitive data. In this case the communication between the remote PIPs and PMPs must be repeated in order to update the data flow state of the receiving system and to transfer policies that have not been transferred before. In the worst case this communication must be repeated upon every *write*, while in the best case the PMPs and PIPs must communicate upon the initial *write* only.

[3]These are all policies referring to some data being stored in some container being a parameter of the system call.
[4]Each Athlon 64 X2 3800+, 4GB RAM, Gigabyte GA-K8NF-9 mainboard, Gigabit LAN; clean OpenBSD 4.9.

number of syscalls	FTP Server: `vsftpd` 100KB	4MB	128MB	HTTP Server: `Apache` 100KB	4MB	128MB
`write()`	43	139	4107	22	996	32740
total	148	358	8294	52	1523	49139
performance overhead factor w.r.t. native execution						
best case	11.66	0.61	0.14	7.29	0.53	0.25
worst case	13.15	1.16	0.65	9.01	4.90	4.65

Table 2: Number of system calls and performance overhead for transferring files in OpenBSD 4.9.

In the **best case** the accumulated overhead induced by the integrated infrastructure ranges from a factor of 0.14 to 11.66 w.r.t. native execution (□) (cf. Table 2). For transferring small files (100KB), we observe that the accumulated overhead is higher for FTP than for HTTP. This is because FTP operates on two TCP channels, a data channel and a control channel, and our DDI monitors both. Operating on two TCP channels also results in `vsftpd` issuing much more system calls for transferring the same file of 100KB than `Apache` (cf. Table 2). On the other hand, we observe that for the 128MB test case, the overhead induced by our DDI is higher for HTTP than for FTP. There are two reasons for this: First, the overhead for establishing and monitoring the FTP control channel is negligible for larger files. Second, `Apache` issues eight times more *write* system calls than `vsftpd` for transferring the same file (cf. Table 2). Since the DDI has to re-evaluate the data flow state upon every *write* in order to decide whether the PIPs and PMPs need to communicate, this leads to a higher overhead for HTTP.

In the **worst case**, the accumulated overhead ranges between factors of 0.65 to 13.15 w.r.t. native execution (cf. Table 2). Since in this case the PMPs and PIPs must communicate upon every *write*, the amount of *write* system calls determines the induced overhead. Again, since `Apache` issues much more *write* system calls for transferring large files, the accumulated worst case overhead is higher for HTTP.

In sum, large parts of the performance overhead induced by our infrastructure depends on the amount of system calls issued by the server process. For this reason, the performance overhead depends on the implementation of a protocol rather than the protocol itself. Also, we observed that the performance overhead is smaller when transferring larger files (cf. Table 2). Reasons for this are the bootstrapping process of the file transfer, which has smaller performance impact for large files, as well as the one-time policy transfer and data flow tracking in our best case scenario.

While we did not take into account other protocols or other implementations of HTTP/FTP, we are confident that our measurement results are close to best and worse case scenarios for other protocols and implementations as well. We base our confidence on two facts: First, the internal handling of buffers and system calls of `vsftpd` and `Apache` is quite different: While `vsftpd` operates with little *write* system calls on large buffers (4107 *write*s for 128MB), `Apache` operates with many *write*s on small buffers (32740 *write*s for 128MB). Second, HTTP and FTP are quite different from a techni-

cal perspective, since FTP operates on two TCP channels. Considering these differences and the fact that performance overheads for our best case scenario are similar[5], we are confident that our results are representative. Considering the difference in performance between our best case and worse case scenario, we are convinced that our best case results are close to real-world data usage scenarios, since (1) real world applications do not read additional data while sending data over the network and (2) usage control policies are rather long-lived and thus they usually do not have to be resend upon data transfer. Of course, additional overhead would be introduced by securing our infrastructure with IPSec [9], tcpcrypt [3], or remote attestation [15, 25, 38].

