Data Usage Control for Distributed Systems

FLORIAN KELBERT, Imperial College London, United Kingdom

ALEXANDER PRETSCHNER, Technical University of Munich, Germany

Data usage control enables data owners to enforce policies over how their data may be used after it has been released and accessed. We address distributed aspects of this problem, which arise if the protected data resides within multiple systems. We contribute by formalizing, implementing, and evaluating a fully decentralized system that (i) generically and transparently tracks protected data across systems, (ii) propagates data usage policies along, and (iii) efficiently and preventively enforces policies in a decentralized manner. The evaluation shows that (i) data flow tracking and policy propagation achieve a throughput of 21%–54% of native execution, and (ii) decentralized policy enforcement outperforms a centralized approach in many situations.

1 INTRODUCTION

Due to the ever increasing value of data, its continuous protection becomes ever more important. Corresponding solutions are applicable in many contexts, such as business, military and government secrets, as well as privacy and copyright protection. In all these cases, the owners of valuable data would like to constrain its future usage.

Data usage control solutions [Park and Sandhu 2004; Pretschner et al. 2006] allow for the specification and enforcement of policies expressing propositional, temporal, cardinal, and spatial constraints on future data usages. Different to traditional access control, these policies are enforced even *after* access was granted.

Reference monitors ensure, or observe, policy compliance by continuously intercepting and monitoring relevant events. Because data constitutes an abstraction which materializes in different forms, such as files or database entries, reference monitors must: (i) record the flow of data through the system; (ii) enforce, or observe, policy compliance for *all* of the recorded data representations [Pretschner et al. 2012]. In the following, policy enforcement refers to both detective and preventive enforcement.

Existing approaches fall short in the presence of distributed data sharing and processing, in which case policies may refer to the storage and processing of data *across* systems, such as "this data must not reside in more than three

Authors' addresses: Florian Kelbert, Imperial College London, 180 Queen's Gate, London, SW7 2AZ, United Kingdom, fkelbert@acm.org; Alexander Pretschner, Technical University of Munich, Boltzmannstrasse 3, Garching bei München, 85748, Germany, alexander.pretschner@tum.de.

systems", or "access this data at most five times". Hence, policies must be enforced on *all* systems that store and process the sensitive data. We investigate two related research questions (RQs):

First, for controlling data usage in distributed systems, it is essential to know all representations of protected data across all systems. Hence, whenever data is exchanged between systems, its flow must be recorded and its policies must be made available.

While there exist solutions that track usage controlled data and corresponding policies across systems [Kumari et al. 2011; Lazouski et al. 2014; Papagiannis and Pietzuch 2012], they rely on particular applications, protocols, or file types. We deem such tailoring inadequate, as data is continuously transformed and exchanged by a multitude of applications and protocols. Further solutions exist (see §7), but they (i) limit the number of trackable data and/or policies, (ii) propagate labels rather than complex policies, (iii) rely on particular hardware, hypervisors, or application(-protocol)s, or (iv) necessitate the adaptation of existing applications.

Ideally, however, tracking of data flows across systems should be generic and transparent to applications and operating systems. We thus tackle the research question

RQ1: How can the flow of data across systems be generically and

transparently tracked and how can data usage policies be propagated?

Second, policies referring to distributed data representations and events thereon must be enforced. Information about data usage must thus be available to the corresponding decision points. While policies could be enforced by a centralized infrastructure, this poses a single point of failure, privacy concerns, high hypothesized communication and performance overheads, and availability concerns.

We thus aim for the deployment of distributed monitors, each of them (i) observing local system behavior, (ii) taking policy decisions, and (iii) communicating essential information to remote monitors. While several solutions aiming at such decentralization have been proposed (see §7), they (i) present central components, (ii) do not cater to data being propagated across systems, (iii) do not enforce policies preventively, or (iv) are incompatible with commodity systems. Hence, we address the research question

RQ2: How can usage control policies be enforced effectively, preventively,

and decentrally if data, events thereon, and policies are distributed?

Solution. Our solution is a generic model and implementation for cross-system data flow tracking, policy propagation, and distributed policy enforcement. Cross-system data flow tracking and policy propagation are transparent and generic, i.e., independent of applications, application-protocols, and the operating system (*RQ1*). Global data usage policies are enforced in a fully decentralized manner: distributed decisions are decentrally, continuously, and consistently taken, agreed upon, and enforced (*RQ2*).

Contributions. Our contributions are:

- A generic and formal model that allows for the explicit distinction of different systems, their individual behaviors, as well as their interplay. In the usage control context, this is the first operationalized model that allows to reason about both independent systems as well as the distributed system they form.
- The first model and implementation that transparently and generically tracks data flows and policies across systems.
- The first model and implementation for the fully decentralized and preventive enforcement of usage control policies that refer to data or events that are distributed.

While some of those results were published earlier [Kelbert and Pretschner 2013; 2014; 2015], original contributions of the article at hand are:

Manuscript submitted to ACM

- The consolidation, integration, and consistent presentation of the models and formalisms published in [Kelbert and Pretschner 2013; 2014] (§3).
- A communication-protocol independent approach to track data flows across systems (§3.2), which is more generic than earlier work [Kelbert and Pretschner 2013].
- A formalization of (i) how the distributed system's data flow state is composed of the individual systems' data flow states (§3.1.3), and (ii) the cross-system data flow semantics of system calls *sendto* and *recvfrom* (§3.2.2).
- The detailed description of an architecture and implementation that realizes the proposed concepts (§4). Different to [Kelbert and Pretschner 2015], we describe the infrastructure up to a level of detail that is consistent with the formal models.
- Novel evaluation results (§5): Preventive cross-system data flow tracking and policy propagation achieves 21%–54% performance of native execution (§5.2); Decentralized policy enforcement outperforms a centralized infrastructure in many situations (§5.3).

Threat Model. We want to prevent or detect data usage that is not compliant with the corresponding policies. Considered attacker models are: (i) End users without administrative privileges who act intentionally or unintentionally. (ii) A Man-in-the-Middle eavesdropping both the system's actual communication as well as our infrastructure's. (iii) System administrators accessing and controlling all or most aspects of the system. (iv) Malicious software, such as malware or misconfigured or corrupted benign software.

Example. Consider an insurance company that provides customers the ability to access services via a web interface, e.g., to request contract offers and issue damage reports. Internally, the insurance company performs market research and data analysis to produce valuable reports and predictions. Both the customers' personal records and the insurance company's internal data require protection from misuse and leakage. All of these data flow through different systems in various formats and is viewed, stored, and processed by several users. Some data is derived from other data, e.g., a contract being created from the customer's personal data. All these different distributed representations of the same data ought to be protected by policies such as:

Policy 1: 'Exactly one contract offer must be sent to the customer not later than 30 days after a request for a contract offer has been received.'

Policy 2: 'If the customer declines a contract offer, then all associated and derived data must not be used anymore.'

Policy 3: 'Each contract must be reviewed and approved by at least two clerks.'

Policy 4: 'At each point in time no two clerks might have a local copy of the same customer record.'

As policy 1 shows, usage control can not only enforce constraints on data usage, but also oblige certain events to improve service. Note that all of the above policies are *global policies*, meaning that they refer to data and events that are distributed.

While a centralized infrastructure can enforce these policies, a decentralized approach is more efficient. For example, consider policy 2: In a centralized approach, *every* data usage from *all* clients must be signaled to the central infrastructure for decision making. If, however, individual clients are aware of decision logics, then most policy decisions can be taken locally, as clients are aware whether they are handling such derived data.

2 AN EXISTING USAGE CONTROL MODEL

This chapter describes previous work upon which this article builds, i.e. a formal data usage control model (§2.1) and a corresponding enforcement infrastructure (§2.2).

2.1 Formal Data Usage Control Model

The model we leverage [Harvan and Pretschner 2009; Hilty et al. 2007; Pretschner et al. 2008, 2012] defines usage control policies (§2.1.3) as constraints over system traces (§2.1.1) and system states (§2.1.2) as follows.

2.1.1 System Events and System Traces. Events $\mathcal{E}$ are defined by a name (set $\mathcal{N}$) and a set of parameters, which are, in turn, defined by a name (set $\mathcal{N}$) and a value (set $\mathcal{V}$): $\mathcal{E} \subseteq \mathcal{N} \times \mathbb{P}(\mathcal{N} \times \mathcal{V})$. For an event $e \in \mathcal{E}$, let $e.n$ denote its name and $e.p$ its set of parameters. We write $(e.n, \{(p_1, v_1), \ldots, (p_m, v_m)\})$ for the event $e \in \mathcal{E}$ with $m \in \mathbb{N}$ parameters $(p_1, v_1), \ldots, (p_m, v_m) \in \mathcal{N} \times \mathcal{V}$.

Each event carries two mandatory parameters, $obj, actual \in \mathcal{N}$, which we refer to as $e.obj$ and $e.actual$: $e.obj$ denotes the primary object of e, such as a file; $e.actual$ is boolean: $e.actual = true$ indicates that event e has already happened, while $e.actual = false$ indicates that event e is intended. Hence, we obtain two subsets of $\mathcal{E}$, *actual events* $\mathcal{E}^A \stackrel{\text{def}}{=} \{e \in \mathcal{E} \mid e.actual\}$ and *intended events* $\mathcal{E}^I \stackrel{\text{def}}{=} \mathcal{E} \setminus \mathcal{E}^A$. As an example event, consider a system call that is initiated by an application and executed by the OS kernel.

Events might be *data usage events* $\mathcal{E}_U$ or *data flow events* $\mathcal{E}_F$. The former process data, while the latter cause data to migrate between representations. There might exist events that belong to both sets at the same time, hence $\mathcal{E}_U \cap \mathcal{E}_F \neq \emptyset$.

For example, event $e_r = (review, \{(obj, d), (actual, true), (role, clerk)\}) \in \mathcal{E}$ denotes that a user with role *clerk* actually reviews data d, hence $e_r \in \mathcal{E}^A$. Intuitively, reviewing is considered data usage, hence $e_r \in \mathcal{E}^A \cap \mathcal{E}_U$; e_r also is a data flow event if the implementation is such that e_r leads to the creation of a new representation of d.

Event refinement. When specifying policies (§2.1.3), it is useful to specify only relevant event parameters, quantifying over unmentioned ones. Hence, $ref \subseteq \mathcal{E} \times \mathcal{E}$ defines a refinement relation: event e_1 refines event e_2 iff their names are the same and iff the parameters of e_1 are a superset of those of e_2:

$$\forall e_1, e_2 \in \mathcal{E} : e_1\ ref\ e_2 \stackrel{\text{def}}{\iff} e_1.n = e_2.n \wedge e_1.p \supseteq e_2.p$$

System events $\mathcal{S}$ are events that are observed at runtime. They are maximally refined, i.e. all parameters are determined: $\mathcal{S} \stackrel{\text{def}}{=} \{e \in \mathcal{E} \mid \nexists e' \in \mathcal{E} : e' \neq e \wedge e'\ ref\ e\}$. We categorize system events $\mathcal{S}$ into actual system events $\mathcal{S}^A \stackrel{\text{def}}{=} \mathcal{E}^A \cap \mathcal{S}$, intended system events $\mathcal{S}^I \stackrel{\text{def}}{=} \mathcal{E}^I \cap \mathcal{S}$, data usage system events $\mathcal{S}_U \stackrel{\text{def}}{=} \mathcal{E}_U \cap \mathcal{S}$, and data flow system events $\mathcal{S}_F \stackrel{\text{def}}{=} \mathcal{E}_F \cap \mathcal{S}$. We further define $\mathcal{S}_F^A \stackrel{\text{def}}{=} \mathcal{S}^A \cap \mathcal{S}_F$. Parameter $time \in \mathcal{N}$ indicates the event's time of observation: $\forall e \in \mathcal{S} \exists r \in \mathbb{R}^{\geq 0} : (time, r) \in e.p$. We access r by $e.time$.

Traces $\mathcal{T}$ model system runs by mapping abstract points in time $i \in \mathbb{N}$ to all system events that happened since $i - 1$:

$$\mathcal{T} \colon \mathbb{N} \to \mathbb{P}(\mathcal{S}),\ \text{such that } \forall t \in \mathcal{T}, \forall i \in \mathbb{N}, i > 0, \forall e \in t(i) : i - 1 < e.time \leq i$$

We assume that no two events happen at exactly the same point in time:

$$\forall t \in \mathcal{T}, i \in \mathbb{N}, e_1, e_2 \in t(i) : e_1.time = e_2.time \implies e_1 = e_2.$$

2.1.2 Generic Data Flow Model and System States. At runtime the data to be protected exists in multiple, continuously evolving, representations. The system's *data flow state* captures which data takes which representations at which point in time. A data flow model [Harvan and Pretschner 2009; Pretschner et al. 2012] (over-)approximates these data representations. Data flow events $\mathcal{S}_F$ initiate state transitions.

Within this model, $\mathcal{D}$ is the set of *data* to be protected. Representations containing data are called *containers*, denoted by C, and $\mathcal{D} \cap C = \emptyset$. $\mathcal{D}$, C and special value *nil* constitute possible values for event parameter obj: $\forall e \in \mathcal{E}, \exists v \in$

$\mathcal{D} \cup C \cup \{nil\} : e.obj = v$. *Identifiers* $\mathcal{I}$ identify containers. In practice, system events carry parameters that allow to identify the container on which the event operates.

The set of all possible *data flow states* is defined as $\Sigma \overset{\text{def}}{=} s \times a \times n$. Each state consists of three mappings: (1) A *storage function* $s : C \to \mathbb{P}(\mathcal{D})$ capturing which containers potentially store which data. (2) An *alias function* $a : C \to \mathbb{P}(C)$ capturing that some containers may implicitly get updated whenever other containers do: If $c_2 \in a(c_1)$ for $c_1, c_2 \in C$, then any data written into c_1 is immediately propagated to c_2. (3) A *naming function* $n : \mathcal{I} \to C$ mapping identifiers to containers. For a state $\sigma \in \Sigma$, $\sigma.s$, $\sigma.a$, and $\sigma.n$ refer to those mappings. The *initial state* of trace $t \in \mathcal{T}$ is denoted σ_t^0.

Transition relation $\mathcal{R} \subseteq \Sigma \times \mathcal{S}_F^A \times \Sigma$ defines how the above state changes in correspondence with the execution of actual data flow system events $\mathcal{S}_F^A$. Given a state $\sigma \in \Sigma$ and a set of events $S \in \{S' \subseteq \mathcal{S}_F^A \mid \forall e, e' \in S' : e.time = e'.time \implies e = e'\}$, function $\widetilde{\mathcal{R}} : \Sigma \times \mathbb{P}(\mathcal{S}_F^A) \to \Sigma$ updates the data flow state:

$$\forall \sigma \in \Sigma, S \in \{S' \subseteq \mathcal{S}_F^A \mid \forall e, e' \in S' : e.time = e'.time \implies e = e'\} :$$

$$\widetilde{\mathcal{R}}(\sigma, S) \overset{\text{def}}{=} \begin{cases} \sigma & \text{if } S = \emptyset \\ \widetilde{\mathcal{R}}(\mathcal{R}(\sigma, e), S \setminus \{e\}) & \text{for } e \in S : \nexists e' \in S : e'.time < e.time \end{cases}$$

For trace $t \in \mathcal{T}$ and timestep $i \in \mathbb{N}$, the set of system events causing data flows is $t(i) \cap \mathcal{S}_F^A$. Thus, the system's state at the end of timestep $i \in \mathbb{N}$, $i > 0$, is computed as $\sigma_t^i \overset{\text{def}}{=} \widetilde{\mathcal{R}}(\sigma_t^{i-1}, t(i) \cap \mathcal{S}_F^A)$.

Instantiations of this model, including semantics of $\mathcal{R}$, exist, among others, for Unix [Harvan and Pretschner 2009], X11 [Pretschner et al. 2009], and Microsoft Windows [Wüchner and Pretschner 2012]. Upon their application at runtime, storage function $\sigma.s$ reflects at each point in time which data resides in which containers.

