A Fully Decentralized Data Usage Control Enforcement Infrastructure

Florian Kelbert and Alexander Pretschner

Technische Universität München, Germany
{kelbert,pretschn}@cs.tum.edu

Abstract. Distributed data usage control enables data owners to constrain how their data is used by remote entities. However, many data usage policies refer to events happening within several distributed systems, e.g. "at each point in time at most two clerks might have a local copy of this contract", or "a contract must be approved by at least two clerks before it is sent to the customer". While such policies can intuitively be enforced using a centralized infrastructure, major drawbacks are that such solutions constitute a single point of failure and that they are expected to cause heavy communication and performance overhead. Hence, we present the first fully decentralized infrastructure for the preventive enforcement of data usage policies. We provide a thorough evaluation of our infrastructure and show in which scenarios it is superior to a centralized approach.

1 Introduction

Due to the ever increasing value of data, the continuous protection of sensitive data throughout its entire lifetime has drawn much attention in recent years. Corresponding solutions are applicable in many contexts: businesses, military and governments aim at protecting their internal procedures, research reports, financial reports, and the like; individuals want to constrain businesses from using or releasing their private data, e.g. for advertisement or market research; copyright owners want their licenses to be respected.

Usage control [1, 2] tackles such challenges by proposing different models and enforcement infrastructures [3–6]. Generally, policies describe how data may or may not be used once initial access has been granted. Additionally, policies might specify obligations that must be fulfilled before, upon, or after usage. Corresponding solutions [7–10] inject reference monitors, or Policy Enforcement Points (PEP), into different layers of the computing system. These PEPs intercept events within the system and enforce the Policy Decision Point's (PDP) decision such as allowing, modifying, inhibiting or delaying the event. By tracking *data flows*, such as when copying files or loading content from a database into a process, aforementioned solutions allow to enforce data usage policies on all representations of some data rather than on particular files or database entries. Hence, data usage policies are enforced independently of the data's concrete *representations* at runtime. Enforcement may be *preventive* or *detective* [1, 6], meaning that policy violations never occur, or that they can be detected in hindsight, respectively.

This work tackles the problem of enforcing data usage policies on data that has been disseminated to remote systems. In this respect, solutions that track data flows across

In: Proc. 13th International Conference on Applied Cryptography and Network Security (ACNS), Springer LNCS 9092, pages 409–430, June 2015, New York City, New York, USA.
The final publication is available at
http://link.springer.com/chapter/10.1007%2F978-3-319-28166-7_20.

systems and attach the corresponding policies have been proposed [11, 12]. Further, these solutions enable the enforcement of policies that can be independently evaluated on every single system, such as "do not open this document with editor X", or "do not print this document after 5pm". However, the preventive enforcement of more sophisticated *global policies* pertaining to events and/or the states of multiple systems, such as "not more than five instances of this software might be executed simultaneously", or "all copies of this document must be deleted upon the owner's demand", still poses challenges [6, 13, 14]. We are not aware of solutions that achieve preventive policy enforcement (i) without the need for any central components, (ii) on all copies and derivations of the original data, and (iii) which are deployable on commodity networks.

While Digital Rights Management solutions handle such challenges by deploying central license servers [15], such a solution comes with the drawbacks of being a single point of failure, privacy concerns, and the necessity that the central component must be always reachable by all PEPs. Moreover, a centralized solution is expected to impose significant performance and communication overhead [13, 16]. The main reason is that the PEP is stateless. Hence, whenever a potentially relevant system event is observed by the PEP, it is unknown whether it is of actual importance for evaluation by the PDP. Consequently, all observed events would need to be signaled to the central PDP. While recent works addressed this problem by decentralizing some aspects of policy evaluation, data flow tracking, and/or information retrieval [6, 8, 17, 18], some of them do not allow for preventive policy enforcement [6, 18], while others effectively make use of central components [8, 17], or do not integrate data flow tracking [8, 17, 18].

Problem. We tackle the problem of *enforcing global data usage control policies* if (i) the data to be protected resides, (ii) the data usage events occur, and (iii) the data flow events occur within and across multiple distributed systems. While a solution could naively be implemented in a centralized fashion, such a solution imposes drawbacks such as being a single point of failure. Intuitively, a centralized solution is also expected to impose significant performance and network communication overhead [13, 16].

Solution. We design and implement a fully decentralized enforcement infrastructure with the goal to minimize aforementioned drawbacks and overheads. This infrastructure deploys one PDP at each site which takes all decisions pertaining to all local PEPs. Global policies are enforced by synchronizing the PDPs using a distributed database. We optimize the information being exchanged according to theoretical results [13].

Contribution. To the best of our knowledge, our contributions are:

1. The first fully decentralized architecture and implementation for the preventive enforcement of global data usage control policies (§3).
2. A thorough evaluation of the proposed and implemented architecture, showing in which scenarios its adoption is beneficial (§4).

Further, we provide the source code of our implementation as open source[1].

Attacker Model and Assumptions. Our infrastructure prevents users from using data in a way that does not comply with the corresponding policy—be the attempt intentional

[1] https://github.com/fkelbert/uc/ and https://github.com/fkelbert/uc4linux/.