While we measured performance overheads of factors between 0.14 and 13.15 w.r.t. native execution (the worst case overhead is thus one order of magnitude), we contextualize these measurements in order to give an impression whether such overheads are acceptable. In case of a one-time transfer of 100KB, our measurements resulted in performance overheads between factors of 7.29 and 13.15. Considering that the transfer of a single file may then take 240ms instead of 17ms, such an overhead is likely to be acceptable in *user-interactive* workflows where such transfers happen occasionally. In contrast, when large amounts of small files are transferred in a sequence, e.g. 100 files of 100KB, a total transfer time of 24s instead of 1.7s is likely not to be acceptable in *user-interactive* applications. For transferring larger files in the best case scenario, performance overhead factors of 0.14 (FTP 128MB) to 0.61 (FTP 4MB) w.r.t. native execution are likely to be acceptable. In general, acceptability of performance overheads depends on the specific context. Note that we did not optimize performance of our prototype; the results thus show an upper bound for solutions like ours.

7. LIMITATIONS & FUTURE WORK

In our solution we focused on locally enforceable policies, meaning that the local information flow state is sufficient to take usage control decisions. We plan to extend our work to support usage control policies that refer to the global information flow state, therefore being able to enforce policies like "this data item must not reside in more than three systems," "not more than five instances of this process may be run at the same time," or "access that data only four times."

In this work we assumed a static network structure. Since technologies such as DHCP and network address translation (NAT) are in widespread use in today's internet, we plan to extend our work to this dynamic dimension.

Although our implementation supports most internet applications by realizing cross-system data flow tracking at the level of TCP, other important usage control applications like multimedia streaming and Voice over IP are not covered. This is because such applications use UDP as transport layer, as low-latency delivery is preferred to the guarantees provided by TCP. We thus plan to instantiate our model for UDP and/or tackle the problem of usage control at the level of IP. This way, we are also able to address some "non-TCP communication" attacks described in §6.2. This necessitates the definition of state transitions ϱ for UDP/IP and adjustment of the infrastructure to cope with the unreliability of UDP/IP. Multimedia streaming then also motivates to investigate multicast communication in usage control.

At the level of system call interposition, processes are considered black boxes that might write any of their knowledge into a TCP channel. By tainting the communication channel once usage-controlled data is (potentially) sent, all further data transmitted on the same channel is considered tainted as well. In order to overcome these overapproximations in an application-independent manner, we plan to (1) apply declassification techniques to our proposed solution and (2) bring in knowledge about the application-level protocol.

Our solution does not cover side channels such as timing attacks or power monitoring. Moreover, we did not put excessive effort into securing the prototypically implemented infrastructure. Solutions to this have been proposed and implemented; we give an overview in §8.

8. RELATED WORK

Application-dependent distributed usage control. Some distributed usage control concepts [12, 18] are application(-protocol) dependent by integrating the PEP into the application and incorporating policies into the application protocol. These solutions fix data provider and data consumer beforehand, thus not catering to bidirectional data flows and redistribution of data as usual in today's internet. [5, 21, 22] realize usage control for grid computational services by making the grid user deploy her application together with the policy. The application is monitored at the level of the Java Virtual Machine, whereby system calls are considered as security-relevant actions. Their approach is different from ours in that the policy is defined for the application instead of data. Since this approach does not consider data flow, cooperating applications could circumvent the usage control policy, which is not possible with our solution due to intra-system and cross-system data flow tracking.

Sticky Policies. In terms of sticking policies to data, the back channel model [4] is close to our approach. In this approach, on each system the communication between PEP and PDP is mediated through an application independent PEP (AIPEP). Once data is sent to another system, the AIPEP is responsible for sending the sticky policy to the AIPEP of the receiving system. The model differs from ours, since it was one of our goals to integrate seamlessly into an existing infrastructure for independent systems. Furthermore, we are not aware of corresponding implementations.

Trustworthiness of security mechanisms. As usage control is naturally distributed and policies must be enforced at the data consumer's site, the latter must implement at least the PEP. In order to convincingly spread distributed usage control mechanisms, any remotely deployed component must "behave in a 'good' manner and this manner [must be verifiable] by the policy stakeholder" [45]. To this end, solutions leveraging the Trusted Platform Module (TPM) have been proposed [25, 37, 39, 45]. Their basic idea is to verify the integrity of crucial system components (e.g., BIOS, Bootloader, Operating System, usage control infrastructure) before their execution and verify their integrity by comparing the measurements to a set of known "good" values. [2, 16] propose to send data that is to be controlled only to data consumers that persuade the data provider of having usage control mechanisms deployed.