In the insurance company example, $\mathcal{D}$ represents the set of all customer data, contracts, and the insurance's business secrets. Containers C are, e.g., workstations, emails, files, database records, and data processing software. This clear distinction between abstract data ($\mathcal{D}$) and their concrete technical representations (C) allows for flexible and generic data flow tracking by means of updating their mapping ($\sigma.s$) in correspondence with observed system events ($\mathcal{S}$).[1] §3.2 instantiates this model for processes, files, and sockets (C) and system calls ($\mathcal{S}$). Further, state transitions ($\mathcal{R}$) for system calls are defined to model how system calls propagate data between containers, thereby modifying $\sigma.s$.

Relation $ref_\Sigma \subseteq (\mathcal{S} \times \Sigma) \times \mathcal{E}$ describes event refinement in the presence of a given state. The rationale is that system events $\mathcal{S}$ *always* operate on containers, while policies (§2.1.3) might be specified in terms of data. Hence, state $\sigma \in \Sigma$ determines whether an event refines another: (e_1, σ) refines e_2 if either both e_1 and e_2 operate on the same container and if e_1 *ref* e_2, or if e_1 operates on some container $c \in C$ and e_2 operates on some data $d \in \mathcal{D}$ within c ($d \in \sigma.s(c)$) and e_1 *ref* e_2 when ignoring parameter *obj*:

$$\forall e_1 \in \mathcal{S}, e_2 \in \mathcal{E}, \sigma \in \Sigma : (e_1, \sigma) \ ref_\Sigma \ e_2 \overset{\text{def}}{\iff} \exists c \in C : e_1.obj = c \land e_2.obj = c \land e_1 \ ref \ e_2$$

$$\lor \exists c \in C, d \in \mathcal{D} : e_1.n = e_2.n \land e_1.obj = c \land e_2.obj = d$$

$$\land d \in \sigma.s(c) \land e_1.p \setminus \{(obj, c)\} \supseteq e_2.p \setminus \{(obj, d)\}$$

On this basis, §2.1.3 defines the syntax and semantics of data usage policies.

---

[1]While conservatively tracking flows, this model leads to overapproximations of the data analysis. This problem has been tackled elsewhere in terms of increasing precision by considering multiple abstraction layers [Lovat et al. 2016], data structure [Lovat and Kelbert 2014], and data quantities [Lovat et al. 2014].

2.1.3 Specification of Data Usage Policies. We specify policies using the Obligation Specification Language (OSL), an extension of linear temporal logics [Hilty et al. 2007; Pretschner et al. 2008, 2012]. Previous work by Kumari and Pretschner [2013] showed how OSL policies can be translated into **E**vent-**C**ondition-**A**ction (ECA) rules for enforcement purposes. Hence we focus on the latter. Their semantics are as follows: If a system event $e' \in \mathcal{S}$ refining the *trigger* **E**vent is observed at timestep i and if the execution of this event would make the **C**ondition true, then additional **A**ctions might be performed at timestep $i+1$. These include allowance, delaying, and inhibition of event e', as well as execution of additional events. ECA conditions Φ are specified in terms of past linear temporal logics. [Lichtenstein et al. 1985]. Their syntax is:

$\Psi \overset{\text{def}}{=} \underline{\mathit{true}} \mid \underline{\mathit{false}} \mid \mathcal{E}$

$\Omega \overset{\text{def}}{=} \underline{\mathit{notIn}}(\mathcal{D}, \mathbb{P}(C)) \mid \underline{\mathit{combined}}(\mathcal{D}, \mathcal{D}, \mathbb{P}(C)) \mid \underline{\mathit{maxIn}}(\mathcal{D}, \mathbb{N}, \mathbb{P}(C))$

$\Phi \overset{\text{def}}{=} (\Phi) \mid \Psi \mid \Omega \mid \underline{\mathit{not}}(\Phi) \mid \Phi \underline{\ \mathit{and}\ } \Phi \mid \Phi \underline{\ \mathit{or}\ } \Phi \mid \Phi \underline{\ \mathit{since}\ } \Phi \mid \Phi \underline{\ \mathit{before}\ } \mathbb{N} \mid \underline{\mathit{repmin}}(\mathbb{N}, \mathbb{N}, \mathcal{E}) \mid \underline{\mathit{repmax}}(\mathbb{N}, \mathbb{N}, \mathcal{E}) \mid \underline{\mathit{always}}(\Phi) \mid \underline{\mathit{evalExt}}(\Gamma)$

$\Gamma \overset{\text{def}}{=} \Psi \mid \mathbb{N} \mid \mathit{op} \mid \mathit{String} \mid \ldots$

For brevity, we omit the description of the formal semantics which are provided in [Kelbert and Pretschner 2014; Pretschner et al. 2012]. Intuitively they are as follows. Ψ refers to constants *true*, *false*, and events $\mathcal{E}$. An event operator $e \in \mathcal{E}$ evaluates to true iff, given the current data flow state, an event refining e happens in the current timestep (see ref_Σ). Ω defines *state-based operators* that constrain the data flow state: $\underline{\mathit{notIn}}(d, C)$ is true iff data d is not in any of the containers C; $\underline{\mathit{combined}}(d_1, d_2, C)$ is true iff there exists at least one container in C that contains both data d_1 and d_2; $\underline{\mathit{maxIn}}(d, m, C)$ is true iff data d is contained in at most m containers in C. Given a state $\sigma \in \Sigma$, operators Ω thus constrain which data items $\mathcal{D}$ must or must not be contained in particular containers C, i.e., they constrain the set of 'allowed' states σ. Φ defines *propositional*, *temporal*, and *cardinality operators*. The semantics of $\underline{\mathit{not}}$, $\underline{\mathit{and}}$ and $\underline{\mathit{or}}$ are intuitive; $\alpha \underline{\ \mathit{since}\ } \beta$ is true iff β was true some time earlier and α was true ever since, or if α was always true; $\alpha \underline{\ \mathit{before}\ } j$ is true iff α was true exactly j timesteps ago; $\underline{\mathit{repmin}}(j, m, e)$ is true iff event e happened at least m times in the last j timesteps. Further, $\underline{\mathit{repmax}}(j, m, e) \equiv \underline{\mathit{not}}(\underline{\mathit{repmin}}(j, m+1, e))$, and $\underline{\mathit{always}}(\alpha) \equiv \alpha \underline{\ \mathit{since}\ } \underline{\mathit{false}}$. Operator $\underline{\mathit{evalExt}}$ allows to incorporate external specification and evaluation logics, e.g. to specify conditions that refer to subject and object attributes or environmental conditions, such as (absolute) time and location. We leave the semantics of $\underline{\mathit{evalExt}}$ and its parameters Γ (including operations op) unspecified. Earlier implementations leveraged XPath for such purposes [Feth and Pretschner 2012; Wüchner and Pretschner 2012].

For trace $t \in \mathcal{T}$, timestep $i \in \mathbb{N}$, and condition $\varphi \in \Phi$, notation $(t, i) \models \varphi$ denotes that trace t satisfies φ at time i.

Table 1 expresses the example policies from §1 as ECA rules. Rule 1a notifies the manager about data d if 30 days ago there was a contract offer request involving d (($\mathit{requestOffer}$, $\{(\mathit{obj}, d)\}$) $\underline{\mathit{before}}$ 30) and if no contract offer involving d was sent meanwhile ($\underline{\mathit{repmax}}(30, 0, (\mathit{sendOffer}, \{(\mathit{obj}, d)\}))$). Rule 1b inhibits sending of a contract offer involving data d if there was no corresponding contract request in the last 30 days ($\underline{\mathit{repmax}}(30, 0, (\mathit{requestOffer}, \{(\mathit{obj}, d)\}))$) or if a contract offer was already sent ($\underline{\mathit{repmin}}(30, 1, (\mathit{sendOffer}, \{(\mathit{obj}, d)\}))$). Rule 2 inhibits any usage of customer data d that was part of a declined contract offer at some earlier point in time ($\underline{\mathit{not}}(\underline{\mathit{always}}(\underline{\mathit{not}}((\mathit{declineOffer}, \{(\mathit{obj}, d)\}))))$). Rule 3 inhibits sending of contract d if it was not reviewed ($\underline{\mathit{repmax}}(30, 1, (\mathit{review}, \{(\mathit{obj}, d)\}))$) or approved ($\underline{\mathit{repmax}}(30, 1, (\mathit{approve}, \{(\mathit{obj}, d)\}))$) more than once. Rule 4 inhibits events that lead to a state in which customer data d is in more than one of the clerk's workstations.

The above policies constitute templates: To protect a certain data item, the corresponding policies must be instantiated and deployed for this particular data item.

Policy 1	**E**vent: $< any >$		
(a)	**C**ondition: $((requestOffer, \{(obj, d)\})$ *before* $30)$ *and* *repmax*$(30, 0, (sendOffer, \{(obj, d)\}))$		
	Action: $(notifyManager, \{(obj, d)\})$		
	Event: $(sendOffer, \{(obj, d)\})$		
(b)	**C**ondition: *repmax*$(30, 0, (requestOffer, \{(obj, d)\}))$ *or* *repmin*$(30, 1, (sendOffer, \{(obj, d)\}))$		
	Action: *inhibit*		
Policy 2	**E**vent: $(use, \{(obj, d)\})$		
	Condition: *not*$(always(not((declineOffer, \{(obj, d)\}))))$		
	Action: *inhibit*		
Policy 3	**E**vent: $(sendOffer, \{(obj, d)\})$		
	Condition: *repmax*$(30, 1, (review, \{(obj, d)\}))$ *or* *repmax*$(30, 1, (approve, \{(obj, d)\}))$		
	Action: *inhibit*		
Policy 4	**E**vent: $(< any >, \{(obj, d)\})$		
	Condition: *not*$(maxIn(1, d, C_{Workstation}))$		
	Action: *inhibit*		

Table 1. Example policies from §1 as ECA rules.

2.2 Enforcement Infrastructure

To enforce the above concepts, technical infrastructures (i) monitor system events $\mathcal{S}$, (ii) track the flow of data on the basis of $\mathcal{S}_F$, $\mathcal{R}$, and Σ, (iii) decide whether data usage system events $\mathcal{S}_U$ ought to be allowed, modified, inhibited, delayed, or whether compensating actions need to be taken, (iv) enforce the decisions [Hilty et al. 2007; Kumari et al. 2011; Neisse et al. 2011b; Pretschner et al. 2009, 2012].

2.2.1 Architecture Overview. All of the cited infrastructures are implemented along the following lines: Initially, the Policy Management Point (PMP) deploys the ECA rules to be enforced at the Policy Decision Point (PDP). System-layer specific Policy Enforcement Points (PEPs) intercept system events $\mathcal{S}$, temporarily block them from execution, and signal them to the PDP. The PDP first forwards these events to the Policy Information Point (PIP), which maintains the system's data flow state $\sigma \in \Sigma$ in correspondence with the signaled data flow system events. The PDP then evaluates the signaled event against all deployed ECA rules, possibly involving the PIP's state. Finally, the decision is sent back to the PEP, which enforces it and continues execution of the original event. Because PEPs are stateless, *every* such event must be evaluated by the PDP.

2.2.2 Policy Decision Point. The PDP must ensure compliance with any deployed ECA rules. The PDP thus continuously evaluates them, considering conditions $\varphi_p \in \Phi$ as *expression trees* as depicted in Fig. 1: Leaves represent constants *true* and *false*, events $\mathcal{E}$, state-based operators Ω, and *evalExt*. Internal nodes represent operators such as *not*, *and*, *before*, *since*, and *repmin*. The tree's nodes are stateful, storing whether the operator's state changed during the current or previous timesteps.

Now, consider an ECA rule with condition $\varphi_p \in \Phi$ to be deployed at the PDP and that some PEP signals a system event $e \in \mathcal{S}$. The PDP first forwards e to the PIP, which updates the data flow state as detailed in §2.2.3. The subsequent policy evaluation thus grounds on an up-to-date data flow state.
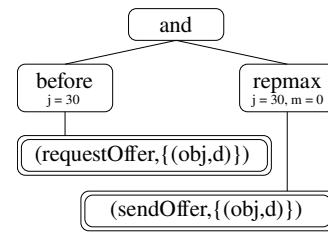

Fig. 1. Expression tree of the condition of ECA rule 1a.

If e is a data usage event, $e \in \mathcal{S}_U$, then the PDP first updates the states of all *leaf nodes* of the expression tree representing φ_p. The states of operators $\underline{\textit{true}}$ and $\underline{\textit{false}}$ are invariant. For a leaf representing event $e' \in \mathcal{E}$, the leaf's state is a counter indicating how many events refining e' happened within the current timestep. This counter is incremented if $(e, \sigma) \textit{ ref}_{\Sigma} \ e'$, where $\sigma \in \Sigma$ denotes the current data flow state. For leaf nodes representing state-based operators Ω, this update process is delegated to the PIP. For nodes of type $\underline{\textit{evalExt}}$, the update process is delegated to the corresponding external framework. In sum, the tree's leaves track how often events happened and whether state-based operators or operators of type $\underline{\textit{evalExt}}$ changed their state.

After updating the leaf nodes, evaluation of the entire condition φ_p may be triggered for two reasons: (1) If the signaled event e refines the trigger event e_p, i.e. $(e, \sigma) \textit{ ref}_{\Sigma} \ e_p$, then it must be evaluated whether e complies with the policy; (2) If a timestep has passed, then it must be evaluated whether all temporal constraints are satisfied. In both cases, the expression tree of φ_p is evaluated recursively from the root. We denote the evaluation result by $\textit{eval}(\varphi_p)$. It reflects $(t, i) \models \varphi_p$ with $t \in \mathcal{T}$ being the executing trace and $i \in \mathbb{N}$ the current point in time. If $\textit{eval}(\varphi_p) = \textit{true}$ then p's actions a_p are applied. Otherwise, the signaled event e is allowed.

Recursive evaluation of operators $\underline{\textit{not}}$, $\underline{\textit{and}}$, and $\underline{\textit{or}}$ is trivial: the corresponding nodes are stateless and their result is computed by combining the results of their child nodes. We explain the evaluation of selected complex operators in the following.

For $e' \in \mathcal{E}$, $\textit{eval}(e')$ returns *true* if at least one event refining e' happened within the current timestep. The state is reset if evaluation takes place at the end of a timestep.

For $\varphi = \alpha \ \underline{\textit{before}} \ j$, an array stores the evaluation results of α of the last j timesteps. The result of $\textit{eval}(\varphi)$ corresponds to the value that was stored j timesteps ago. When evaluating at the end of a timestep, the result of $\textit{eval}(\alpha)$ overwrites the oldest entry.

For $\varphi = \underline{\textit{repmin}}(j, m, e')$, an array stores the number of refinements of e' for the last j timesteps. $\textit{eval}(\varphi)$ returns *true* if the sum of those values is greater than or equal to m.

Note: If the PEP signaled an intended event, then the changes made to φ_p's expression tree's nodes' states, as well as all changes made to the PIP's data flow state are rolled back after the above evaluation of φ_p. This is because the PDP and PIP simulate how the allowance of the intended event *would* change their states. However, since the event was *not* actual, any such changes have not happened in the real system.

2.2.3 Policy Information Point.

2.2.3 Policy Information Point. The PIP maintains the system's data flow state and evaluates state-based operators. Whenever the PDP signals a system event $e \in \mathcal{S}$, the PIP evaluates whether there exists a corresponding state transition, i.e. whether $\exists \sigma, \sigma' \in \Sigma, e' \in \mathcal{S}_F : (\sigma, e', \sigma') \in \mathcal{R} \wedge (e, \sigma) \textit{ ref}_{\Sigma} \ e'$. If so, then the data flow state is updated according to the semantics defined by $(\sigma, e', \sigma') \in \mathcal{R}$.

To evaluate event refinement $\textit{ref}_{\Sigma}$, the PDP can query the PIP for all data contained in a specific container c, i.e. $\sigma.s(c)$. Using this result set, the PDP is able to evaluate whether $d \in \sigma.s(c)$ for any $d \in \mathcal{D}$, and consequently whether $(e_1, \sigma) \textit{ ref}_{\Sigma} \ e_2$.

The PDP delegates evaluation of state-based operators Ω to the PIP. The PIP evaluates those on the basis of their semantics (see §2.1.3) and the current data flow state using simple set operations. The PDP uses the result to update the state-based operator's state within the expression tree of the condition being evaluated.