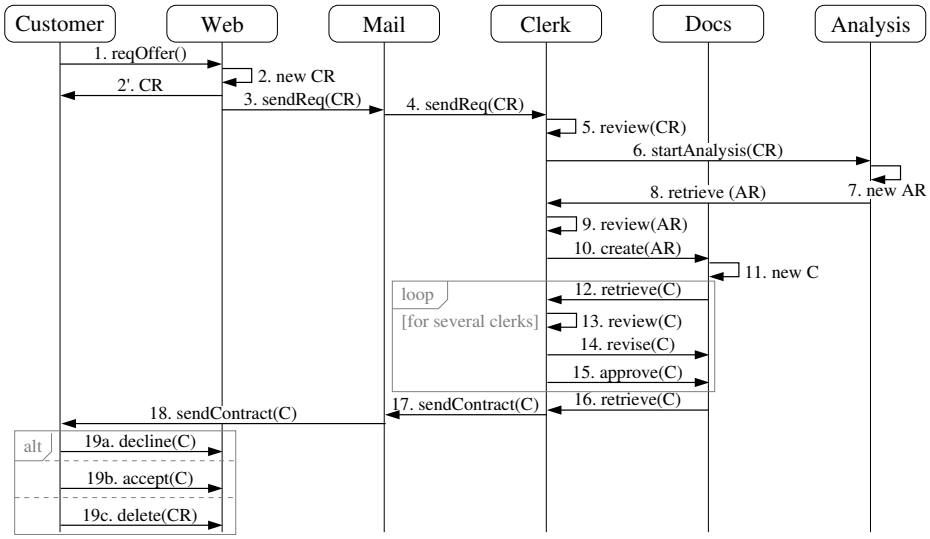

Fig. 1: Sequence of events in the running example.

or unintentional. Foremost, we consider users without administrative privileges. Such a scenario is pervasive in business environments, where employees are given ready-to-use computing systems. To defend against stronger attackers, the trust anchor must be embedded at a lower layer, e.g. by using TPMs or SmartCards. Since our infrastructure runs as a process within the operating system, we assume both to be free of vulnerabilities. Otherwise, an attacker might be able to gain administrative privileges and switch off our infrastructure and/or tamper with it. Moreover, we assume state-of-the-art access control mechanisms to be in place.

Running Example. We illustrate our work along a running example, in which an insurance company provides potential customers the ability to request contract offers via a web interface. After internal processing of the request, the customer retrieves a contract offer via email, which may be accepted or declined via a web link. The entire scenario, including the insurance provider's internal data processing, is depicted in Fig. 1.

First, the customer fills a web form on the insurance provider's website. By submitting the form (1), a new ContractRequest (CR) object is created (2) and the web server sends the CR to a set of clerks via the mail server (3,4). One of the clerks will then review the attached CR (5) and start an analysis job on the internal data analysis server (6), thereby creating a new AnalysisResult (AR) object (7). Once the analysis is performed, the clerk retrieves the AR (8) and performs a manual review on her workstation (9). The clerk then creates a Contract (C) object using a collaborative word processor (10,11). Once created, C might be retrieved (12), reviewed (13) and revised (14) by several clerks. After C has been approved by a predefined number of clerks (15), one of the clerks retrieves its final version (16) and sends it to the customer via the mail server (17,18). Once the customer receives the offer, he might decline (19a) or accept (19b) the Contract. Alternatively, he might delete his initial ContractRequest altogether (19c).

Besides the application-specific events mentioned above, we also consider events originating at the operating system layer, i.e. system calls [10]. Using such an approach, we are able to detect data flows that happen outside the application context or that have not been anticipated within the application context, e.g. if a clerk creates a copy of a Contract using a file manager or a shared file system.

Clearly, the customer's data flows through many different systems in many different formats. Further, the AnalysisResult and the Contract are data items that have been derived from the original ContractRequest and must as such be treated as containing the customer's personal data. All of these data items are stored and processed by many different systems and users, all of which must enforce data usage policies such as:

Policy 1: 'Exactly one contract offer must be sent to the customer not later than 30 days after a request has been received.'
Policy 2: 'If the customer declines an offer, all derived data items must not be used anymore.'
Policy 3: 'Each contract must be reviewed and approved by at least two clerks.'
Policy 4: 'At no point in time might two clerks have a copy of the same analysis result.'

Note, that all of those policies are *global policies*, meaning that they refer to data and events that are distributed across several systems.

2 Background

2.1 Existing Data Usage Control Infrastructures

Data usage control infrastructures have been built for various system layers and scenarios [4, 6–10, 12], and policy enforcement is usually performed using a PEP, a PDP, and a Policy Information Point (PIP). Once the PEP observes an attempt of using an object, this attempt is signaled to the PDP which is configured with the policies to be enforced. Depending on these policies, its internal state, and additional information from the PIP, the PDP decides whether to allow, inhibit, modify, or delay the usage attempt. The PEP is then in charge of enforcing the decision. The information provided by the PIP differs slightly in different models and includes subject and object attributes, environmental information, and details about which data takes which representations within the system, i.e. the system's *data flow state*.

The set of events intercepted by the PEP is categorized into two, possibly overlapping, subsets: *data usage events* and *data flow events*. Informally, data usage events are events whose occurrence is obliged or constrained by data usage policies. As such, all data usage events must be signaled to the PDP. Data flow events, in contrast, must be signaled to the PIP. According to an event's predefined semantics and its actual parameters, the PIP will update its data flow state. For example, if a ContractRequest data item is known to be stored as a database entry, then all result sets of database queries selecting this entry will also be associated with the same ContractRequest data item, and hence with the same data usage policies.

Using such a combination of policy enforcement and data flow tracking technology, data usage control infrastructures allow to not only protect one single data representation, such as a file or database entry, but rather all representations of the same data.

To differentiate between *detective* and *preventive* enforcement, the distinction between *desired events* and *actual events* is needed. Desired events are intercepted by PEPs *before* their execution and they may be inhibited or modified in correspondence with the PDP's decision. Actual events are intercepted by the PEP *after* their execution. They can not be inhibited or modified, but only be compensated for. Thus, desired events must be intercepted and evaluated for preventive enforcement, while actual events must be monitored because they cause state changes within the PDP and PIP.