Digital Rights Management (DRM). DRM [1, 8, 19, 24, 41] refers to concepts and techniques that aim at con-

[5] 100KB: 11.66 FTP overhead vs. 7.29 HTTP overhead; 4MB: 0.61 vs. 0.53; 128MB: 0.14 vs. 0.25; cf. Table 2.

trolling the usage and distribution of copyrighted, usually read-only, digital information at the data consumer's site. DRM can therefore be considered a specialization of usage control [32] that largely focuses on payment-based dissemination [26]; end users are not considered content providers: DRM does not provide means to protect their valuable (personal) information. While some solutions [1] work with central servers for storing keys, such a central component is not needed in our approach. Also, DRM solutions [1, 24] are tailored to specific file types and rely on specific applications to enforce digital licenses. Instead, our approach is independent of particular file types and applications.

Detective enforcement. The work presented in this paper enforces policies in a preventive manner: it is made sure that policies are adhered to. Enforcement can also be detective [27]: if a policy violation is detected, then various measures can be taken; prevention is thus replaced by accountability [43]. Our infrastructure can also be used for detective enforcement: rather than preventing Alice from forwarding the original business report to Charlie, the fact that she *did not* blacken certain parts before forwarding can be stored. A different approach is taken by Seneviratne [40] who directly embeds information accountability into HTTP. Different to our work, this approach is limited to one particular application-layer protocol and does not implement preventive enforcement mechanisms but rather gives users the opportunity to figure out how their data was misused and to take appropriate action.

9. CONCLUSIONS

Today's highly distributed computing environments lack mechanisms when it comes to enforcing restrictions on the usage of data after its release. This is relevant for both privacy and the protection of intellectual property or secrets. To fill this gap, we have extended a model that allows for intra-system data flow tracking to the cross-system case. We have concretized the model for tracking all TCP data flows at the level of the operating system by considering networking-related system calls as security-relevant actions. We based our work on a previously implemented infrastructure that can detect local *intra-system* data flows. In order to extend this infrastructure to the enforcement of usage control policies in *distributed* systems, we developed a distributed infrastructure for transparently and application-independently tracking *cross-system* data flows as well. In addition to tracking data flows to remote systems, this infrastructure takes care of transferring policies along with the data to be controlled. By integrating this infrastructure with an existing infrastructure for single independent systems, we manage to track data both within and across systems and enforce usage control policies at all systems processing usage controlled data. We instantiated and evaluated the data flow model and the enforcement infrastructure for the OpenBSD operating system. Our measurement results yielded overheads between factors of 0.14 and one order of magnitude w.r.t. native execution in a best case and a worst case scenario, respectively. We conclude that acceptability of such overheads depends on the specific context.

With an infrastructure like ours, it is now possible to not only track and control the local flow of data in-between different representations (files, pixmaps, Java objects, etc.) on one machine, but also across several machines. We believe that this constitutes a further step towards the representation-independent protection of data in today's company intranets as well as cloud-based systems on the internet.

Acknowledgment. The work described was funded by a Google Focused Research Award on Cloud Computing.

10. REFERENCES

[1] Adobe Systems Incorporated. Adobe Content Server, http://www.adobe.com/products/content-server.html, 2012.

[2] A. Berthold, M. Alam, R. Breu, M. Hafner, A. Pretschner, J.-P. Seifert, and X. Zhang. A Technical Architecture for Enforcing Usage Control Requirements in Service-Oriented Architectures. In *Proc. of the 2007 ACM Workshop on Secure Web Services*, pages 18–25, 2007.

[3] A. Bittau, M. Hamburg, M. Handley, D. Mazières, and D. Boneh. The case for ubiquitous transport-level encryption. In *Proc. 19th USENIX Conference on Security*, 2010.