3 DISTRIBUTED DATA USAGE CONTROL

The model and infrastructure from §2 are monolithic, as they refer to one single trace and system state: One single PDP/PIP regulates the execution of all system events. This necessitates communication with the central PDP/PIP for every observed system event. However, such a centralized approach is expected to be impractical for distributed system setups [Basin et al. 2015; Janicke et al. 2008]: Vital central components might not be able to scale and communication delays might significantly impact performance.

Example. We revisit policy 2 from the insurance company example. Assume that the company set up a centralized PDP and that the customer's data is used and stored on multiple systems, e.g. the web, email, and data analysis server. Since those systems' PEPs are stateless, they signal *every* event to the PDP—even if (i) the event does not operate on any customer data, and if (ii) the policy does not apply for other reasons, e.g. because the customer has *not* declined the offer. If, instead, *local* PDPs could decide whether events comply with the policy, then no communication would be required.

This section presents a model that deploys local PDPs close to PEPs. The local PDPs retain enough information to take many decisions independently and conclusively, thus avoiding communication with a central PDP. We hypothesize that taking policy decisions locally decreases the overall communication and performance overheads—even though synchronization between the distributed PDPs becomes necessary.

We formalize a distributed system model (§3.1), on the basis of which we track data flows across systems (§3.2, *RQ1*) and decentrally enforce global policies (§3.3, *RQ2*).

3.1 Distributed System Model

We extend the model from §2.1 by modeling different systems, their individual behaviors, and their interplay. We refer to the monolithic system from §2 as the *distributed system*, in which multiple PDPs/PIPs observe disjoint parts. We formalize how these individual decentral observations reassemble the observations of one single central PDP/PIP. This allows us to use the policy language from §2.1.3 within a distributed environment.

3.1.1 Individual Systems. Systems $\mathcal{Y}$ constitute autonomous parts of the distributed system. We define a *system* to be a non-empty set of system layers whose PEPs share the same PDP and PIP. Systems communicate via the network and may in themselves be distributed, as PEPs sharing the same PDP/PIP may reside on different hosts.

For any system $y \in \mathcal{Y}$, $\mathcal{S}_y \subseteq \mathcal{S}$ denotes its unique set of system events, $C_y \subseteq C$ its unique set of containers, $\mathcal{I}_y \subseteq \mathcal{I}$ its unique set of identifiers, $\mathcal{T}_y \subseteq \mathcal{T}$ its set of all possible system runs, and $\Sigma_y \subseteq \Sigma$ its set of all possible data flow states:

$$\forall y \in \mathcal{Y} : \mathcal{T}_y \subseteq \mathcal{T} : \mathbb{N} \to \mathbb{P}(\mathcal{S}_y) \text{ and } \Sigma_y \subseteq \Sigma : (C_y \to \mathbb{P}(\mathcal{D})) \times (C_y \to \mathbb{P}(C_y)) \times (\mathcal{I}_y \to C_y)$$

Each system event $e \in \mathcal{S}_y$ is required to carry parameter $sys \in \mathcal{N}$ with value $y \in \mathcal{Y}$. State transitions $\mathcal{R}$, specifying how data flows between systems, are defined in §3.2.2.

In practice, individual systems run in parallel and produce independent system traces and data flow states: The PDP of system $y \in \mathcal{Y}$ (PDP_y) observes trace $t_y \in \mathcal{T}_y$, while the corresponding PIP (PIP_y) observes data flow states $\sigma_{t_y} \in \Sigma_y$. It is the union of these local observations that one single global PDP/PIP ($\text{PDP}_\mathcal{Y}/\text{PIP}_\mathcal{Y}$) would observe.

3.1.2 Sets of Systems. We want to reason about the distributed system as a whole as well as subsets thereof. We therefore define for a set of systems $Y \subseteq \mathcal{Y}$: (i) its set of system events $\mathcal{S}_Y \overset{\text{def}}{=} \bigcup_{y \in Y} \mathcal{S}_y$, (ii) its set of containers $C_Y \overset{\text{def}}{=} \bigcup_{y \in Y} C_y$, (iii) its set of identifiers $\mathcal{I}_Y \overset{\text{def}}{=} \bigcup_{y \in Y} \mathcal{I}_y$, and (iv) its set of all possible system runs $\mathcal{T}_Y : \mathbb{N} \to \mathbb{P}(\mathcal{S}_Y)$.
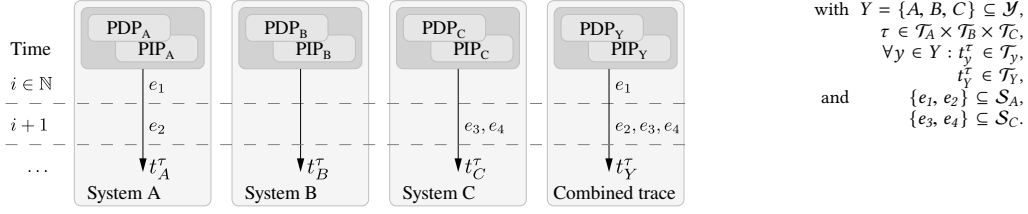
Fig. 2. Three systems and the combination of their traces.

3.1.3 Relating Individual Systems with Sets of Systems. Running systems in parallel produces independent system traces and data flow states (§3.1.1). Because the policy language (§2.1.3) only considers one single system trace and data flow state, we formalize how multiple decentralized system runs and data flow states correlate with the observations of a single central PDP/PIP.

Let $\prod$ denote the Cartesian product. Then $\tau \in \prod_{y \in \mathcal{Y}} \mathcal{T}_y$ is a tuple of traces of all systems; $t_y^\tau \in \mathcal{T}_y$ refers to the trace of system $y \in \mathcal{Y}$. We define $t_Y^\tau \in \mathcal{T}_Y$ as the combined trace of systems $Y \subseteq \mathcal{Y}$. For each timestep $i \in \mathbb{N}$, the set of system events observed in Y corresponds to the union of the system events observed in all single systems $y \in Y$. This corresponds to what a central PDP would observe; see Fig. 2 for an example:

$$\forall Y \subseteq \mathcal{Y}, i \in \mathbb{N}, \tau \in \prod_{y \in \mathcal{Y}} \mathcal{T}_y, t_Y^\tau \in \mathcal{T}_Y : t_Y^\tau(i) \overset{\text{def}}{=} \bigcup_{y \in Y} t_y^\tau(i)$$

Analogously, we combine the individual systems' data flow states to a single *global data flow state* $\sigma_{t_Y^\tau}$, resembling what a central PIP would observe. We obtain $\sigma_{t_Y^\tau}$ by unifying the mappings of the individual systems' storage, alias, and naming functions:

$$\forall Y \subseteq \mathcal{Y}, i \in \mathbb{N}, \tau \in \prod_{y \in \mathcal{Y}} \mathcal{T}_y, y \in Y, t_Y^\tau \in \mathcal{T}_Y, t_y^\tau \in \mathcal{T}_y, \sigma_{t_Y^\tau}^i \in \Sigma_Y, \sigma_{t_y^\tau}^i \in \Sigma_y, c \in C_y, j \in \mathcal{I}_y :$$

$$\Sigma_Y \subseteq \Sigma : (C_Y \to \mathbb{P}(\mathcal{D})) \times (C_Y \to \mathbb{P}(C_Y)) \times (\mathcal{I}_Y \to C_Y)$$

$$\text{with } \sigma_{t_Y^\tau}^i.s(c) \overset{\text{def}}{=} \sigma_{t_y^\tau}^i.s(c) \ \wedge \ \sigma_{t_Y^\tau}^i.a(c) \overset{\text{def}}{=} \sigma_{t_y^\tau}^i.a(c) \ \wedge \ \sigma_{t_Y^\tau}^i.n(j) \overset{\text{def}}{=} \sigma_{t_y^\tau}^i.n(j)$$

In summary, t_Y^τ and $\sigma_{t_Y^\tau}$ reflect the trace and state of a set of systems Y, respectively. This allows to reason about the behavior of individual systems, sets of systems, their interrelations, and the distributed system as a whole. Our models for cross-system data flow tracking (§3.2) and decentralized policy enforcement (§3.3) rely on these definitions.

3.2 Cross-System Data Flow Tracking

To protect all sensitive data copies across all systems, the enforcement infrastructure must know all data representations at all times. With the model from §3.1, each system $y \in Y$ records its internal trace $t_y \in \mathcal{T}_y$ and associated data flow states $\sigma_{t_y} \in \Sigma_y$, recording which data resides in which containers at which point in time. Earlier work used this model to track data within Unix systems [Harvan and Pretschner 2009] and Microsoft Windows [Wüchner and Pretschner 2012], e.g. by following the flow of data from a file into a process and back to another file. In a distributed system, the infrastructure must further be aware of all data copies *across* systems.

Example. In our example, policy 2 inhibits the execution of any events that *use* data that is associated with, or was derived from, a declined contract offer. Consider such a declined contract offer d that was archived in a database. Clerks requesting copies of (parts of) the declined contract offer do not violate the policy. To keep the clerks from using these copies of data d and thus violating the policy, all such (partial) copies of d must be tracked across all of the insurance company's systems at all times.

We instantiate the data flow tracking model (§2.1.2) for distributed systems. Cross-system data flows are recorded within the involved systems' data flow states. Our approach is independent of the communication protocol and transparent to applications and the operating system. This allows to track data for a multitude of applications and protocols. E.g., in the insurance company example, data is exchanged using applications and protocols such as email, web, and possibly proprietary protocols.

Our model integrates with aforementioned works that track data within single systems. We detail some aspects of intra-system tracking to outline the models' integration.

Addresses. Communication methods such as the Internet Protocol (IP) use addresses to identify network participants. Multiple addresses may be assigned to each system, reflecting that a system may in itself be distributed (see §3.1.1). We assume *addresses* $\mathcal{A}$ to be globally unique: no address may be assigned to more than one system over time. The integration of localhost addresses, NAT and DHCP is explained in [Kelbert and Pretschner 2013]. Each system $y \in \mathcal{Y}$ is defined by its set of unique addresses: $\forall y_1, y_2 \in \mathcal{Y} \subseteq \mathbb{P}(\mathcal{A}) \setminus \{\emptyset\} : y_1 = y_2 \overset{\text{def}}{\iff} y_1 \cap y_2 \neq \emptyset$.

3.2.1 Model Instantiation.

To be transparent to applications, we track data flows in correspondence with system calls $e \in \mathcal{S}$. Because these might be intercepted before or after execution by the kernel, they might be intended ($e \in \mathcal{S}^I$) or actual ($e \in \mathcal{S}^A$).

Of interest are system calls whose semantics are such that they influence the data flow state. Transition relation $\mathcal{R}$ reflects these semantics and has been defined for system-internal system calls in [Harvan and Pretschner 2009]. E.g., system calls *creat*, *pipe*, and *fork* create new containers (files, pipes, and processes) and assign corresponding identifiers (filenames, file descriptors, process ids), thus modifying naming function $\sigma.n$. Others (e.g., *open*, *dup*) create new identifiers (file descriptors) for existing containers. System calls such as *read* and *write* migrate data between containers, thereby modifying storage function $\sigma.s$. E.g., if a process reads from a file, then $\sigma.s$ is updated such that all data within the file is also considered to be within the reading process' memory.

For the data flow state σ to correctly reflect (more precisely: overapproximate) the distribution of data within the system, transition relation $\mathcal{R}$ must correctly (and possibly conservatively) define for every system call how it changes mapping $\sigma.s$, $\sigma.a$, and $\sigma.n$. §3.2.2 provides corresponding definitions for networking-related system calls; these are based on system calls' manpage descriptions and observable behaviours.

We base cross-system data flow tracking on the fact that processes (i) send data to the network by writing data from their process memory to a socket (system call *sendto* or equivalent), and (ii) receive data from the network by reading from a socket into their process memory (*recvfrom* or equivalent). Thus, two sets of containers are of particular interest: process memory $C_{Proc} \subseteq C$ and sockets $C_{Sock} \subseteq C$.

We identify process memory C_{Proc} by a system-relative process id $\mathcal{I}_{Pid}$, i.e., $(\mathcal{I}_{Sys} \times \mathcal{I}_{Pid}) \subseteq \mathcal{I}$, with $\mathcal{Y} = \mathcal{I}_{Sys}$. Sockets C_{Sock} are identified via process-relative file descriptors $\mathcal{I}_{Fdsc}$, $(\mathcal{I}_{Sys} \times \mathcal{I}_{Pid} \times \mathcal{I}_{Fdsc}) \subseteq \mathcal{I}$, as well as via addresses $\mathcal{A} = \mathcal{I}_{Addr}$ and ports $\mathcal{I}_{Port}$, i.e., by the address ($\mathcal{A} = \mathcal{I}_{Addr}$) and port ($\mathcal{I}_{Port}$) of the sender and the receiver (*local socket name* and *remote socket name*, respectively): $((\mathcal{I}_{Addr} \times \mathcal{I}_{Port}) \times (\mathcal{I}_{Addr} \times \mathcal{I}_{Port})) \subseteq \mathcal{I}$.

To formalize how data propagates between processes via sockets, we need additional notation: For sets S, T, mappings $m, m' : S \to T$ and $x \in X \subseteq S$, define $m[x \leftarrow expr]_{x \in X} = m'$ such that $m'(y) = expr$ if $y \in X$ and $m'(y) = m(y)$ otherwise.

3.2.2 Data Propagation.
Processes propagate data across systems in three steps: (1) socket creation, (2) sending of data to a socket, (3) receiving of data from a socket. Assume that process $p \in \mathcal{I}_{Pid}$ on system $y_p \in \mathcal{I}_{Sys}$ has read sensitive data $d \in \mathcal{D}$ into its process memory $c_p \in C_{Proc}$. System-internal data flow tracking reflects this as follows: $c_p = \sigma.n((y_p, p))$ and $d \in \sigma.s(c_p)$ [Harvan and Pretschner 2009]. Once process p writes to a network socket, the data written might include data d. Our model conservatively assumes that a process always writes *all* sensitive data associated with its memory.

(1) Socket creation. Each communication partner first creates a socket using system call *socket*. The returned file descriptor enables the calling process to write to and read from the socket. With $c_l \in C$ representing the newly created socket, $fd \in \mathcal{I}_{Fdsc}$ its file descriptor, and p the calling process on host y_p, transition relation $\mathcal{R}$ models this behaviour by updating the data flow state with mapping $c_l = \sigma.n((y_p, p, fd))$. Formally:

$$\forall \sigma, \sigma' \in \Sigma, \forall y_p \in \mathcal{I}_{Sys}, \forall p \in \mathcal{I}_{Pid}, \forall fd \in \mathcal{I}_{Fdsc}, \forall c_l \in C_{Sock} :$$

$$(\sigma, (socket, \{(obj, c_l), (sys, y_p), (proc, p), (ret, fd)\}), \sigma') \in \mathcal{R}$$

$$\implies \sigma'.s = \sigma.s \quad \wedge \quad \sigma'.a = \sigma.a \quad \wedge \quad \sigma'.n = \sigma.n[(y_p, p, fd) \leftarrow c_l]$$

The socket can only be used for communication after it has been assigned a (local) socket name, i.e., a IP address and port $(a_l, o_l) \in (\mathcal{I}_{Addr} \times \mathcal{I}_{Port}) \subseteq \mathcal{I}$. Depending on the communication protocol, this assignment may be implicit (upon first communication) or explicit (using system call *bind*). The assigned socket name can later be obtained from the operating system or from the parameters of system calls *sendto* and *recvfrom*.