2.2 Data Usage Control Policies: Syntax, Semantics, Evaluation

Building upon previous works [5, 7, 13, 19], we assume policies to be specified as **E**vent-**C**ondition-**A**ction (ECA) rules: once a triggering **E**vent is observed and if the execution of this event would make the **C**ondition true, then additional **A**ctions might be performed. Notably, the triggering event might also be an artificial event, e.g. to indicate that a certain amount of time has passed. We will use the terms 'policy' and 'ECA rule' interchangeably. ECA conditions (Φ) are specified in terms of past linear temporal logics and their syntax is specified as:

$$\Psi = \underline{true} \mid \underline{false} \mid \mathcal{E}$$
$$\Sigma = \underline{isNotIn}(\mathcal{D}, \mathbb{P}(C)) \mid \underline{isCombined}(\mathcal{D}, \mathcal{D}, \mathbb{P}(C)) \mid \underline{isMaxIn}(\mathcal{D}, \mathbb{N}, \mathbb{P}(C))$$
$$\Phi = (\Phi) \mid \Psi \mid \Sigma \mid \underline{not}(\Phi) \mid \Phi\ \underline{and}\ \Phi \mid \Phi\ \underline{or}\ \Phi \mid \Phi\ \underline{since}\ \Phi \mid \Phi\ \underline{before}\ \mathbb{N} \mid \underline{repmin}(\mathbb{N}, \mathbb{N}, \mathcal{E})$$

While the formal semantics of Φ are detailed in [13], we recap the intuitive semantics: $\mathcal{E}$ denotes the set of all data usage events (cf. §2.1); $\mathcal{D}$ denotes the set of data items to be protected; C denotes the set of all possible representations, or *containers*, for data, such as files and database entries. Ψ refers to boolean constants ($\underline{true}$, $\underline{false}$) and data usage events $\mathcal{E}$. Σ refers to so-called state-based operators, allowing to express constraints over the system's data flow state as computed and maintained by the PIP: $\underline{isNotIn}(d, C)$ is true iff data d is not in any of the containers C; $\underline{isCombined}(d_1, d_2, C)$ is true iff there is at least one container in C that contains both data d_1 and d_2; $\underline{isMaxIn}(d, m, C)$ is true iff data d is contained in at most m containers in C. For Φ, the semantics of $\underline{not}$, $\underline{and}$ and $\underline{or}$ are intuitive; $\alpha\ \underline{since}\ \beta$ is true iff β was true some time earlier and α was true ever since, or if α was always true; $\alpha\ \underline{before}\ j$ is true iff α was true exactly j timesteps ago; $\underline{repmin}(j, m, e)$ is true iff event e happened at least m times in the last j timesteps. Further, we define $\underline{repmax}(j, m, e) \equiv \underline{not}(\underline{repmin}(j, m+1, e))$ and $\underline{always}(\alpha) \equiv \alpha\ \underline{since}\ \underline{false}$.

Fixing one data item d, Table 1 shows the example policies from §1 as ECA rules. Rule 1a expresses that the CEO must be notified via mail if no contract offer has been sent to the customer 30 days after a corresponding request. Note that this rule does have a wildcard trigger event, implying that the rule is evaluated upon every event. Further, this rule is detective only: satisfaction of the condition results in a compensating action; actual violation of the policy is not prevented. Rule 1b expresses that a contract offer must not be sent if there was no corresponding contract request, or if a contract offer was already sent. Rule 2 expresses that any attempt to use data item d is inhibited if the corresponding contract offer was declined in the past. Note, that we have used event *use* to refer to a set of events. This set might include events such as AnalysisServer.start, Docs.create and Mail.sendContract. Rule 3 expresses that sending of a contract is inhibited if this contract was not reviewed or approved by at least two clerks in the last 30

Policy 1	**E**vent:	*<any>*
(a)	**C**ondition:	(*Web.reqOffer*(*d*) *before* 30) *and* *repmax*(30, 0, *Mail.sendContract*(*d*))
	Action:	*Mail.notifyCEO*(*d*)
	Event:	*Mail.sendContract*(*d*)
(b)	**C**ondition:	*repmax*(30, 0, *Web.reqOffer*(*d*)) *or* *repmin*(30, 1, *Mail.sendContract*(*d*))
	Action:	*inhibit*
Policy 2	**E**vent:	*use*(*d*)
	Condition:	*not*(*always*(*not*(*Web.decline*(*d*))))
	Action:	*inhibit*
Policy 3	**E**vent:	*Mail.sendContract*(*d*)
	Condition:	*repmax*(30, 1, *Workstation.review*(*d*)) *or* *repmax*(30, 1, *Docs.approve*(*d*))
	Action:	*inhibit*
Policy 4	**E**vent:	*<any>(d)*
	Condition:	*not*(*isMaxIn*(*d*, 1, $C_{\text{Workstation}}$))
	Action:	*inhibit*

Table 1: Example policies as ECA rules.

days. Rule 4 expresses that any event must be inhibited if its execution would lead to a state in which data *d* is in more than one of the clerk's workstations.

Policy Evaluation. A policy is evaluated whenever a trigger event occurs or if a predefined amount of time has passed. The amount of time is configurable per policy and the interval between two subsequent time-based policy evaluations is called a *timestep*. The introduction of timesteps is necessary for practical reasons: If an ECA condition such as φ = (*Web.reqOffer*(*d*) *before* 30[*days*]) is to be evaluated, then it is unlikely that event *Web.reqOffer*(*d*) has happened *exactly* 30 days (i.e. 2592000 seconds) ago. What is more likely and practical, however, is that *Web.reqOffer*(*d*) has happened 'approximately' 30 days ago, e.g. 30days ± 12hours. Similarly, consider the conjunction and disjunction of operators, *and* and *or*. While it is unlikely that two events happen at *exactly* the same point in time, what is more likely and practical is that two events happen within a specified time interval, i.e. within the same timestep.

For policy evaluation purposes, we consider conditions $\varphi \in \Phi$ as expression trees. Leaves represent the constants *true* and *false*, events $\mathcal{E}$, and state-based operators Σ; internal nodes are operators such as *before*, *since*, *and*, etc. Fig. 2 depicts ECA rule 1a as expression tree. Leaves are stateful by storing whether the represented operand has become *true* or *false*, depending on the actual operand, during the current timestep. Whenever a PEP signals an event to the PDP, it is evaluated against all of φ's leaves, potentially changing their states. E.g., if a leaf represents the event Mail.sendContract, then this leaf's state changes to *true* once the PEP signals event Mail.sendContract. If a leaf corresponds to a state-based operator Σ, then its state is examined with the help of the PIP under consideration of the signaled event's data flow semantics. In a nutshell, the expression trees' leaves track which events have happened and which state-based operators have changed their state during the ongoing timestep.