[4] D. W. Chadwick and S. F. Lievens. Enforcing "Sticky" Security Policies throughout a Distributed Application. In *Proc. of the 2008 Workshop on Middleware Security*, pages 1–6, 2008.

[5] M. Colombo, F. Martinelli, P. Mori, and A. Lazouski. On Usage Control for GRID Services. In *International Joint Conference on Computational Sciences and Optimization*, pages 47–51, Apr. 2009.

[6] T. Dierks and E. Rescorla. RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2, 2008.

[7] D. Feth and A. Pretschner. Flexible Data-Driven Security for Android. In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 41–50, June 2012.

[8] D. Geer. Digital Rights Technology Sparks Interoperability Concerns. *IEEE Computer*, 37(12):20–22, 2004.

[9] G. Hadjichristofi, I. Davis, N.J., and S. Midkiff. IPSec Overhead in Wireline and Wireless Networks for Web and Email Applications. In *Proc. of the 2003 IEEE International Performance, Computing, and Communications Conference*, pages 543–547, 2003.

[10] M. Harvan and A. Pretschner. State-Based Usage Control Enforcement with Data Flow Tracking using System Call Interposition. In *Proc. 2009 Third International Conference on Network and System Security*, pages 373–380, 2009.

[11] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. A Policy Language for Distributed Usage Control. *Proc. 12th European Symp. on Research in Computer Security*, pages 531–546, 2007.

[12] B. Katt, X. Zhang, R. Breu, M. Hafner, and J.-P. Seifert. A General Obligation Model and Continuity-Enhanced Policy Enforcement Engine for Usage Control. In *Proc. 13th ACM Symposium on Access Control Models and Technologies*, pages 123–132, 2008.

[13] F. Kelbert and A. Pretschner. Towards a Policy Enforcement Infrastructure for Distributed Usage Control. In *Proc. 17th ACM Symposium on Access Control Models and Technologies*, pages 119–122, 2012.

[14] S. Kent and K. Seo. RFC 4301: Security Architecture for the Internet Protocol, 2005.

[15] C. Kil, E. Sezer, A. Azab, P. Ning, and X. Zhang. Remote Attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence. In *IEEE/IFIP Intl. Conf. on Dependable Systems Networks*, pages 115–124, July 2009.

[16] P. Kumari, F. Kelbert, and A. Pretschner. Data Protection in Heterogeneous Distributed Systems: A Smart Meter Example. In *INFORMATIK 2011 - Dependable Software for Critical Infrastructures*, 2011.

[17] P. Kumari and A. Pretschner. Deriving Implementation-level Policies for Usage Control Enforcement. In *Proc. 2nd ACM Conf. on Data and Application Security and Privacy*, pages 83–94, 2012.

[18] P. Kumari, A. Pretschner, J. Peschla, and J.-M. Kuhn. Distributed Data Usage Control for Web Applications: A Social Network Implementation. *Proc. 1st ACM Conference on Data and Application Security and Privacy*, pages 85–96, 2011.

[19] Q. Liu, R. Safavi-Naini, and N. P. Sheppard. Digital Rights Management for Content Distribution. In *Proc. Australasian Information Security Workshop Conference*, volume 21, pages 49–58, 2003.

[20] E. Lovat and A. Pretschner. Data-centric Multi-layer Usage Control Enforcement: A Social Network Example. In *Proc. 16th ACM Symposium on Access Control Models and Technologies*, pages 151–152, 2011.

[21] F. Martinelli and P. Mori. On Usage Control for GRID Systems. *Future Generation Computer Systems*, 26(7):1032–1042, July 2010.

[22] F. Martinelli, P. Mori, and A. Vaccarelli. Towards Continuous Usage Control on Grid Computational Services. In *Joint Intl. Conference on Autonomic and Autonomous Systems and Intl. Conference on Networking and Services*, 2005.

[23] D. Meekins. UDP Tunnel - Tunnel TCP data through UDP messages, http://code.google.com/p/udptunnel/, 2009-2011.