(2) Sending data. Consider a process $p_A \in \mathcal{I}_{Pid}$ on host $y_{p_A} \in \mathcal{I}_{Sys}$ that calls *sendto* on file descriptor $fd_A \in \mathcal{I}_{Fdsc}$ identifying socket $c_A = \sigma.n((y_{p_A}, p_A, fd_A))$. This leads to a data flow from the process memory of process p_A to socket c_A. With σ representing the old data flow state and σ' the updated data flow state, transition relation $\mathcal{R}$ models this data flow by updating the storage function of c_A to contain all previous data *plus* all data associated with process p_A: $\sigma'.s(c_A) = \sigma.s(c_A) \cup \sigma.s(\sigma.n((y_{p_A}, p_A)))$. With (a_A, o_A) representing the local socket name of c_A and (a_B, o_B) the name of the remote socket to which p_A sends data, $\mathcal{R}$ further creates a new identifier for c_A, $c_A = \sigma'.n(((a_A, o_A), (a_B, o_B)))$. Formally, $(\sigma, (sendto, < params >), \sigma') \in \mathcal{R}$ is defined as:

$$\forall \sigma, \sigma' \in \Sigma, \forall y_{p_A} \in \mathcal{I}_{Sys}, \forall p_A \in \mathcal{I}_{Pid}, \forall fd_A \in \mathcal{I}_{Fdsc}, \forall a_A, a_B \in \mathcal{I}_{Addr}, \forall o_A, o_B \in \mathcal{I}_{Port} :$$

$$(\sigma, (sendto, \{(obj, \sigma.n((y_{p_A}, p_A, fd_A))), (sys, y_{p_A}), (proc, p_A), (fdscr, fd_A), (lname, (a_A, o_A)), (rname, (a_B, o_B))\}), \sigma') \in \mathcal{R}$$

$$\implies \sigma'.s = \sigma.s[\sigma.n((y_{p_A}, p_A, fd_A)) \leftarrow \sigma.s(\sigma.n((y_{p_A}, p_A, fd_A))) \cup \sigma.s(\sigma.n((y_{p_A}, p_A)))]$$

$$\wedge \sigma'.a = \sigma.a \quad \wedge \quad \sigma'.n = \sigma.n[((a_A, o_A), (a_B, o_B)) \leftarrow \sigma.n((y_{p_A}, p_A, fd_A))]$$

(3) Receiving data. Process p_B on host y_{p_B} then receives data from socket $c_B \in C_{Sock}$ by issuing system call *recvfrom* on file descriptor $fd_B \in \mathcal{I}_{Fdsc}$. Consequently, all data sent by the remote process is propagated into the process memory of p_B. Transition relation $\mathcal{R}$ reflects this data flow by updating the storage function of the process memory of p_B to contain all previous data *plus* all data contained in the remote socket identified by $((a_A, o_A), (a_B, o_B))$: $\sigma'.s(\sigma.n((y_{p_B}, p_B))) = \sigma.s(\sigma.n((y_{p_B}, p_B))) \cup \sigma.s(\sigma.n(((a_A, o_A), (a_B, o_B))))$. Formally, $(\sigma, (recvfrom, < params >), \sigma') \in \mathcal{R}$ is defined as:

$\forall \sigma, \sigma' \in \Sigma, \forall y_{p_B} \in \mathcal{I}_{Sys}, \forall p_B \in \mathcal{I}_{Pid}, \forall fd_B \in \mathcal{I}_{Fdsc}, \forall a_A, a_B \in \mathcal{I}_{Addr}, \forall o_A, o_B \in \mathcal{I}_{Port}:$

$(\sigma, (recvfrom, \{(obj, \sigma.n((y_{p_B}, p_B, fd_B))), (sys, y_{p_B}), (proc, p_B), (fdscr, fd_B), (lname, (a_B, o_B)), (rname, (a_A, o_A))\}), \sigma') \in \mathcal{R}$

$\implies \sigma'.s = \sigma.s[\sigma.n((y_{p_B}, p_B)) \leftarrow \sigma.s(\sigma.n((y_{p_B}, p_B))) \cup \sigma.s(\sigma.n(((a_A, o_A), (a_B, o_B))))] \quad \wedge \quad \sigma'.a = \sigma.a \quad \wedge \quad \sigma'.n = \sigma.n$

In combination with intra-system data flow tracking, these definitions track data flows within and across systems. They ensure soundness by overapproximating data flows. While overapproximations might lead to policies being enforced for containers that in reality do not contain sensitive data, they are required because system calls can not capture process internal data flows. Higher precision can be achieved by application internal data flow tracking. In practice, overapproximations are limited for two reasons: (i) Sockets are identified by the local and remote socket name. Data written to a socket thus only affects the communicating processes. (ii) Server applications often fork worker processes to process requests. The parent process remains untainted and keeps forking untainted workers. We discuss and evaluate overapproximations in §5.1 and §5.2.2.

We defer cross-system policy propagation and further implementation details to §4.2. With the above models, policy propagation boils down to attaching the corresponding policy whenever a cross-system data flow is detected. §5.2 provides evaluation results.

3.3 Coordinating Policy Decisions Across Systems

After protected data was distributed to multiple sytems, the corresponding policies must be consistently enforced. We envision PDPs to be decentrally deployed, thus overcoming the drawbacks of a centralized infrastructure by taking policy decisions locally, efficiently, conclusively, and consistently.

Example. Consider again policy 2 from our example and a situation in which (i) the customer has declined the corresponding contract offer and (ii) multiple clerks have local copies of associated, possibly derived, data d. Using a centralized enforcement infrastructure, *all* data usage events on *any* data from *all* clerks must always be evaluated by the central PDP/PIP. This is because local PEPs lack any logic and knowledge to decide whether a particular event handles protected data. If PDPs and PIPs were decentralized, then events not involving data d can always be allowed locally. Events that do involve d only necessitate coordination with other PDPs if the contract offer was never declined; otherwise local PDPs can instantly deny any such events.

Enforcing global policies in a decentralized manner introduces the necessity for coordination: Each PDP's decisions might depend on past decisions and observations of other PDPs. While each PDP could reveal all its information to all other PDPs, such a solution imposes significant runtime and communication overheads. We limit the coordination effort by identifying a minimal and complete set of systems necessary to evaluate a given policy. §3.3.1 provides a first (over-)approximation of the set of systems *relevant* for evaluating a given policy; §3.3.2 identifies situations in which coordination between such relevant systems can be safely omitted without compromising policy enforcement.

3.3.1 Identifying Relevant Systems. This section identifies an overapproximation of the set of systems relevant for evaluating a given ECA rule p. We define auxiliary functions:

(1) $knowC : \mathbb{P}(C) \to \mathbb{P}(\mathcal{Y})$. Given a set of containers, $knowC$ returns the set of systems in which one of those containers resides: $\forall C \subseteq C : knowC(C) \overset{\text{def}}{=} \{y \in \mathcal{Y} \mid C_y \cap C \neq \emptyset\}$.

(2) $knowD : \mathbb{P}(\mathcal{D}) \times \mathbb{N} \times \prod \mathcal{T} \to \mathbb{P}(\mathcal{Y})$. Given a set of data items, a point in time, and a tuple of concurrently executing traces, $knowD$ relies on the systems' data flow states to return the set of systems in which there exists a container

containing at least one of the data items:

$$\forall D \subseteq \mathcal{D}, i \in \mathbb{N}, \tau \in \prod_{y \in \mathcal{Y}} \mathcal{T}_y : knowD(D, i, \tau) \stackrel{def}{=} \{y \in \mathcal{Y} \mid \exists c \in C_y, t_y^\tau \in \mathcal{T}_y, \sigma_{t_y^\tau}^i \in \Sigma_y : D \cap \sigma_{t_y^\tau}^i.s(c) \neq \emptyset\}.$$

knowD is parameterized in time because data keeps being propagated across systems.

(3) *happens* : $\mathcal{E} \times \mathbb{N} \times \prod \mathcal{T} \rightarrow \mathbb{P}(\mathcal{Y})$. Given an event, a point in time, and a tuple of concurrently executing traces, *happens* returns the set of systems in which an event refining e happens:

$$\forall e \in \mathcal{E}, i \in \mathbb{N}, \tau \in \prod_{y \in \mathcal{Y}} \mathcal{T}_y : happens(e, i, \tau) \stackrel{def}{=} \{y \in \mathcal{Y} \mid \exists t_y^\tau \in \mathcal{T}_y, e' \in t_y^\tau(i), \sigma_{t_y^\tau}^i \in \Sigma_y : (e', \sigma_{t_y^\tau}^i) \; ref_\Sigma \; e\}.$$

Taken together, *knowC*, *knowD*, and *happens* allow to identify the set of systems that 'know' a certain container or data, or in which a particular event happened. On this basis, we define function *relevant* : $\Phi \times \mathbb{N} \times \prod \mathcal{T} \rightarrow \mathbb{P}(\mathcal{Y})$. For an ECA rule p, *relevant* identifies the set of systems that *potentially* contribute to the evaluation of p's condition $\varphi_p \in \Phi$ at a given point in time $i \in \mathbb{N}$ and given a set of concurrently executing traces $\tau \in \prod_{y \in \mathcal{Y}} \mathcal{T}_y$. The observation behind the definition of *relevant* is that solely atomic operators of φ_p ($\mathcal{E}$, *combined*, *notIn*, *maxIn*, *repmin*) determine which systems are relevant[2]:

$$\forall \varphi \in \Phi, i \in \mathbb{N}, \tau \in \prod_{y \in \mathcal{Y}} \mathcal{T}_y : relevant(\varphi, i, \tau) \stackrel{def}{=} \{y \in Y \mid$$

$$((\varphi = \underline{true} \vee \varphi = \underline{false}) \implies Y = \emptyset)$$

$$\vee \exists e \in \mathcal{E} \bullet (\varphi = e \implies Y = happens(e, i, \tau))$$

$$\vee \exists d \in \mathcal{D}, m \in \mathbb{N}, C \subseteq C \bullet ((\varphi = \underline{notIn}(d, C) \vee \varphi = \underline{maxIn}(d, m, C)) \implies Y = knowD(\{d\}, i, \tau) \cap knowC(C))$$

$$\vee \exists d_1, d_2 \in \mathcal{D}, C \subseteq C \bullet (\varphi = \underline{combined}(d_1, d_2, C) \implies Y = knowD(\{d_1\}, i, \tau) \cap knowD(\{d_2\}, i, \tau) \cap knowC(C))$$

$$\vee \exists \alpha \in \Phi \bullet (\varphi = \underline{not}(\alpha) \implies Y = relevant(\alpha, i, \tau))$$

$$\vee \exists \alpha, \beta \in \Phi \bullet ((\varphi = \alpha \; \underline{and} \; \beta \vee \varphi = \alpha \; \underline{or} \; \beta) \implies Y = relevant(\alpha, i, \tau) \cup relevant(\beta, i, \tau))$$

$$\vee \exists \alpha, \beta \in \Phi \bullet (\varphi = \alpha \; \underline{since} \; \beta \implies Y = \bigcup_{j=0}^{i}(relevant(\alpha, j, \tau) \cup relevant(\beta, j, \tau)))$$

$$\vee \exists \alpha \in \Phi, j \in \mathbb{N} \bullet (\varphi = \alpha \; \underline{before} \; j \implies Y = relevant(\alpha, i - j, \tau))$$

$$\vee \exists j, m \in \mathbb{N}, e \in \mathcal{E} \bullet (\varphi = \underline{repmin}(j, m, e) \implies Y = \bigcup_{k=0}^{min\{i,j\}-1} happens(e, i - k, \tau)) \quad \}$$

The rationale is as follows: Trivial formulas of $\varphi = \underline{true}$ and $\varphi = \underline{false}$ can always be evaluated locally. For $\varphi = e$, relevant systems are those in which an event refining e happens at the current timestep. For operators $\underline{notIn}(d, C)$ and $\underline{maxIn}(d, m, C)$, relevant systems know data d and some $c \in C$. For $\underline{combined}(d_1, d_2, C)$, relevant systems are those that know d_1, d_2, and some $c \in C$. For $\underline{repmin}(j, m, e)$, relevant systems are those in which an event refining e has happened in the last j timesteps. For $\underline{not}$, $\underline{and}$, $\underline{or}$, $\underline{since}$ and $\underline{before}$, relevant systems are determined recursively.

Our goal was to identify a minimal and complete set of systems necessary to evaluate a given policy. The following theorem states that $relevant(\varphi, i, \tau)$ is complete. Intuitively: systems $\mathcal{Y} \setminus relevant(\varphi, i, \tau)$ do not change the policy evaluation result. The proof is provided in [Kelbert and Pretschner 2014]:

Theorem. $\forall \varphi \in \Phi, i \in \mathbb{N}, \forall \tau \in \prod_{y \in \mathcal{Y}} \mathcal{T}_y, Y = relevant(\varphi, i, \tau), X \subseteq \mathcal{Y} \setminus Y : \quad (t_Y^\tau, i) \models \varphi \iff (t_{Y \cup X}^\tau, i) \models \varphi.$

---

[2]Here and in the following we omit the description of the underspecified policy operator $\underline{evalExt}$. Given one concrete framework realizing $\underline{evalExt}$, similar complementary analyses would need to be performed.

Due to the above disambiguity, we will write $t^\tau_{\mathcal{Y}}$ to refer to trace $t^\tau_{relevant(\varphi,i,\tau)}$.

In summary, if $|relevant(\varphi, i, \tau)| \leq 1$ then decisions can be taken locally. Otherwise, PDPs and PIPs must reveal all event occurrences and state changes to all systems $relevant(\varphi, i, \tau)$. This already improves over revealing all information to all systems $\mathcal{Y}$. §3.3.2 further identifies situations in which communication with even fewer systems is needed, thus working towards a minimal set of systems.

3.3.2 Omitting Unnecessary Communication. Given a set of systems $Y \subsetneq relevant(\varphi, i, \tau)$, it is generally not possible to conclusively evaluate formula $\varphi \in \Phi$ at time $i \in \mathbb{N}$: the evaluation of φ might depend on information unavailable within Y. However, in certain situations, φ can be evaluated even if some information is unavailable. We identify such situations to further reduce the required communication.

Given $\tau \in \prod_{y \in \mathcal{Y}} \mathcal{T}_y$, there might exist a formula $\varphi' \in \Phi$ such that Y satisfies φ', and this implies global satisfaction of φ. Formally: $(t^\tau_Y, i) \models \varphi' \implies (t^\tau_{\mathcal{Y}}, i) \models \varphi$. Whether such Y and φ' exist depends on the characteristics of φ. We first define *formula projections*, 'hiding' parts of φ that are unknown within $Y \subsetneq relevant(\varphi, i, \tau)$, i.e. parts that can only be evaluated by systems $\mathcal{Y} \setminus Y$. We define the projection $\varphi_Y \in \Phi$ of $\varphi \in \Phi$ as:

$\forall \varphi \in \Phi, Y \subsetneq relevant(\varphi, i, \tau), \exists \varphi_Y :$

$((\varphi = \underline{true} \vee \varphi = \underline{false}) \implies \varphi_Y = \varphi)$

$\vee \, \exists e \in \mathcal{E} \bullet (\varphi = e \implies \varphi_Y = e)$

$\vee \, \exists d \in \mathcal{D}, C \subseteq C \bullet (\varphi = \underline{notIn}(d, C) \implies \varphi_Y = \underline{notIn}(d, C \cap C_Y))$

$\vee \, \exists d_1, d_2 \in \mathcal{D}, C \subseteq C \bullet (\varphi = \underline{combined}(d_1, d_2, C) \implies \varphi_Y = \underline{combined}(d_1, d_2, C \cap C_Y))$

$\vee \, \exists d \in \mathcal{D}, m \in \mathbb{N}, C \subseteq C \bullet (\varphi = \underline{maxIn}(d, m, C) \implies \varphi_Y = \underline{maxIn}(d, m, C \cap C_Y))$

$\vee \, \exists \alpha \in \Phi \bullet (\varphi = \underline{not}(\alpha) \implies \varphi_Y = \underline{not}(\alpha_Y))$

$\vee \, \exists \alpha, \beta \in \Phi \bullet (\varphi = \alpha \ \underline{and} \ \beta \implies \varphi_Y = \alpha_Y \ \underline{and} \ \beta_Y)$

$\quad \vee \, (\varphi = \alpha \ \underline{or} \ \beta \implies \varphi_Y = \alpha_Y \ \underline{or} \ \beta_Y)$

$\quad \vee \, (\varphi = \alpha \ \underline{since} \ \beta \implies \varphi_Y = \alpha_Y \ \underline{since} \ \beta_Y)$

$\vee \, \exists \alpha \in \Phi, j \in \mathbb{N} \bullet (\varphi = \alpha \ \underline{before} \ j \implies \varphi_Y = \alpha_Y \ \underline{before} \ j)$

$\vee \, \exists j, m \in \mathbb{N}, e \in \mathcal{E} \bullet (\varphi = \underline{repmin}(j, m, e) \implies \varphi_Y = \underline{repmin}(j, m, e))$

These projections allow to identify and formalize situations in which only a subset of relevant systems is sufficient to evaluate a given formula. Consequently, coordination is only required among a subset of all relevant systems, thus reducing communication.