Only if the event signaled by the PEP matches the ECA rule's trigger event, then the entire condition φ is evaluated, denoted $eval(\varphi)$. For this, the expression tree's internal nodes recursively query their child nodes for their current state. Subsequently, the internal nodes are evaluated using this information. Internal nodes also maintain a state, capturing historical values of child nodes. E.g. if $\varphi = (Web.reqOffer(d)\ \underline{before}\ 30[days])$, then *before* will keep a history of occurrences of $Web.reqOffer(d)$ for 30 days.

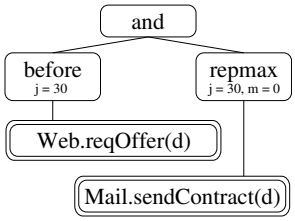

Fig. 2: Expression tree of ECA rule 1a.

If $eval(\varphi) = true$, then the ECA's actions will be triggered. Notably, evaluation of a condition $\varphi \in \Phi$ at the end of a timestep is different in that the leaves' evaluation results correspond to the truth values that have been 'accumulated' during the elapsed timestep: an event's truth value is *true* iff the event happened at least once during the elapsed timestep, while a state-based operator's truth value is *true* iff the operator was *true* at least once during the elapsed timestep. Note that cardinality operators such as *repmin* count *all* occurrences of an event during a timestep. Once $eval(\varphi)$ has been computed, the leaves' truth values are reset for the next timestep.

2.3 Distributed Policy Decisions

As motivated in §2.1, all data usage events and data flow events must be signaled to the PDP and PIP for decision making and data flow tracking purposes. Moreover, both the PDP and the PIP maintain an internal state necessary to perform those tasks. As discussed in [13], this leads to new challenges if the data to be protected, as well as the data usage and data flow events are distributed. One naive solution to enforce global policies is to deploy one central PDP/PIP. However, such an approach is expected to be poorly performing in terms of runtime and communication overhead [13, 16].

The remaining challenge is to build an enforcement infrastructure that enforces global policies without the need for central components [14]. As such, it has been proposed to deploy PDPs and PIPs locally and consequently to keep all communication between PEP and PDP/PIP local [13]. However, consistent enforcement of global policies across all PDPs then necessitates their coordination. While naively each PDP/PIP could notify all internal state changes to all other PDPs/PIPs, we optimize our implementation according to formal results presented in [13]. In a nutshell, the paper analyzes for which policies and event traces coordination between PDPs may or may not be omitted. E.g., if $\varphi = e_1\ \underline{or}\ e_2$ with $e_1, e_2 \in \mathcal{E}$, then if e_1 happens within system A while e_2 happens within system B, then two decentrally deployed PDPs on systems A and B can both *locally* conclude $eval_A(\varphi) = eval_B(\varphi) = true$ without contacting the other PDP.

3 Architecture and Implementation

Our implementation deploys a PDP and a PIP at each site, such as a single system, or an organizational unit, cf. Fig. 3. Those components are responsible for local data flow

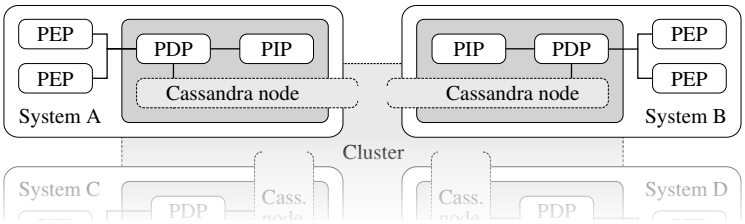

Fig. 3: High-level architecture view.

tracking and policy evaluation (§2.1–§2.2), as well as cross-system data flow tracking and policy shipment [12]. For deciding global policies, the PDPs coordinate their decisions using a distributed database (§3.2), leveraging previous results on how to efficiently enforce global data usage policies in distributed systems [13].

3.1 Distributed Policy Evaluation

Once a policy has been deployed at multiple PDPs, their decisions are expected to be consistent at all times. To explain how we achieve such consistency, we take the view of the PDP within system A, $\mathrm{PDP_A}$, enforcing policy $\mathrm{p_1}$ with trigger event $e_{\mathrm{p_1}} \in \mathcal{E}$, condition $\varphi_{\mathrm{p_1}} \in \Phi$, and action $a_{\mathrm{p_1}}$. As described in §2.2, any event signaled to $\mathrm{PDP_A}$ potentially changes the state of leaves within $\varphi_{\mathrm{p_1}}$. Since such state changes are of importance for other PDPs enforcing $\mathrm{p_1}$, $\mathrm{PDP_A}$ publishes any such state changes via the distributed database. We assume this database to be always available and strongly consistent; §3.4 explains how this is achieved in practice.