[24] Microsoft Corporation. Architecture of Windows Media Rights Manager, http://www.microsoft.com/windows/windowsmedia/howto/articles/drmarchitecture.aspx, 2004.

[25] R. Neisse, D. Holling, and A. Pretschner. Implementing Trust in Cloud Infrastructures. *Proc. 11th IEEE/ACM International Conference on Cluster Cloud and Grid Computing*, 2011.

[26] J. Park and R. Sandhu. Towards Usage Control Models: Beyond Traditional Access Control. In *Proc. 7th ACM Symposium on Access Control Models and Technologies*, pages 57–64, 2002.

[27] D. Povey. Optimistic Security: A New Access Control Paradigm. In *Proc. Workshop on New Security Paradigms*, pages 40–45, 1999.

[28] A. Pretschner. An Overview of Distributed Usage Control. In *Knowledge Engineering: Principles and Techniques Conference*, pages 25–33, 2009.

[29] A. Pretschner, M. Büchler, M. Harvan, C. Schaefer, and T. Walter. Usage Control Enforcement with Data Flow Tracking for X11. In *Proc. 5th Intl. Workshop on Security and Trust Management*, pages 124–137, 2009.

[30] A. Pretschner, M. Hilty, and D. Basin. Distributed Usage Control. *Communications of the ACM*, 49(9):39–44, 2006.

[31] A. Pretschner, M. Hilty, D. Basin, C. Schaefer, and T. Walter. Mechanisms for Usage Control. In *Proc. 2008 ACM Symposium on Information, Computer and Communications Security*, pages 240–244, 2008.

[32] A. Pretschner, M. Hilty, F. Schütz, C. Schaefer, and T. Walter. Usage Control Enforcement: Present and Future. *IEEE Security & Privacy*, 6(4):44–53, 2008.

[33] A. Pretschner, E. Lovat, and M. Büchler. Representation-Independent Data Usage Control. In *Data Privacy Management and Autonomous Spontaneus Security*, volume 7122 of *Lecture Notes in Computer Science*, pages 122–140. 2012.

[34] A. Pretschner, J. Rüesch, C. Schaefer, and T. Walter. Formal Analyses of Usage Control Policies. *Proc. 2009 International Conference on Availability, Reliability and Security*, pages 98–105, 2009.

[35] A. Pretschner, F. Schütz, C. Schaefer, and T. Walter. Policy Evolution in Distributed Usage Control. *Electron. Notes Theor. Comput. Sci.*, 244:109–123, Aug. 2009.

[36] N. Provos. Improving Host Security with System Call Policies. In *Proc. 12th USENIX Security Symp.*, 2003.

[37] R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn. Attestation-based Policy Enforcement for Remote Access. In *Proc. 11th ACM Conference on Computer and Communications Security*, pages 308–317, 2004.

[38] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[39] R. Sandhu and X. Zhang. Peer-to-peer Access Control Architecture Using Trusted Computing Technology. In *Proc. 10th ACM Symposium on Access Control Models and Technologies*, pages 147–158, 2005.

[40] O. W. Seneviratne. Augmenting the Web with Accountability. In *Proc. 21st Intl. Conf. Companion on World Wide Web*, pages 185–190, 2012.

[41] S. Subramanya and B. K. Yi. Digital Rights Management. *IEEE Potentials*, (April):31–34, 2006.

[42] TrueCrypt Foundation. TrueCrypt, http://www.truecrypt.org/, 2004-2012.

[43] D. J. Weitzner, H. Abelson, T. Berners-Lee, J. Feigenbaum, J. Hendler, and G. J. Sussman. Information Accountability. *Commun. ACM*, 51(6):82–87, June 2008.

[44] T. Wüchner and A. Pretschner. Data Loss Prevention Based on Data-Driven Usage Control. In *Proc. 23rd IEEE Intl. Symposium on Software Reliability Engineering*, Nov. 2012.

[45] X. Zhang, J.-P. Seifert, and R. Sandhu. Security Enforcement Model for Distributed Usage Control. *IEEE Intl. Conf. on Sensor Networks, Ubiquitous, and Trustworthy Computing*, pages 10–18, 2008.