This fact is captured by predicate $Sat \subseteq \prod \mathcal{T} \times \mathbb{P}(\mathcal{Y}) \times \mathbb{N} \times \Phi$, which is shortly defined to hold *true* iff for $\tau \in \prod_{y \in \mathcal{Y}} \mathcal{T}_y$ and $Y \subsetneq relevant(\varphi, i, \tau)$, trace $t^\tau_Y \in \mathcal{T}_Y$ satisfies φ_Y at time $i \in \mathbb{N}$ $((t^\tau_Y, i) \models \varphi_Y)$ and if this implies global satisfaction of formula $\varphi \in \Phi$ at the same point in time $((t^\tau_{\mathcal{Y}}, i) \models \varphi)$:

Theorem. $\forall \tau \in \prod\limits_{y \in \mathcal{Y}} \mathcal{T}_y, Y \subsetneq relevant(\varphi, i, \tau), i \in \mathbb{N}, \varphi \in \Phi : (t^\tau_Y, i) \models \varphi_Y \ \wedge \ Sat(\tau, Y, i, \varphi) \implies (t^\tau_{\mathcal{Y}}, i) \models \varphi$

Intuitively: if φ_Y is satisfied locally and if $Sat(\tau, Y, i, \varphi)$ holds, then the set of systems Y can conclusively infer the global evaluation result of formula φ. No communication with any other systems is thus required.

The definition of Sat is based on three observations: (i) some atomic operators (e.g., $\mathcal{E}$, $\underline{combined}$), are satifisfied globally if they are satisfied in at least one system; (ii) some atomic operators (e.g., $\underline{notIn}$, $\underline{maxIn}$) are violated globally if they are violated in only one system; (iii) the global satisfaction of compound operators (e.g., $\underline{and}$, $\underline{since}$) depends on their semantics as well as the local satisfaction of the nested operators.

To define Sat, we demand $\varphi \in \Phi$ to be given in disjunctive normal form (DNF). The reason is that for atomic operators the results of Sat should be different depending on whether they are 'encapsulated' in an even or an odd number of negations. When φ is in DNF, then negations only occur next to atomic operators. The following definition only enumerates cases for which $Sat = true$; other cases yield $Sat = false$. Proofs of correctness of the above theorem are provided in [Kelbert and Pretschner 2014].

$$\forall \tau \in \prod_{y \in \mathcal{Y}} \mathcal{T}_y, Y \subseteq \mathcal{Y}, i \in \mathbb{N}, \varphi \in \Phi :$$

$$Sat(\tau, Y, i, \varphi) \overset{\text{def}}{\iff} \varphi = \underline{true} \lor \varphi = \underline{false} \lor relevant(\varphi, i, \tau) \subseteq Y$$

$$\lor \exists e \in \mathcal{E} \bullet (\varphi = e)$$

$$\lor \exists d \in \mathcal{D}, C \subseteq C \bullet (\varphi = \underline{not}(\underline{notIn}(d, C)))$$

$$\lor \exists d_1, d_2 \in \mathcal{D}, C \subseteq C \bullet (\varphi = \underline{combined}(d_1, d_2, C))$$

$$\lor \exists d \in \mathcal{D}, m \in \mathbb{N}, C \subseteq C \bullet (\varphi = \underline{not}(\underline{maxIn}(d, m, C)))$$

$$\lor \exists \alpha, \beta \in \Phi \bullet ((\varphi = \alpha \ \underline{and} \ \beta \land Sat(\tau, Y, i, \alpha) \land Sat(\tau, Y, i, \beta))$$

$$\lor (\varphi = \alpha \ \underline{or} \ \beta \land ((t_Y^\tau, i) \models \alpha_Y \land Sat(\tau, Y, i, \alpha) \lor (t_Y^\tau, i) \models \beta_M \land Sat(\tau, Y, i, \beta)))$$

$$\lor (\varphi = \alpha \ \underline{since} \ \beta \land ((\exists j \in [0, i] : (t_Y^\tau, j) \models \beta_Y \land Sat(\tau, Y, j, \beta)$$

$$\land \forall k \in (j, i] : (t_Y^\tau, k) \models \alpha_Y \land Sat(\tau, Y, k, \alpha))$$

$$\lor (\forall k \in [0, i] : (t_Y^\tau, k) \models \alpha_Y \land Sat(\tau, Y, k, \alpha)))))$$

$$\lor (\varphi = \underline{not}(\alpha \ \underline{since} \ \beta) \land ((\forall j \in [0, i] : (t_Y^\tau, j) \not\models \beta_Y \land Sat(\tau, Y, j, \underline{not}(\beta))$$

$$\land \exists k \in (j, i] : (t_Y^\tau, k) \not\models \alpha_Y \land Sat(\tau, Y, k, \underline{not}(\alpha)))$$

$$\lor (\exists k \in [0, i] : (t_Y^\tau, k) \not\models \alpha_Y \land Sat(\tau, Y, k, \underline{not}(\alpha))))))$$

$$\lor \exists \alpha \in \Phi, j \in \mathbb{N} \bullet (\varphi = \alpha \ \underline{before} \ j \land Sat(\tau, Y, i - j, \alpha) \lor \varphi = \underline{not}(\alpha \ \underline{before} \ j) \land Sat(\tau, Y, i - j, \underline{not}(\alpha)))$$

$$\lor \exists e \in \mathcal{E}, j, m \in \mathbb{N} \bullet (\varphi = \underline{repmin}(j, m, e))$$

In summary, predicate Sat identifies situations in which no coordination between systems is required despite $|relevant(\varphi, i, \tau)| > 1$. §4.3 details how our implementation leverages these results. §5.3 shows how the above considerations improve both performance and communication overheads when enforcing global data usage policies.

4 ARCHITECTURE AND IMPLEMENTATION

In line with §3, we present an architecture and implementation that (i) tracks data flows across systems and propagates the corresponding policies (§4.2), and (ii) preventively, efficiently, and consistently enforces global policies (§4.3).
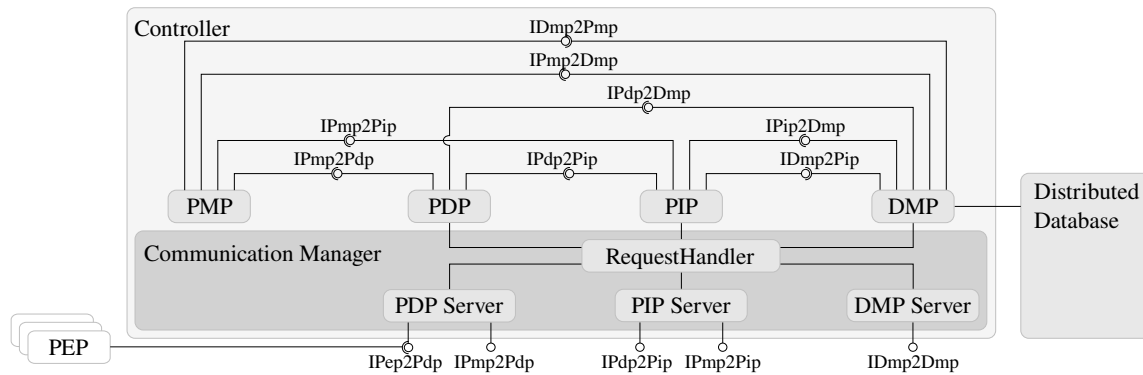
Fig. 3. Our architecture and its most important components and interfaces.

4.1 High-level Architecture

By definition (see §3.1.1), a system consists of multiple PEPs sharing the same PDP/PIP. The fact that these PEPs might be distributed makes our architecture (see Fig. 3) highly flexible: our system might be deployed (i) on each physical system to implement a fully decentralized infrastructure, (ii) as a single central component, (iii) in a hybrid fashion.

The architecture is composed of PEPs, a `Controller`, and a distributed database. The `Controller` features the PDP, PIP, and PMP, and coordinates their internal communication. We minimize changes to existing local enforcement infrastructures by introducing a Distribution Management Point (DMP). The DMP organizes cross-system data flow tracking and policy propagation in a peer-to-peer manner (§4.2) and leverages a distributed database to coordinate distributed policy decisions (§4.3).

The `CommunicationManager` manages all external communication and runs servers that exhibit the components' functionalities to the outside. Most importantly, the `PdpServer` provides the interface for PEPs to request data usage control decisions. Additional servers allow to deploy PDP, PIP, and PMP remotely from one another, thus catering to complex deployment requirements. The `DmpServer` enables the communication with remote DMPs. We secure all communication using TLS and assume the integrity and mutual authentication of remote `Controllers`; this is discussed in §5.1.

The `RequestHandler` organizes the sequential and in-order processing of requests arriving at the servers, thus avoiding event reordering and race conditions.

We implemented the above infrastructure in Java. To support its flexible deployment, the internal components can communicate both via function calls as well as via the provided servers. A straightforward strategy is to deploy one single central `Controller`, thus mimicking the behavior of a centralized infrastructure. However, our rationale is to deploy PDPs and PIPs locally, e.g. per machine or department. This way, PEPs query their local PDP and each PIP keeps a local data flow state.

We further implemented a Linux PEP on the basis of strace [Kranenburg and Levin 2015]. The PEP intercepts system calls both before and after their execution by the kernel [Harvan and Pretschner 2009], thus signaling both intended and actual system calls to the PDP. Note that our architecture supports existing PEPs, e.g. for Microsoft Windows [Wüchner and Pretschner 2012] and Android [Feth and Pretschner 2012].

Several interfaces organize the communication between the components:

The PDP provides interfaces `IPmp2Pdp` and `IPep2Pdp`, providing methods to deploy and retrieve policies, as well as to signal system events and await policy decisions.

At the PIP, the PDP uses interface `IPdp2Pip` to evaluate state-based operators, retrieve all data within a given container, and signal data flow system events. The PMP uses interface `IPmp2Pip` to inform the PIP about initial data representations (see §4.2). Interface `IDmp2Pip` is used by the DMP in case cross-system data flows occur.

The PMP provides interface `IDmp2Pmp`, which is used by the local DMP to deploy policies and to retrieve currently deployed policies.

The DMP provides four interfaces: (1) `IPip2Dmp` allows the PIP to inform the DMP about data flows to remote socket containers. (2) `IPdp2Dmp` enables the coordination of policy decisions across PDPs. This includes methods to (i) notify that an operator's state has changed, (ii) query whether the state of some operator has changed at remote PDPs, and (iii) synchronize the points in time in which policies are evaluated. (3) `IPmp2Dmp` allows the PMP to register policies at the DMP. (4) `IDmp2Dmp` provides functionalities for cross-system data flow tracking and policy propagation between remote DMPs.

In the following, we assume one fully functional `Controller` to be deployed per system. §4.2 describes how we achieve cross-system data flow tracking and policy propagation, while §4.3 describes the coordination of distributed policy decisions.

4.2 Cross-System Data Flow Tracking and Policy Propagation

Socket communication involves two communication partners. We call these a client and a server process, albeit there is no technical difference once communication was initiated. Our approach applies transitively for larger numbers of processes. We focus on the case in which the client and the server process are remote in the sense that different PIPs are responsible for tracking the processes' data flows. We describe a communication protocol independent implementation in line with §3.2.

4.2.1 Communication Initiation. When the server process s (resp. client process c) executes system call *socket*, the implementation creates a socket container $c_s \in C_{Sock}$ (resp. $c_c \in C_{Sock}$) and an auxiliary *proxy socket container* $c_c^p \in C_{Proxy} \subseteq C_{Sock}$ (resp. $c_s^p \in C_{Proxy}$). Proxy socket containers constitute local placeholders for the actual remote socket container and they are identified via the same identifier. For the server, c_c^p represents remote container c_c. For the client, c_s^p represents remote container c_s. Upon connection initiation, the two responsible PIPs, PIP_S and PIP_C, bidirectionally alias c_s with c_c^p and c_c with c_s^p. Note that this does not incur communication between the two PIPs.

4.2.2 Data Transmission and Policy Propagation. The two processes then exchange data by writing to and reading from the sockets. The DMPs track those data flows and transfer the corresponding policies. We describe how the implementation reflects a data transfer from a client process p_c on system C to a server process p_s on system S.

First, PEP_C intercepts the client process' *intended sendto*$(o, y_{p_c}, p_c, fd, (a_c, o_c), (a_s, o_s))$ (see §3.2.2) and signals it to PDP_C. In case PDP_C allows the event's execution, PIP_C detects the *intended* cross-system data flow. This is the case if the current state $\sigma \in \Sigma$ is such that data $D = \sigma.s(\sigma.n((y_{p_c}, p_c, fd)))$ is propagated to a container of type C_{Proxy}, i.e. if $\exists i \in ((\mathcal{I}_{Addr} \times \mathcal{I}_{Port}) \times (\mathcal{I}_{Addr} \times \mathcal{I}_{Port})), c_s^p \in C_{Proxy}$ such that $\sigma.n(i) = c_s^p$ and $c_s^p \in \sigma.a(\sigma.n((y_{p_c}, p_c, fd)))$. PIP_C then informs DMP_C about the intended remote data flow, conveying data D as well as the destination container's identifier i.

DMP_C then retrieves from PMP_C the set of policies P that constrain the usage of any data within set D. DMP_C then contacts DMP_S, conveying policies P and the fact that data D is about to flow into the container identified by i. DMP_S

informs PIP_S about the incoming data flow, which updates the storage function of the container identified by i with D. DMP_S also informs PMP_S about policies P and PMP_S deploys them at PDP_S.

After this call to DMP_S succeeded, the actual *sendto* system call is executed and the payload data flows. Once the server process p_s invokes *recvfrom* on the socket, PDP_S and PIP_S are already aware of the data flow and enforce policies P on the server's system.

The PDPs' and PIPs' states should only evolve persistently in the presence of *actual* system events (§2.2.2). This, however, leads to race conditions if *sendto* and *recvfrom* interleave. We mitigate this problem by conservative modeling: whenever the PDP allows an *intended sendto*, both the PDPs' and PIPs' states immediately evolve persistently.

4.3 Taking Distributed Policy Decisions

Global policies must be enforced across all systems. Based on §3.3, we detail the practical evaluation of conditions Φ. We take the view of the PDP of system A, PDP_A, which enforces ECA rule p with trigger event $e_p \in \mathcal{E}$, condition $\varphi_p \in \Phi$, and action a_p.

Any event signaled to PDP_A potentially changes the state of leaves within the expression tree of φ_p (see §2.2.2). Since such state changes are of potential importance for other PDPs as approximated by function *relevant*, PDP_A must inform them about any such state changes. Hence, whenever state changes occur at PDP_A, it informs its DMP, DMP_A. DMP_A makes these state changes available to other PDPs via their DMPs.

§4.3.1 abstractly describes decision coordination, assuming that DMPs are consistently, reliably, and timely informed about state changes. §4.3.2 describes technicalities.

4.3.1 Coordinating Distributed Policies. ECA rule p is evaluated whenever a timestep has passed or whenever an event matches p's trigger event e_p (see §2.2.2). In any case each PDP first evaluates φ_p locally, yielding $eval(\varphi_p)$ as described in §2.2.2.

If $eval(\varphi_p) = true = Sat(\tau, Y, i, \varphi_p)$, i.e. if local satisfaction of φ_p implies its global satisfaction, then no further coordination is necessary: action a_p *will* be executed. Similarly, if $eval(\varphi_p) = false = Sat(\tau, Y, i, \varphi_p)$, then local violation of φ_p implies the global violation of φ_p and no coordination is required; action a_p will *not* be executed. Hence, no coordination is required if $eval(\varphi_p) = Sat(\tau, Y, i, \varphi_p)$.