As described in §2.2, $\varphi_{\mathrm{p_1}}$ must be evaluated whenever a timestep has passed or whenever a signaled event matches $\mathrm{p_1}$'s trigger event $\mathrm{e_{p_1}}$. In any of those cases, $\mathrm{PDP_A}$ first evaluates $\varphi_{\mathrm{p_1}}$ locally. If this local evaluation yields $eval(\varphi_{\mathrm{p_1}}) = \mathit{true}$, no further coordination with other PDPs is necessary: action $\mathrm{a_{p_1}}$ will be executed. However, if $eval(\varphi_{\mathrm{p_1}}) = \mathit{false}$, then it might still be the case that $\varphi_{\mathrm{p_1}}$ is true globally, $eval^g(\varphi_{\mathrm{p_1}}) = \mathit{true}$, i.e. when considering other PDPs' state changes. Hence, $\varphi_{\mathrm{p_1}}$ is re-evaluated: For each leaf of $\varphi_{\mathrm{p_1}}$ whose local state was *false*, a lookup within the distributed database is performed. If the lookup yields *true*, implying that the operator was satisfied at some other PDP, the parent nodes are recursively re-evaluated up to the root node.[2] For example, consider condition $\varphi_{\mathrm{p_1}} = ev_1 \ \underline{and} \ \underline{isCombined}(d_1, d_2, C)$, where at system A $\mathrm{ev_1}$ is happening, while system B combines data items $\mathrm{d_1}$ and $\mathrm{d_2}$. Locally, both $\mathrm{PDP_A}$ and $\mathrm{PDP_B}$ evaluate $\varphi_{\mathrm{p_1}}$ to false, $eval_A(\varphi_{\mathrm{p_1}}) = eval_B(\varphi_{\mathrm{p_1}}) = \mathit{false}$. Subsequently, $\mathrm{PDP_A}$ looks up $\underline{isCombined}(d_1, d_2, C)$ in the distributed database, while $\mathrm{PDP_B}$ looks up whether $\mathrm{ev_1}$ happened. Hence, distributed evaluation of $\varphi_{\mathrm{p_1}}$ results in $eval^g_A(\varphi_{\mathrm{p_1}}) = eval^g_B(\varphi_{\mathrm{p_1}}) = \mathit{true}$.

It is important to note that time-based policy evaluations must consistently happen at the same time across all PDPs. Otherwise, the PDPs might come to different conclusions when evaluating the same policy. Consider once again $\varphi_{\mathrm{p_1}}$, a point in time t, a timestep

[2] In fact, for operators $\underline{isNotIn}$ and $\underline{isMaxIn}$ a lookup is performed if their local evaluation result is *true* rather than *false*. This reflects that local satisfaction of those operators never implies their global satisfaction, while their local violation always implies their global violation [13].

interval of 10 minutes, and the fact that PDP_A evaluates at times $t, t + 10, t + 20, \ldots$, while PDP_B evaluates at times $t + 5, t + 15, \ldots$. Further, assume ev_1 happens at time $t + 2$, while $\underline{isCombined}(d_1, d_2, C)$ is only true at time $t + 7$. Then, PDP_A's evaluation at time $t + 10$ yields *true*, while PDP_B's evaluation yields *false* at times $t + 5$ and $t + 15$. For this reason, our decentral PDPs always evaluate at the same time. While we are aware that such synchronization is subject to scheduling and clock synchronization issues, our experiments (cf. §4) did not reveal evaluation inconsistencies.

3.2 Using Cassandra as a Distributed Database

As indicated in Fig. 3, our infrastructure is built on top of Cassandra—a distributed database originally developed at Facebook [20] and now maintained by The Apache Foundation [21]. Cassandra's purpose is to provide a "highly available service with no single point of failure" being run "on top of [...] hundreds of nodes" [20]. As such, Cassandra has been designed to achieve high scalability, availability, and performance.

Data replication. In Cassandra, the entire set of nodes forming the distributed database is called a *cluster*. The cluster's data is organized via *keyspaces*, and each *table* is associated with exactly one keyspace. Keyspaces take a central role, since each keyspace's *replication strategy* defines among which nodes of the cluster its associated tables are replicated. Hence, data with the same replication requirements should be organized within the same keyspace. In our context, each PDP might need to enforce several policies at the same time and for each the set of remote PDPs with which coordination is required might differ. Hence, we represent each policy by exactly one keyspace. Consider policy p_1 constraining the usage of data d_1 which has representations in systems A and B. Then, in our implementation there exists keyspace k_{p_1} with replication strategy $k^{\text{rep}}_{\text{p}_1} = \{A, B\}$. Thus, if PDP_A shares a state change of φ_{p_1} within keyspace k_{p_1}, this information is replicated to exactly those PDPs for which it is of interest, i.e. PDP_B.

Data Consistency. With the CAP theorem [22] stating that consistency, availability, and partition-tolerance can not all be achieved at the same time, many eventually consistent databases have emerged. In this respect, Cassandra is flexible by allowing to trade consistency with performance. For the time being, we assume strong data consistency; §3.4 shows how this is efficiently achieved in practice. In case strong consistency is not sufficient, Cassandra provides linearizable consistency (compare-and-set transactions) on the basis of the Paxos consensus protocol [23].

As described in §4, our architecture can be flexibly deployed: While in Fig. 3 PDP, PIP, and Cassandra are local to the PEPs, it is possible to deploy those components remotely, allowing to set up a centralized infrastructure. We also assume all Cassandra nodes to know at least one seed node that is already part of the cluster; this is discussed in §3.5.

3.3 Bootstrapping and Cross-System Data Flows

Consider a set of PDPs/PIPs with their corresponding Cassandra nodes and assume that no data usage policy has yet been deployed. Then, at some point in time the first policy p_1 is deployed at PDP_A. While deploying, one or more containers are marked to

contain data d_1 whose usage is constrained by p_1. This initial classification is performed by PIP_A. Since p_1 and d_1 are only known to PDP_A, PDP_A can independently take all decisions about p_1 as described in §2.2.

Now, consider that system A shares data d_1 with system B, e.g. via network transfer. From then on, also system B might influence the evaluation of p_1. Our implementation reflects this first cross-system data transfer of d_1 by creating keyspace k_{p_1} with $k_{p_1}^{rep} = \{A, B\}$. Consequently, all data written to k_{p_1} is immediately replicated to nodes A and B. As Cassandra's database triggers are experimental, actual data flow tracking and policy transfer to system B is performed via remote procedure calls using Apache Thrift [24].