If, however, $eval(\varphi_p) \neq Sat(\tau, Y, i, \varphi_p)$, then the evaluation result of φ_p might depend on other PDPs' state changes. PDP_A re-evaluates φ_p with the help of DMP_A, reflecting the definition of predicate *Sat*: For leaf operators $o \in \{\mathcal{E}, \underline{combined}, \underline{repmin}\}$, PDP_A queries DMP_A if (i) o is *not* negated and if $eval(o) = false$, or if (ii) it *is* negated ($\underline{not}(o)$) and if $eval(\underline{not}(o)) = true$. For leaf operators $o' \in \{\underline{notIn}, \underline{maxIn}\}$, PDP_A queries DMP_A if (i) o' is *not* negated and if $eval(o') = true$, or if (ii) it *is* negated ($\underline{not}(o')$) and if $eval(\underline{not}(o')) = false$. If any of those queries yields a result different from ($eval(\varphi_p)$), then φ_p is re-evaluated, considering this newly available information.

To obtain consistent decisions for time-based policies, the PDPs coordinate policy evaluation times via the DMPs. While such synchronization is subject to scheduling and clock synchronization issues, our implementation synchronizes clocks using NTP. We did not experience any evaluation inconsistencies.

4.3.2 Implementation using Cassandra. We synchronize decisions using the distributed database Cassandra. Here, the set of all nodes is called a *cluster*. The data is organized via *keyspaces* and each *table* is associated with exactly one keyspace. Each keyspace is configured with a *replication strategy*, defining among which nodes its tables are replicated. Hence, data with the same replication requirements is organized within the same keyspace. In our context, PDPs need to enforce several policies and for each the set of coordinating PDPs might differ. Hence, we represent each policy by

one dynamically adjusting keyspace. E.g., for policy p constraining the usage of data d with representations in systems A and B, we employ keyspace k_p with replication strategy $k_p^r = \{A, B\}$. If PDP_A notifies a state change of p to DMP_A, then this information is made available within k_p and replicated to DMP_B.

We achieve strong data consistency by performing all queries with consistency level *Quorum*. Each query is thus acknowledged by at least half of the nodes. This avoids inconsistent states and race conditions when evaluating policies. Because queries might fail due to network partitions, it is configurable how often failed queries are retried. If queries keep failing, the PDP takes a fallback decision as configured within the policy.

Bootstrapping and Cross-System Data Flows. Once the first policy p, protecting data d, is deployed at system A, PMP_A (i) informs PIP_A about the initial representation of d as configured within p, (ii) registers policy p at DMP_A, which prepares keyspace k_p with $k_p^r = \{A\}$, and (iii) deploys p at PDP_A. For now, PDP_A can independently evaluate p.

Once system A shares data d with system B (see §4.2), the latter might influence evaluation p. DMP_B thus adapts keyspace k_p to incorporate the Cassandra node of system B: $k_p^r = \{A, B\}$. Subsequently, all information in k_p is replicated to both DMP_A and DMP_B. If data d is further shared with system C, DMP_C adapts the keyspace to $k_p^r = \{A, B, C\}$, thus replicating all information in k_p to nodes A, B, and C.

Connecting Nodes. In Cassandra, nodes join the cluster by contacting well-known seed nodes. Since our infrastructure ought to be fully decentralized, we proceed as follows.

Reconsider the scenario in which policy p, protecting data d, is deployed only at PDP_A. Later, d, and hence p, is transferred to system B. Earlier we assumed that system B's Cassandra node already participated in the cluster. Otherwise, Cassandra node B must join the cluster without any well-known seed nodes. We do so by starting each Cassandra node only once the first global policy must be enforced: Once DMP_B is informed about some cross-system data flow from DMP_A, it gets to know address a of system A's Cassandra node. Cassandra node B is then started, using a as a seed node.

Consider an extended scenario in which systems A and B enforce policy p, protecting data d, while system C enforces policy p' protecting data d'. Since the sets of systems enforcing p and p' are disjoint, the overall cluster is partitioned. Once data d is transferred from system A to system C, we merge these partitions by starting a temporary Cassandra node, which uses both Cassandra nodes A and C as seed nodes. Once the previously autonomous parts merged, the temporary node is taken down again.

5 EVALUATION

We perform a security analysis of our infrastructure (§5.1) and evaluate which communication and performance overheads our solution introduces (§5.2 and §5.3).

5.1 Security Analysis

Since our infrastructure enforces compliance with data usage policies, we primarily consider attacks trying to obtain unprotected data copies. We further consider attacks on the integrity and availability of our infrastructure and usage controlled data.

5.1.1 Possible Attacks and Countermeasures. We assume that all aspects of our infrastructure were modeled and implemented correctly. To ensure soundness, we conservatively modeled data flow tracking (§3.2) and the identification of relevant systems (§3.3).

Privilege Escalation. Attackers might leverage vulnerabilities in our infrastructure, the operating system, cryptographic libraries and access control mechanisms to escalate their privileges. If successful, they may tamper with the usage control infrastructure and disable policy enforcement or data flow tracking. We consider the mitigation of such security vulnerabilities an orthogonal problem that we do not address in this work.

Copy Data at Rest. Attackers might try to create unprotected copies of usage controlled data that has been stored to disk or other storage media. Since our PEP mediates system calls, our infrastructure thwarts such attacks that originate from system layers above the operating system. In particular, usage control policies can keep users from saving protected data to removable storage media [Feth and Pretschner 2012]. Attacks can therefore only be performed at or below the operating system layer. Disk-encryption software can keep users and administrators from physically unmounting the harddisk and accessing the protected data on non-usage-controlled systems. Encryption keys must then only be accessible to the usage control infrastructure, see §7. Usage control metadata, such as policies and configuration files, can be secured alike.

Share Data with Unmonitored Systems. Attackers might try to use application-layer protocols to share usage controlled data with systems that do not run a usage control infrastructure. Once our infrastructure detects such an intended cross-system data flow, it contacts the remote system's infrastructure (see §4.2.2). Our prototype denies sharing of usage controlled data with non usage controlled systems by inhibiting cross-system data flows if the remote infrastructure does not respond according to the protocol. Mechanisms such as remote attestation can further secure this procedure [Agreiter et al. 2007], see §7. Future solutions might also allow to release protected data after it was encrypted with a key that is only known to the usage control infrastructure.

Eavesdrop Data in Transit. A Man-in-the-Middle might eavesdrop sensitive payload data if applications transfer usage controlled data via plaintext protocols such as HTTP. Such attacks can be prevented by (i) disallowing plaintext protocols for protected data, e.g. using firewalls or deep packet inspection, or by (ii) tunneling such traffic over secure protocols such as VPN or SSH. Both deep packet inspection and protocol tunneling may be implemented in a transparent manner as part of the usage control infrastructure. We protect any of our infrastructure's remote communication via the TLS protocol.

Infrastructure Integrity. Attackers may try to tamper with the infrastructure, e.g. by modifying its binaries. The infrastructure's integrity must therefore be protected by digital signatures and corresponding verification procedures. While such techniques are increasingly implemented in commodity systems, we consider this an orthogonal issue.

Data and System Availability. Attackers might taint large parts of the system with sensitive data by 'leveraging' data flow tracking overapproximations (§3.2). If the data's policy disallows any data usage, no further usage of any tainted parts of the system would be possible. However, such attacks are limited to resources for which the attacker has write permissions, since sensitive data can only be propagated to those data containers. To mitigate such attacks, data flow tracking models must be refined to track data at a finer granularity. Corresponding solutions that track data within web browsers, databases, email clients and windowing systems have been proposed [Kumari et al. 2011; Lienert 2012; Lörscher 2012; Pretschner et al. 2009, 2012].

Denial of Service attacks may be mounted by issuing massive amounts of system events, leading to system overload and to (temporary) unavailability of usage controlled data. Mutual authentication between all infrastructure components, e.g., using certificates or remote attestation, may thwart such attacks and introduce liabilities. Sophisticated distributed denial of service attacks are hard to counteract to date.

Side Channels and Media Breaks are out of scope of this work. Once data is printed or displayed, we can no longer guarantee its protection. Corresponding protection mechanisms are being researched, e.g., [Echizen et al. 2015].

5.1.2 Provided Guarantees. In the light of the above attacks and countermeasures, our usage control infrastructure provides the following security guarantees.

End Users. In a number of scenarios end users do not have administrative privileges on the devices on which they store and process sensitive data. This includes employees of companies, governments, and other organizations, as well as most end users of today's integrated and embedded devices, mobile phones, and tablet computers. In any of these scenarios our solution prevents both intentional and accidental policy violations. Software and malware running on behalf of the end user falls into the same category.

For example, clerks might try to use and/or share 'quarantined' customer records (§1, policy 2), send out contract offers before they were approved (policy 3), or create additional copies of customer records (policy 4). By implementing the PEP at the system call layer, any such policy violation attempts can be detected independent of the applications being used. Through the use of both system-internal and cross-system data flow tracking, the end user is not able to obfuscate such attacks, e.g., by first creating local or remote copies or derivations of the protected data: any such data copies/derivations are detected, tracked, and protected just as the original data.

Administrators. If end users do have administrative privileges, our infrastructure reliably prevents illegitimate data usage as long as the user does not deliberately turn off the usage control infrastructure. For example, our infrastructure can prevent the user, misconfigured applications, or malware from misusing protected data or from releasing sensitive personal data on the Internet.

To defend against intentional attacks from users with administrative privileges, our infrastructure necessitates integration with protection mechanisms below the operating system layer. We discuss such technologies on the basis of trusted hardware in §7.

Summary. Our infrastructure prevents illegitimate data misuse and leakage as long as it is not deliberately disabled by administrators. We argue that these guarantees are sufficient in many scenarios: Large parts of the world's sensitive data is collected, processed and stored in environments in which end users are unable to disable security solutions that operate with administrative privileges at the operating system layer.

5.2 Cross-System Data Flow Tracking and Policy Propagation

In §5.2.1 we evaluate the performance of cross-system data flow tracking and policy propagation. §5.2.2 analyzes the precision of our data flow tracking system.

5.2.1 Performance. We evaluate the performance of cross-system data flow tracking and policy propagation by measuring (i) the maximum HTTPS request throughput of the Apache web server, (ii) the maximum network throughput of Iperf, and (iii) FTPS file transfer times of the Vsftpd FTP server. We evaluate both the preventive and detective policy enforcement mode of our infrastructure. Detective enforcement provides better performance, since the PEP can continue execution once an event was signaled to the `Controller`. Instead, for preventive enforcement the PEP must block upon all events and wait for a decision by the PDP. We also evaluate which parts of our infrastructure cause which overheads, thus separating (i) event processing and signaling by the PEP, and (ii) data flow tracking and policy propagation by the `Controller`.

Setup. We connected several machines via a 10Gbit network, each featuring a 2x2.6GHz CPU and 4GB RAM. Each machine ran Ubuntu 14.04 and a `Controller`. One machine hosted the server application (Apache, Iperf, Vsftpd) that was monitored by the PEP. The PEP signaled both intended and actual system calls to the local `Controller`. The

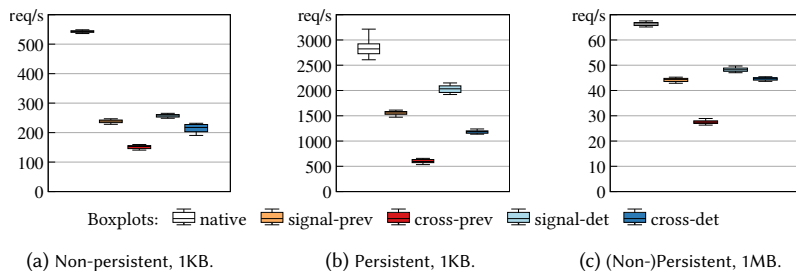(a) Non-persistent, 1KB.   (b) Persistent, 1KB.   (c) (Non-)Persistent, 1MB.

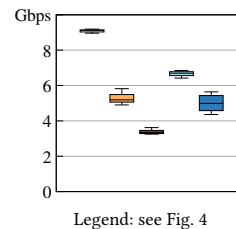Fig. 4. Apache throughput in req/s.

Fig. 5. Iperf bandwidth in Gbps.

clients ran benchmarks against the server, with their `Controllers` answering all of server's `Controller`'s requests as described in §4.2. In particular, this includes the remote communication for cross-system data flow tracking and policy propagation.

Apache HTTPS. We used the ApacheBench benchmark to request files of size 1KB and 1MB from Apache. We performed all experiments with both persistent and non-persistent connections. For all experiments we were limited by the server's CPU.

1KB Files. Using non-persistent connections, Apache serves 1KB files at a native baseline comparison rate of 543req/s (Fig. 4a, ⊟). Signaling events from the PEP to the `Controller` imposes a performance penalty for both preventive and detective policy enforcement. This is because of the expensive JNI communication between the PEP and the `Controller`. Signaling of events to the `Controller`, i.e., without further processing and data flow tracking, degrades performance to 239req/s (44.0% of native throughput) and 257req/s (47.3%) for preventive and detective enforcement, respectively (Fig. 4a, ⊟, ⊟). With data flow tracking and policy propagation, the respective resulting throughputs are 153req/s (28.2%) and 218req/s (40.1%) (Fig. 4a, ⊟, ⊟).

With persistent connections, native Apache achieves 2823req/s (Fig. 4b, ⊟). Event signaling for preventive and detective enforcement results in throughputs of 1556req/s (55.1%) and 2034req/s (72.0%), respectively (Fig. 4b, ⊟, ⊟). With data flow tracking and policy propagation, the respective resulting throughputs are 595req/s (21.1%) and 1188req/s (42.1%) (Fig. 4b, ⊟, ⊟).

1MB Files. For larger files of 1MB, persistent connections did not improve performance. We thus combine the results of persistent and non-persistent connections. Native Apache achieves 66.3req/s (Fig. 4c, ⊟). Event signaling for preventive and detective enforcement results in throughputs of 44.4req/s (66.8%) and 48.2req/s (72.7%), respectively (Fig. 4c, ⊟, ⊟). With data flow tracking and policy propagation, the respective resulting throughputs are 27.5req/s (41.4%) and 44.7req/s (67.3%) (Fig. 4c, ⊟, ⊟).

In sum, our infrastructure achieves 21%–41% throughput of native Apache when operating in preventive enforcement mode and 40%–44% when in detective mode.

Iperf Bandwidth. We use Iperf, a network throughput benchmark, to further measure the impact of our infrastructure on network throughput. Native Iperf achieves a throughput of 9.1Gbps (Fig. 5, ⊟). Event signaling for preventive and detective enforcement results in throughputs of 5.2Gbps (56.9% of native throughput) and 6.7Gbps (73.4%), respectively (Fig. 5, ⊟, ⊟). With data flow tracking and policy propagation, we achieve throughputs of 3.3Gbps (36.6%) and 5.0Gbps (54.7%) (Fig. 4a, ⊟, ⊟).

Order of boxplots: native signal-prev cross-prev signal-det cross-det

(a) 1KB file download.   (b) 1KB file upload.   (c) 16MB file download.   (d) 16MB file upload.
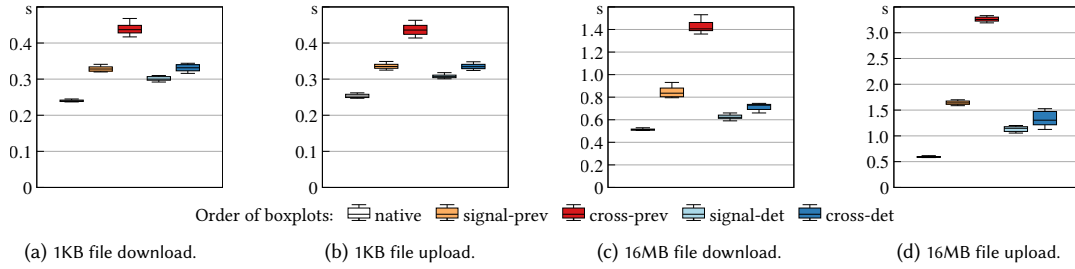
Fig. 6. Vsftpd file download and upload times in seconds.

Vsftpd FTPS. We evaluate the impact of our infrastructure on FTPS file transfer times by measuring download/upload times for 1KB and 16MB files from/to the Vsftpd server.

1KB Files. Since download and upload times for 1KB files are similar (see Fig. 6a and Fig. 6b), we only detail results for file downloads. Native Vsftpd achieves a baseline transfer time of 240ms (Fig. 6a, ⊟). Event signaling for preventive and detective enforcement results in file download times of 328ms (36.7% increased download time) and 300ms (25.0%), respectively (Fig. 6a, ⊟, ⊟). With data flow tracking and policy propagation, download times are 437ms (82.1%) and 331ms (38.1%) (Fig. 6a, ⊟, ⊟).

16MB Files. Native Vsftpd serves 16MB files in 513ms (Fig. 6c, ⊟). Event signaling for preventive and detective enforcement results in download times of 835ms (62.9%) and 621ms (21.2%), respectively (Fig. 6c, ⊟, ⊟). With data flow tracking and policy propagation, download times are 1,405ms (174%) and 729ms (42.1%) (Fig. 6c, ⊟, ⊟).

For 16MB files, native Vsftpd achieves upload times of 595ms (Fig. 6d, ⊟). The overhead imposed by our infrastructure is higher than for file downloads: Event signaling for preventive and detective enforcement results in upload times of 1,642ms (176%) and 1,147ms (92.9%), respectively (Fig. 6d, ⊟, ⊟). With data flow tracking and policy propagation, upload times are 3,252ms (446%) and 1,304ms (119%) (Fig. 6d, ⊟, ⊟). The higher overhead results from Vsftpd issuing approximately 3× more system calls when compared to file downloads. Since our infrastructure intercepts and analyses all of these system calls, this results in longer file transfer times.

Summary. On the basis of the above experiments, our overall infrastructure operates at a network throughput between 21% and 41% of native execution when performing cross-system data flow tracking and policy propagation in a preventive manner. We achieve a throughput of 40%–54% of native execution when operating in detective enforcement mode. File transfer times for FTPS increase by 82%–446% when in preventive mode and by 38%–119% for detective enforcement.

The evaluation shows that signaling of events from the PEP to the `Controller` amounts for large parts of the overhead. This is due to the JNI communication between the PEP, which is necessarily written in C, and the `Controller`, which was written in Java due to its complexity. The evaluation of Vsftpd shows that our infrastructure imposes less overhead if the monitored application issues less system calls. For I/O intensive applications, this overhead can be reduced by increasing the application's I/O buffer sizes, thereby decreasing the amount of required system calls. Overall, it depends on use case scenarios whether the imposed overheads are acceptable in practice.

5.2.2   Precision. We analyze the precision of data flow tracking by serving static and dynamic resources via an Apache web server. We use the Apache mpm_prefork module, which allows for the flexible configuration of Apache worker processes. With this module, Apache pre-forks a given amount of worker processes, each of which sequentially

handles incoming requests. For the experiment, we attach individual policies to the resources and observe how they propagate to the clients.

Static Resources. The first client requesting any static resource is served with exactly the policy that was attached to the requested resource. Further clients suffer from overapproximations: They are not only served the policy of the requested resource, but also a *subset* of the policies that were served to other clients before. The reason is that each worker process accumulates all policies of all resources that it has ever served. To ensure soundness, all of these policies are then always propagated to all clients.

There exist two immediate and simple means to remedy these overapproximations: (i) Configure Apache to kill each worker process after it served one connection (via the MaxConnectionsPerChild 1 directive). This, however, significantly impacts performance; (ii) Enrich the data flow propagation semantics of the *shutdown* system call (which shuts down socket connections, see [Kelbert and Pretschner 2013]) with minimal application specific semantics: If the process p calling *shutdown* on a socket is Apache, then clear the process' storage function, i.e. set $\sigma.s((y_p, p)) = \emptyset$. As a consequence, the following request will only be associated with policies of resources that are accessed from then on.

Dynamic Resources. We further set up an Apache/MySQL/PHP-based Drupal content management system to serve dynamic web pages. Serving such dynamic resources, we observe data flow tracking overapproximations beyond those of static resources. The reason is a combination of the following: (i) We associate policies with fine-grained Drupal data items such as articles or contact form submissions; (ii) Drupal stores this content within the MySQL database; (iii) MySQL stores all content of a database within a file; (iv) the system call PEP operates at a granularity of files. Consequently, whenever Drupal serves content from one particular MySQL database, the requesting client receives all policies associated with any data stored within that database.

While these overapproximations are an essential limitation of tracking data flows at the system call layer, specialized solutions compatible with our infrastructure have been proposed: Lienert [2012] formalizes and implements data flow tracking for MySQL at a granularity of single cells; Lovat et al. [2016] provide a more generic approach to track data flows across different system layers. Using information provided by such application-specific data flow tracking solutions, it is possible to improve the granularity of cross-system data flow tracking in the presence of complex software such as Drupal.

5.3 Distributed Policy Decisions

We evaluate the communication and performance overheads when decentrally enforcing global policies. For this, we enforce several policies on multiple systems simultaneously and compare the results with a centralized infrastructure.

Setup. We use eleven systems as detailed in §5.2 and a twelfth system with 16GB RAM. For the centralized setup, the latter hosts the one and only `Controller`. For the decentralized setup, each of the eleven former systems is equipped with one `Controller`.

The following parameters influenced the results: (i) the *policy* (policies 1, 2 and 3 from §2.1.3); (ii) the number of usage controlled systems ($3 \leq t \leq 11$); (iii) the number of systems enforcing the policy ($e \leq t$); (iv) the global event frequency in events per second ($f \leq 167\mathrm{Ev/s}$); (v) the percentage of events *relevant* for data flow tracking and/or policy evaluation ($0\% \leq r \leq 100\%$). For each measurement, we fixed all parameters and randomly generated event traces that matched f and r. Each event was randomly assigned to one of the t usage controlled systems and the policy was evaluated at every trigger event as well as for a timestep interval of one second. In the centralized case, all t systems must signal every event to the central `Controller` to take policy decisions (see §3). In the decentral case, the e policy-enforcing

(a) Five out of eleven systems enforce the policy.   (b) All eleven systems enforce the policy.
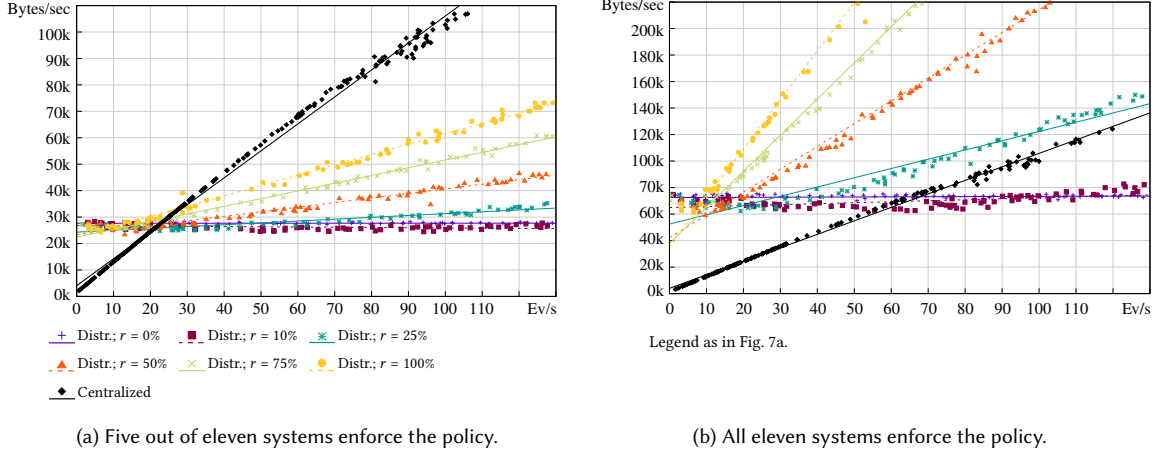
Fig. 7. Communication overhead when enforcing ECA rule 3.

systems evaluate the policy locally and coordinate those decisions with all other $e - 1$ policy-enforcing systems. The remaining $t - e$ systems do not evaluate any policy and do thus not exchange any information.

For the performance evaluation it was of importance whether the policy was evaluated upon actual or intended events. To obtain worst case results, we triggered the (expensive) distributed policy evaluation upon actual events. We obtain best case results by not triggering any policy evaluation upon intended events.

Different values for r reveal the effect of the optimizations proposed in §3.3. Without these, the results for our decentralized approach would coincide with $r = 100\%$.

We detail results for enforcing ECA rule 3 within eleven usage controlled systems ($t = 11$). Out of those, five or eleven systems enforce the policy ($e_1 = 5$, $e_2 = 11$). Results for other policies and amounts of systems are described in [Kelbert 2016]. While absolute numbers differ, the general trends are similar to what we describe here.

5.3.1 Communication Overhead. For the central system setup, each event caused 1170 Bytes to be exchanged between the PEP and the central `Controller`. The communication overhead is thus linear in the event frequency (Fig. 7, ◆). The percentage of relevant events r and the number of policy-enforcing systems e did not influence the results: as the PEPs are stateless, they must signal *every* event to the central PDP/PIP.

Fig. 7a and Fig. 7b show the communication overhead if five and eleven systems enforce the policy, respectively. First, Cassandra causes some base 'noise' to keep the database in a consistent state. Hence, the centralized approach performs inexorably better for very low event frequencies. The decentralized approach performs better the lower the percentage of relevant events: for low percentages of relevant events (■ ($r = 10\%$), ✳ (25%)), policies can often be conclusively evaluated locally, and few state changes must be coordinated. Traces with many relevant events (✕ (75%), ● (100%)), in contrast, necessitate the exchange of many state changes between PDPs.

Comparing Fig. 7a and Fig. 7b, we observe that the communication overhead of the decentralized approach is lower if less usage controlled systems do enforce the policy: If only five systems enforce the policy, then (i) events and state changes must only be coordinated between five systems; (ii) any events issued by the six remaining systems will never trigger any coordination. In sum: the lower the fraction of systems enforcing the policy, the less state changes must be
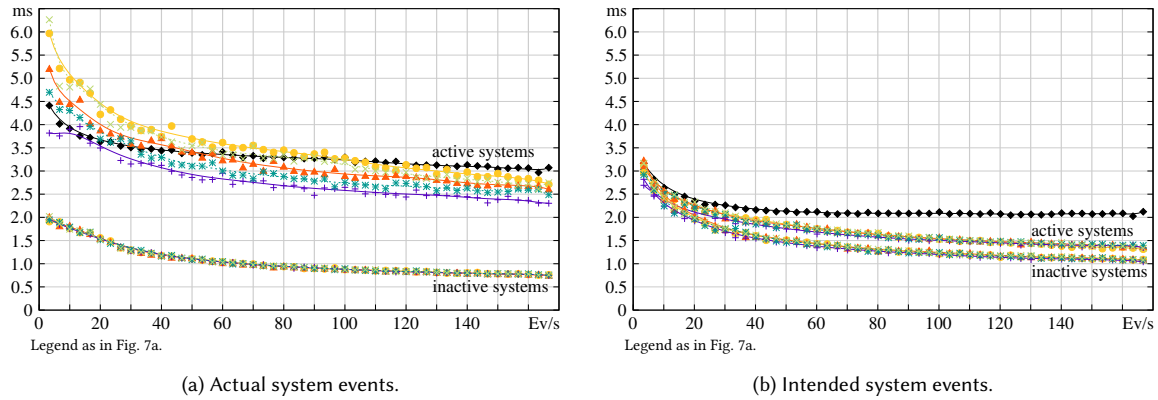
(a) Actual system events.

(b) Intended system events.

Fig. 8. Performance overhead when enforcing ECA rule 3.

coordinated across less systems. For policy 3 and $t = 11$, the decentralized approach performs better than the centralized approach if $e = 5 \wedge f \geq 30\text{Ev/s}$, or $e_2 = 11 \wedge r \leq 10\% \wedge f \geq 70\text{Ev/s}$, or $e_2 = 11 \wedge r \leq 25\% \wedge f \geq 140\text{Ev/s}$.

5.3.2 Performance Overhead. Fig. 8 shows the performance overhead for signaling events from the PEP to the `Controller` and their evaluation by the PDP/PIP. Fig. 8a shows the results for *actual* system events, while Fig. 8b shows those for *intended* system events. Since our policies were configured to be evaluated only for actual events, Fig. 8a yields worst case results, while Fig. 8b yields best case results. We do not further differentiate whether the policy was enforced by five or by eleven systems, as the performance overheads differed only insignificantly. Note: if the policy is enforced by all eleven systems, no inactive systems, as depicted in Fig. 8a and Fig. 8b, exist.

Fig. 8a shows that the performance of the distributed approach is bad when evaluating *actual* events within active systems (i.e., systems that actually enforce the policy) in combination with a low event frequency ($f \lesssim 50\text{Ev/s}$) and many relevant events (50% (▲), 75% (×), 100% (●)). In this case, many state changes must be written to the database, resulting in wait times for the acknowledgements of remote nodes. Some of this overhead is due to code warm up, as the code of both Cassandra and our infrastructure must be warmed up on eleven systems. For higher event frequencies ($f \gtrsim 80\text{Ev/s}$), the reliance on Cassandra is advantageous. The decentralized approach is benevolent towards inactive systems that do not enforce the policy: no remote communication is ever required for those systems and hence performance overheads are low.

Fig. 8b shows that for intended events the decentralized approach outperforms the centralized approach in all situations. In this case no policy evaluation was ever performed, making deployment of a decentralized approach particularly beneficial.

Summary. Communication and performance overheads of our decentralized infrastructure depend on three main factors: (i) the percentage of relevant events, (ii) the fraction of policy-enforcing systems, and (iii) the event frequency. As our experiments show, the decentralized approach performs best for (i) low percentages of relevant events: less relevant events cause less policy evaluations and thus less state changes to be coordinated; (ii) a low fraction of policy-enforcing systems, thus causing less relevant system events and thus less policy evaluations and state changes; and (iii) high event frequencies in combination with a low amount of relevant events.

6   ORTHOGONAL AND FUTURE WORK

Policy Specification and Deployment. We did not discuss particularities of policy specification, translation, and derivation in distributed environments, e.g., which entities should be allowed to specify, deploy, and modify policies for which data. Existing works [Hilty et al. 2007; Kumari and Pretschner 2013] focus on single systems only.

Integration with Solutions for Single Systems. Our approach is compatible with many solutions for single systems [Feth and Pretschner 2012; Harvan and Pretschner 2009; Pretschner et al. 2009, 2012; Wüchner and Pretschner 2012]. It remains to build an encompassing infrastructure that integrates these works, thus allowing for comprehensive intra-system and cross-system usage control enforcement.

Granularity of Data Flow Tracking. Tracking data flows on the basis of system calls comes at the cost of overapproximations, since data contents within processes, files, sockets, etc. can not be distinguished. Cross-system data flow tracking granularity could be improved by integrating with fine-grained data flow tracking at the application-layer.

Dedicated Solutions for Distributed Policy Decisions. Cassandra introduces communication overheads even in case no policy coordination is required. It stands to reason that a tailored implementation would significantly improve upon these overheads.

7   RELATED WORK

Usage control was first introduced by Park and Sandhu [2004], which proposed to evaluate authorizations, obligations, and environmental conditions both before and during the usage of an object. Hilty et al. [2005] generalize this concept to distributed systems, differentiating between observable and non-observable events. Their concepts lay the grounds for preventive and detective policy enforcement [Pretschner et al. 2007]. Hilty et al. [2007] introduce the Obligation Specification Language, while Pretschner et al. [2008] propose consumer-side enforcement mechanisms. Harvan and Pretschner [2009] and Pretschner et al. [2012] integrate these works with data flow tracking.

Cross-System Data Flow Tracking and Policy Propagation. Many approaches that track data flows across systems and propagate policies have limitations not present in our approach: (1) They are hard to deploy in commodity systems: DataSafe [Chen et al. 2012] builds upon custom hardware; CloudFilter [Papagiannis and Pietzuch 2012] and the work by Zhang et al. [2011] necessitate the adaptation of existing applications; Neon [Zhang et al. 2010] requires a particular hypervisor. (2) They are tailored to specific application(protocol)s, e.g. CloudFilter and the work by Janicke et al. [2012]. (3) They only support the propagation of labels rather than complex policies, including CloudFence and Taint-Exchange. (4) Lastly, Taint-Exchange, CloudFence, and Neon are limited in the number of distinct labels or policies that can be tracked.