Now, system B might further share data d_1 with system C. Since keyspace k_{p_1} exists already, our implementation adapts the existing keyspace to incorporate node C, $k_{p_1}^{rep} \leftarrow k_{p_1}^{rep} \cup \{C\} = \{A, B, C\}$. Notably, the keyspace's adaption is immediately perceived by node A, such that from now on all data written to k_{p_1} will be replicated to nodes A, B and C. In order to prevent conflicts and lost updates, this adaption of a keyspace's replication strategy must be atomic; we implemented corresponding locking mechanisms on top of the keyspace being updated. For atomic acquiring of the lock, we use Cassandra's *lightweight transactions*, which provide linearizable consistency.

3.4 Cassandra Consistency

In Cassandra, each single read and write operation can be configured with a *consistency level* (CL), which defines how many nodes of the corresponding keyspace must acknowledge the operation. Among others, Cassandra provides the self-explanatory consistency levels *One*, *Two*, *Three* and *All*. While using CL=*All* guarantees strong data consistency, as assumed in this paper up to now, it comes at the cost of performance and the requirement that all of the keyspace's nodes must be always online and reachable by all other nodes. By providing consistency level *Quorum*, Cassandra allows to achieve strong consistency without such drawbacks: If CL=*Quorum*, then operations must be acknowledged by at least half of the nodes. Consequently, strong consistency can be achieved by using CL=*Quorum* for all reads and writes. Note that strong consistency can also be achieved by using CL=*All* for all writes and CL=*One* for all reads.

Whenever a consistency level different from *One* is used, reads and writes to a keyspace might fail. If CL=*All*, then it is sufficient that only one of the keyspace's nodes is not available in order to make queries to the keyspace fail. Since failing of a node or some network link is not unlikely in practice, a consistency level of *All* can be considered impractical. If CL=*Quorum*, read and write operations might fail if half of the nodes of a keyspace are not available. While such situations are not impossible, e.g. if network partitions occur, they are much more unlikely in practice. Considering the Cassandra cluster from the point of view of a single node, any query to a keyspace with CL≠*One* fails in case the considered node is offline. While configurable, by default our implementation uses CL=*Quorum* for all reads and writes.

Our implementation tackles the aforementioned problems by two means: First, it is configurable how often and in which intervals failed queries are retried. Second, if queries still fail after the predefined amount of tries, the PDP takes a fallback decision. Clearly, such a fallback decision depends on the policy being enforced, the scenario, and the attacker model. Hence, our policies can be configured accordingly.

3.5 Connecting Cassandra Nodes

When starting up, new Cassandra nodes need some way to discover the cluster they ought to participate in. Cassandra achieves this by defining a fixed set of seed nodes, through which new nodes can learn about the cluster. Since our original goal was to develop a fully decentral infrastructure, we provide solutions to the problem of integrating new nodes into an existing cluster without any well-known seed nodes. Unfortunately, Cassandra does not provide an API to explicitly command a running Cassandra node to further explore the cluster via some specific node. Having in mind that such a functionality would simplify the following solutions, we provide the following workarounds.

Recap the scenario described in §3.3, in which the very first policy p_1, protecting data d_1, is deployed at PDP_A, while PDP_B is not yet enforcing any policies. At some point in time, d_1, and subsequently policy p_1, is transferred to system B. In §3.3 we assumed system B's Cassandra node to participate in the cluster. Our solution is to not start the Cassandra node together with the PDP/PIP, but only once the first global policy ought to be enforced: Once PDP_B receives policy p_1 via remote procedure call from PDP_A, this call includes the address of system A's Cassandra node. Knowing this address, system B will start its Cassandra node, using the given address as a seed node.

Now, consider an extended scenario in which systems PDP_A and PDP_B enforce policy p_1, while PDP_C enforces policy p_2 which protects data d_2. Since the sets of systems enforcing p_1 and p_2 are disjoint, the overall cluster can be considered to be partitioned, while the single partitions are not aware of any other partitions. Once d_1 is transferred to system C, these two partitions must be merged. Since an explicit 'explore'-command as described above is missing, we solve this problem as follows: Once d_1 is transferred, we start a temporary Cassandra node which uses both A's Cassandra node as well as C's Cassandra node as seed nodes. Exploring the cluster through this temporary node, the previously autonomous parts of the cluster will get to know about each other. Once this has happened, the temporary node can be taken down again.

4 Evaluation

Since our goal was to improve over the communication and performance overhead imposed by a centralized approach, we conducted case studies to understand which approach causes which overheads in which situations. After detailing our experiment setup, we elaborate on the results obtained by enforcing ECA rules 1a, 1b and 2.

System Setup. Our virtual environment was based on VMware ESXi 5.1.0 with a host capacity of a 8x2.6GHz CPU and 128GB RAM. All machines, s0..s7, were configured with a 4x2.6GHz CPU. Further, s0 was configured with 16GB RAM, s1..s7 with 4GB RAM each. All machines run Linux Mint 17.1 64 bit, kernel 3.13.0; Cassandra was used in version 2.1.2; the infrastructure of PDP/PIP and its connection to Cassandra was written in Java 8; PEP and PDP communicated via Thrift 0.9.2. For the central system setup, s0 was hosting the central PDP/PIP instance, which was responsible for policy evaluation and data flow tracking for several PEPs being run on systems s1..s7. In this case, no Cassandra instance was run. For the decentral setup, systems s1..s7 all