Decentralized information flow control (DIFC) [Myers and Liskov 1997] prevents information flows from trusted to untrusted data locations. After labelling data and data locations, data may only flow towards higher-labeled locations. Zeldovich et al. [2008] track labels across hosts similar to our approach and determine whether cross-system data flows are permissible. Data usage control policies can express similar constraints; DIFC mechanisms and their lattice model can be integrated via operator *evalExt*.

Distributed Policy Decisions. Existing works that aim for the decentralized enforcement of policies impose limitations not present in our approach: (1) For some, the decision process is not entirely distributed, as they rely on single components: E.g., Gay et al. [2012] delegate decisions whenever local knowledge is insufficient. Their static mapping of conflicting events to the same automaton introduces a single point of failure. Lazouski et al. [2014] fix one PDP/PIP for the entire lifetime of each data. (2) Chadwick et al. [2008] and Alzahrani et al. [2010] do not consider dynamic data propagation, thus protecting only static resources. (3) Basin et al. [2014; 2013; 2015] and Bauer and Falcone [2012] focus

on the detection of policy violations. Our approach, instead, allows for both preventive and detective enforcement. Similar to our approach, Basin et al. identify situations in which the evaluation of partial logs suffices because their satisfaction/violation implies satisfaction/violation of the overall log.

Further Approaches. SafeShare [Thilakanathan et al. 2013] introduces encrypted containers containing both protected data and policies. Once a user accesses the container, a background process starts and monitors compliance with the policy.

xDUCON Russello and Dulay [2009] enforces data sharing agreements across organizations. It supports ongoing conditions and may revoke access rights during long-lived accesses. Sensitive resources and policies are shared and coordinated via shared storage.

FlowwolF [Hicks et al. 2010] enforces Mandatory Access Control (MAC) for web applications at multiple layers of the software stack. The solution integrates DIFC, MAC reference monitors, and cross-system policy propagation. Our solution is independent of the application layer and enforces more expressive policies even across systems.

Securing Data Usage Control Infrastructures. Solutions to secure infrastructures as ours build upon the Trusted Platform Module (TPM), measuring the platform's components and the usage control infrastructure at boot time. If the measurements differ from a set of known good values, Zhang et al. [2008] disallow access to protected data.

Agreiter et al. [2007] address distributed aspects: Data providers only release protected data after data requestors proved the presence of 'good' TPM measurements. Neisse et al. [2011a] detect modifications of remote cloud infrastructure configurations.

UCLinux [Kyle and Brustoloni 2007] stores usage controlled files and policies on an encrypted file system. The decryption key is sealed to a trusted TPM configuration, ensuring that it is only accessible to a valid software stack. UCLinux uses a modified version of TLS to perform remote attestation upon data distribution.

Digital Rights Management (DRM, e.g. [Liu et al. 2003]) influenced early usage control work by Park and Sandhu [2004] due to the insight that DRM is insufficient if digital media is not only consumed but also re-used, combined, and extended [Iannella 2000].

DRM primarily aims at the protection of copyrighted content which is encrypted upon distribution. Once a user obtains a license, the content is decrypted. Client-side applications enforce compliance with granted usage rights and payment obligations.

The proprietary nature of DRM hinders interoperability and leads to lock-in effects. This was addressed by Koenen et al. [2004] and Taban et al. [2006]. Recently, HTML5 provides interoperable streaming of protected media. Usage control is not limited to proprietary file types and enables the re-use, combination, and extension of data.

8 CONCLUSIONS

Recently, many works addressed the enforcement of data usage control policies within individual systems. However, the enforcement of policies that refer to data and data usage events that are distributed has not been researched in depth. We overcome the drawbacks of a centralized infrastructure by providing a model, an architecture, and an implementation of a fully decentralized usage control enforcement infrastructure.

Our model allows for the explicit distinction of different systems, their individual behaviors, as well as their interplay. We showed how these individual observations can be reassembled in order to mimic the behavior of classical centralized systems.

We modeled and implemented the tracking of data flows across systems in a manner that is independent of file types, applications, protocols, and hardware. By leveraging system call interpositioning techniques, we avoid modifications to applications and the operating system, achieving a throughput of 21%–54% of native execution.

Further, we modeled and implemented mechanisms that allow for the fully decentralized, efficient, and preventive enforcement of global data usage policies. We identify systems that are potentially relevant to enforce a given policy, as well as situations in which coordination of policy decisions can be safely omitted. The evaluation revealed that—depending on concrete parameters of the scenario considered—our decentralized approach is able to outperform a centralized approach.

REFERENCES

Berthold Agreiter, Muhammad Alam, Ruth Breu, Michael Hafner, Alexander Pretschner, Jean-Pierre Seifert, and Xinwen Zhang. 2007. A Technical Architecture for Enforcing Usage Control Requirements in Service-oriented Architectures. In *Proc. Workshop on Secure Web Services*. ACM, 18–25.

Ali Alzahrani, Helge Janicke, and Sarshad Abubaker. 2010. Decentralized XACML Overlay Network. In *IEEE 10th Intl. Conf. on Computer and Information Technology*. 1032–1037.

David Basin, Germano Caronni, Sarah Ereth, Matúš Harvan, Felix Klaedtke, and Heiko Mantel. 2014. Scalable Offline Monitoring. In *Runtime Verification*. LNCS, Vol. 8734. Springer Intl. Publishing, 31–47.

David Basin, Matúš Harvan, Felix Klaedtke, and Eugen Zălinescu. 2013. Monitoring Data Usage in Distributed Systems. *IEEE Transactions on Software Engineering* 39, 10 (2013), 1403–1426.

David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. 2015. Monitoring Metric First-Order Temporal Properties. *J. ACM* 62, 2, Article 15 (2015), 15:1–15:45 pages.

Andreas Bauer and Yliès Falcone. 2012. Decentralised LTL Monitoring. In *FM 2012: Formal Methods*. LNCS, Vol. 7436. Springer, 85–100.

David W. Chadwick, Linying Su, and Romain Laborde. 2008. Coordinating Access Control in Grid Services. *Concurrency and Computation: Practice & Experience* 20, 9 (2008), 1071–1094.

Yu-Yuan Chen, Pramod A. Jamkhedkar, and Ruby B. Lee. 2012. A Software-Hardware Architecture for Self-Protecting Data. In *Proc. Conf. on Computer and Communications Security*. ACM, 14–27.

Isao Echizen, Takayuki Yamada, and Seiichi Gohshi. 2015. *IR Hiding: Use of Specular Reflection for Short-Wavelength-Pass-Filter Detection to Prevent Re-recording of Screen Images*. Springer, 38–54.

Denis Feth and Alexander Pretschner. 2012. Flexible Data-Driven Security for Android. In *6th Intl. Conf. on Software Security and Reliability*. 41–50.

Richard Gay, Heiko Mantel, and Barbara Sprick. 2012. Service Automata. In *Formal Aspects of Security and Trust*. LNCS, Vol. 7140. Springer, 148–163.

Matús Harvan and Alexander Pretschner. 2009. State-Based Usage Control Enforcement with Data Flow Tracking using System Call Interposition. In *Third Intl. Conf. on Network and System Security*. 373–380.

Boniface Hicks, Sandra Rueda, Dave King, Thomas Moyer, Joshua Schiffman, Yogesh Sreenivasan, Patrick McDaniel, and Trent Jaeger. 2010. An Architecture for Enforcing End-to-end Access Control over Web Applications. In *Proc. 15th ACM Symposium on Access Control Models and Technologies*. ACM, 163–172.

Manuel Hilty, David Basin, and Alexander Pretschner. 2005. On Obligations. In *ESORICS 2005*. LNCS, Vol. 3679. Springer, 98–117.

Manuel Hilty, Alexander Pretschner, David Basin, Christian Schaefer, and Thomas Walter. 2007. A Policy Language for Distributed Usage Control. In *ESORICS 2007*. LNCS, Vol. 4734. Springer, 531–546.

Renato Iannella. 2000. *Open Digital Rights Management*. Technical Report. IPR Systems Pty Ltd. A Position Paper for the W3C DRM Workshop.

Helge Janicke, Antonio Cau, François Siewe, and Hussein Zedan. 2008. Concurrent Enforcement of Usage Control Policies. In *IEEE Workshop on Policies for Distributed Systems and Networks*. 111–118.

Helge Janicke, Mohamed Sarrab, and Hamza Aldabbas. 2012. Controlling Data Dissemination. In *Data Privacy Management and Autonomous Spontaneus Security*. LNCS, Vol. 7122. Springer, 303–309.

Florian Kelbert. 2016. *Data Usage Control for Distributed Systems*. Dissertation. Technical University of Munich, Garching b. München, Germany.

Florian Kelbert and Alexander Pretschner. 2013. Data Usage Control Enforcement in Distributed Systems. In *Proc. Third ACM Conf. on Data and Application Security and Privacy*. ACM, 71–82.

Florian Kelbert and Alexander Pretschner. 2014. Decentralized Distributed Data Usage Control. In *Cryptology and Network Security (LNCS)*, Vol. 8813. Springer Intl. Publishing, 353–369.

Florian Kelbert and Alexander Pretschner. 2015. A Fully Decentralized Data Usage Control Enforcement Infrastructure. In *Applied Cryptography and Network Security*. LNCS, Vol. 9092. Springer, 409–430.

Rob H. Koenen, Jack Lacy, Michael Mackay, and Steve Mitchell. 2004. The Long March to Interoperable Digital Rights Management. *Proc. IEEE* 92, 6 (2004), 883–897.

Paul Kranenburg and Dmitry Levin. 2015. strace. (2015). http://sourceforge.net/projects/strace/

Prachi Kumari and Alexander Pretschner. 2013. Model-Based Usage Control Policy Derivation. In *Engineering Secure Software and Systems*. LNCS, Vol. 7781. Springer.

Prachi Kumari, Alexander Pretschner, Jonas Peschla, and Jens-Michael Kuhn. 2011. Distributed Data Usage Control for Web Applications: A Social Network Implementation. In *Proc. First ACM Conf. on Data and Application Security and Privacy*. ACM, 85–96.

David Kyle and José Carlos Brustoloni. 2007. UCLinux: a Linux Security Module for Trusted-Computing-based Usage Controls Enforcement. In *Proc. Workshop on Scalable Trusted Computing*. ACM, 63–70.

Aliaksandr Lazouski, Gaetano Mancini, Fabio Martinelli, and Paolo Mori. 2014. Architecture, Workflows, and Prototype for Stateful Data Usage Control in Cloud. In *IEEE Security & Privacy Workshops*. 23–30.

Orna Lichtenstein, Amir Pnueli, and Lenore Zuck. 1985. The Glory of the Past. In *Logics of Programs*. LNCS, Vol. 193. Springer, 196–218.

Daniel Lienert. 2012. *Distributed Usage Control for the MySQL Database Server*. Diploma Thesis. Karlsruhe Institute of Technology, Germany.

Qiong Liu, Reihaneh Safavi-Naini, and Nicholas Paul Sheppard. 2003. Digital Rights Management for Content Distribution. In *Proc. Australasian Information Security Workshop Conf. on ACSW Frontiers 2003 - Volume 21*. Australian Computer Society, Inc., 49–58.

Michael Lörscher. 2012. *Data Usage Control for the Thunderbird Mail Client*. Master's thesis. University of Kaiserslautern, Germany.

Enrico Lovat and Florian Kelbert. 2014. Structure Matters – A new Approach for Data Flow Tracking. In *IEEE Security and Privacy Workshops*.

Enrico Lovat, Martín Ochoa, and Alexander Pretschner. 2016. Sound and Precise Cross-Layer Data Flow Tracking. In *Engineering Secure Software and Systems: 8th International Symposium*. Springer, 38–55.

Enrico Lovat, Johan Oudinet, and Alexander Pretschner. 2014. On Quantitative Dynamic Data Flow Tracking. In *Proc. 4th ACM Conf. on Data and Application Security and Privacy*. ACM, 211–222.

Andrew C. Myers and Barbara Liskov. 1997. A Decentralized Model for Information Flow Control. *SIGOPS Oper. Syst. Rev.* 31, 5 (Oct. 1997), 129–142.

Ricardo Neisse, Dominik Holling, and Alexander Pretschner. 2011a. Implementing Trust in Cloud Infrastructures. In *Proc. 11th Intl. Symp. on Cluster, Cloud and Grid Computing*. IEEE, 524–533.

Ricardo Neisse, Alexander Pretschner, and Valentina Di Giacomo. 2011b. A Trustworthy Usage Control Enforcement Framework. In *6th Intl. Conf. on Availability, Reliability and Security*. 230–235.

Ioannis Papagiannis and Peter Pietzuch. 2012. CloudFilter: Practical Control of Sensitive Data Propagation to the Cloud. In *Proc. Workshop on Cloud Computing Security Workshop*. ACM, 97–102.

Jaehong Park and Ravi Sandhu. 2004. The UCON$_{ABC}$ Usage Control Model. *ACM Transactions on Information and System Security* 7, 1 (2004), 128–174.

Alexander Pretschner, Matthias Büchler, Matúš Harvan, Christian Schaefer, and Thomas Walter. 2009. Usage Control Enforcement with Data Flow Tracking for X11. In *5th Intl. Workshop on STM*.

Alexander Pretschner, Manuel Hilty, and David Basin. 2006. Distributed Usage Control. *Commun. ACM* 49, 9 (2006), 39–44.

Alexander Pretschner, Manuel Hilty, David Basin, Christian Schaefer, and Thomas Walter. 2008. Mechanisms for Usage Control. In *Proc. Symp. on Information, Computer and Communications Security*. ACM, 5.

Alexander Pretschner, Enrico Lovat, and Matthias Büchler. 2012. Representation-Independent Data Usage Control. In *Data Privacy Management and Autonomous Spontaneus Security*. LNCS, Vol. 7122. Springer.

Alexander Pretschner, Fabio Massacci, and Manuel Hilty. 2007. Usage Control in Service-Oriented Architectures. In *Trust, Privacy and Security in Digital Business*. LNCS, Vol. 4657. Springer, 83–93.

Giovanni Russello and Naranker Dulay. 2009. xDUCON: Coordinating Usage Control Policies in Distributed Domains. In *Third Intl. Conf. on Network and System Security*. 246–253.

Gelareh Taban, Alvaro A. Cárdenas, and Virgil D. Gligor. 2006. Towards a Secure and Interoperable DRM Architecture. In *Proc. ACM Workshop on Digital Rights Management*. ACM, 69–78.

Danan Thilakanathan, Rafael Calvo, Shiping Chen, and Surya Nepal. 2013. Secure and Controlled Sharing of Data in Distributed Computing. In *16th Intl. Conf. on Computational Science and Engineering*. 825–832.

Tobias Wüchner and Alexander Pretschner. 2012. Data Loss Prevention Based on Data-Driven Usage Control. In *IEEE 23rd Intl. Symp. on Software Reliability Engineering*. 151–160.

Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. 2008. Securing Distributed Systems with Information Flow Control. In *Proc. 5th USENIX Symp. on Networked Systems Design and Implementation*.

Olive Qing Zhang, Markus Kirchberg, Ryan KL Ko, and Bu Sung Lee. 2011. How to Track Your Data: The Case for Cloud Computing Provenance. In *3rd Intl. Conf. on Cloud Computing Technology and Science*.

Qing Zhang, John McCullough, Justin Ma, Nabil Schear, Michael Vrable, Amin Vahdat, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. 2010. Neon: System Support for Derived Data Management. In *Proc. 6th ACM Intl. Conf. on Virtual Execution Environments*. ACM, 63–74.

Xinwen Zhang, Jean-Pierre Seifert, and Ravi Sandhu. 2008. Security Enforcement Model for Distributed Usage Control. *Intl. Conf. on Sensor Networks, Ubiquitous, and Trustworthy Computing* (2008), 10–18.