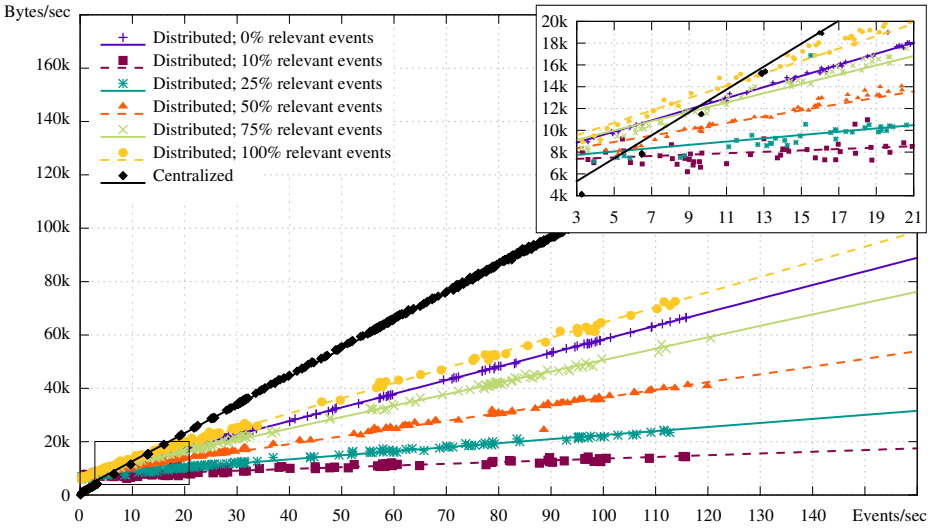

Fig. 4: Communication overhead when enforcing Policy 1a on three systems.

run exactly one instance of PEP, PDP, PIP, and Cassandra; s0 was not used. All cross-system communication was encrypted using SSL; Cassandra used CL=*Quorum*.

Parameters. We identified and experimented with the following parameters: (i) the policy being enforced, (ii) the total number of systems being usage controlled, (iii) the number of systems actually enforcing the policy, (iv) the event frequency, (v) the percentage of events relevant for data flow tracking and/or policy evaluation. Although those parameters impose a huge complexity on the performed experiments, we are confident that our results provide a good understanding of their influence on any overheads.

Experiment Execution. For each measurement we fixed all of the above parameters and randomly generated an event trace that matched the given (global) event frequency; each event was randomly assigned to one of the participating usage controlled systems. We then let the experiment run for 30 seconds, whereby the policy was evaluated upon every trigger event as well as for a timestep interval of one second. After each run, we reset the entire infrastructure. Note, that our PEPs intercepted the system events both before and after their execution, resulting in a *desired event* and an *actual event* being sent to the PDP. The data was gathered using `tcpdump` and standard datetime utilities.

We present the results that we obtained by enforcing ECA rules 1a, 1b and 2. For ECA rules 1a and 1b we fixed the total number of systems being usage controlled to three, and all of those systems where actually enforcing the policy. For ECA rule 2, a total number of seven usage controlled systems were monitored and enforcing the policy.

Communication Overhead. Fig. 4 and Fig. 5 show the global communication overhead when enforcing ECA rules 1a and 1b, respectively. We experimented with the event frequency and the percentage of events relevant for data flow tracking and policy evaluation. Trends are visualized using linear regression.

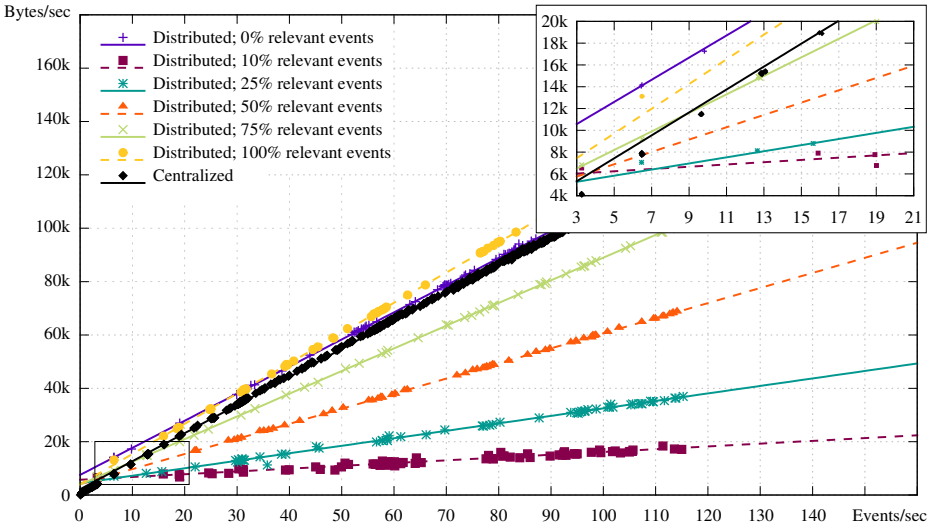

Fig. 5: Communication overhead when enforcing Policy 1b on three systems.

The results produced by the central system setup (Fig. 4 and Fig. 5, ◆) where of little surprise: For each event being observed by a PEP, around 1070 Bytes were exchanged between the PEP and the PDP. The percentage of relevant events did not have any influence on the communication overhead. This is of no surprise when recapping that the PEP is stateless and that *every* event must be signaled to the PDP.

Running our decentralized infrastructure, our first observation is that Cassandra causes some base 'noise' of around 1050 Bytes/sec/node—independent of any operations being performed. This implies that the centralized approach will inexorably perform better in case of very low event frequencies as can be seen in Fig. 4 and Fig. 5. However, depending on the event frequency as well as the percentage of relevant events, our decentralized approach is capable of outperforming the centralized approach.

While in general event traces with a low percentage of relevant events perform particularly well (Fig. 4 and Fig. 5, ■ (10% relevant events), ✳ (25%)), we also observe some remarkable exceptions. First of all, aforementioned traces perform good for two reasons: (i) policies can in many cases be conclusively evaluated locally, avoiding costly lookups within the distributed database; (ii) a low percentage of relevant events implies a small amount of state changes that must be notified to other PDPs and thus written to the database. Secondly, traces with 0% of relevant events perform badly (+), since our infrastructure must perform database lookups for each event and timestep. Thirdly, traces with a high percentage of relevant events also perform badly (× (75%), ● (100%)). While in the latter case the PDPs can almost always decide locally, a high amount of state changes must be notified to other PDPs. Thus, the lion's share of the communication overhead is due to state changes being written to the database.

As depicted in Fig. 4 and Fig. 5, ECA rule 1a can be evaluated more efficiently than ECA rule 1b. The main reason is that evaluation of operator *before* in ECA rule 1a necessitates at most one database lookup per PDP per timestep, while in the worst

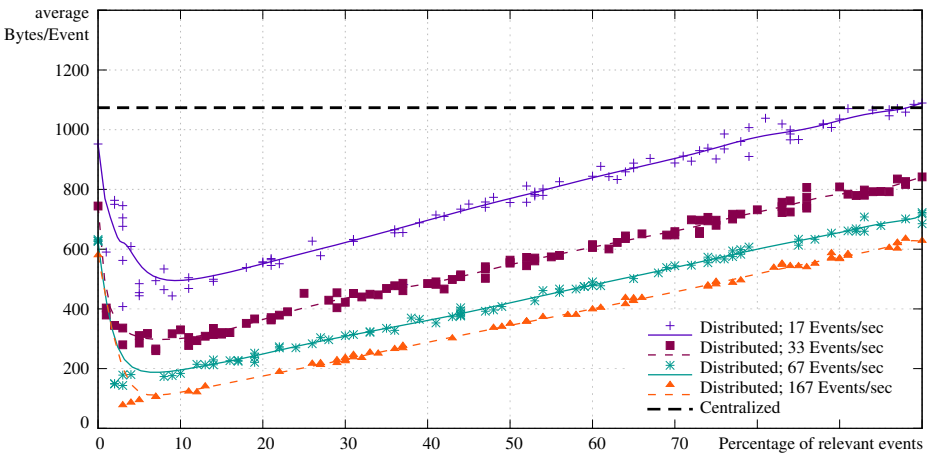

Fig. 6: Communication overhead when enforcing Policy 1a on three systems.

case each *repmin* operator, which occurs twice in ECA rule 1b, necessitates one lookup upon every event.

Fixing several event frequencies, Fig. 6 and Fig. 7 show how the percentage of relevant events influences the total amount of Bytes being exchanged between all involved systems. To compare those numbers, we normalize the measurements by dividing the total amount of Bytes by the number of observed events. Again, for the centralized approach (- - -) the communication overhead is constant (1070 Bytes per event) and the percentage of relevant events does not influence the amount of Bytes being exchanged.

We observe that the decentralized approach performs best for high event frequencies (Fig. 6 and Fig. 7, ✳ (67 Events/sec), ▲ (167 Events/sec)) and if the percentage of relevant events is around 3% to 60%. Firstly, this is because higher event frequencies exploit better Cassandra's base noise, which keeps the database consistent. Secondly, a low percentage of relevant events results in many situations in which the local PDPs can decide conclusively, while a low amount of state changes must be signaled to other PDPs. However, if the amount of relevant events is too low, then many lookups within the database are required, while the presence of many relevant events results in many writes to the database. Hence, the centralized approach outperforms the decentralized approach if the percentage of relevant events is very low or very high ($\lesssim$2%, $\gtrsim$85%; concrete values depend on the policy and the event frequency, cf. Fig. 6 and Fig. 7).

Regarding the enforcement of ECA rule 2 within a total of seven usage controlled systems, the most important difference to ECA rules 1a and 1b is the condition of ECA rule 2. This condition is satisfied if event *Web.decline*(*d*) happened at least once in the past. Once this event is observed for the first time and notified to all other PDPs, no further coordination is ever needed. This is also reflected in our evaluation results. First of all, we again observe a worst case scenario if no events relevant for policy evaluation are happening (§A, Fig. 9, +). In this case each PDP must query the database upon each evaluation in order to learn whether the event in question has happened at some remote point. Since this event never happens, communication overhead is immense. However,

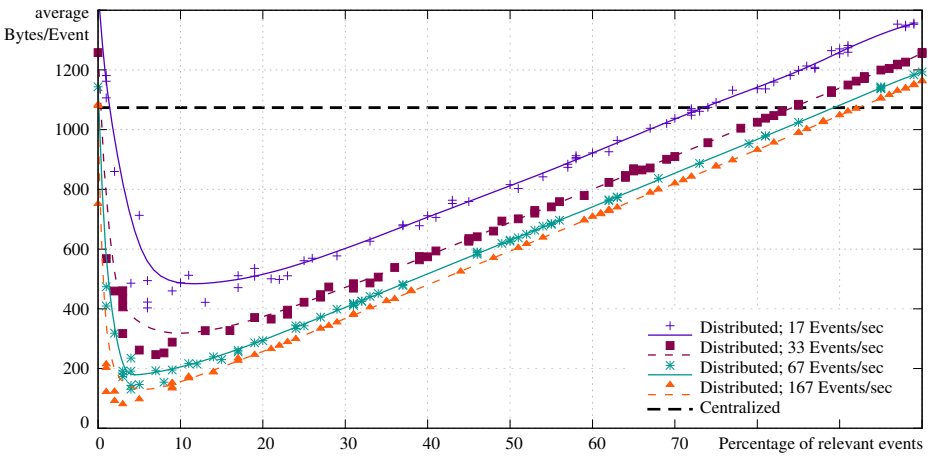

Fig. 7: Communication overhead when enforcing Policy 1b on three systems.

once event *Web.decline*(*d*) has happened, then no further communication is required, and we only observe Cassandra's base noise (§A, Fig. 9, (10%), (25%), (50%), (75%), (100%)). As for the other scenarios, the communication overhead caused by the centralized infrastructure is linear in the number of events. Again, §A, Fig. 10 shows that a very low percentage of relevant events (i.e. $\lesssim$2%) causes very high communication overheads. However, different to the previous policies, §A, Fig. 10 reveals that for ECA rule 2 the communication overhead for higher percentages of relevant events is constant; the decentralized approach outperforms the centralized approach if the global event frequency is higher than approximately 20 Events/sec.

In addition, we enforced ECA rule 1a within a total of seven usage controlled systems, only three of them being 'aware' of the protected data and thus enforcing the policy. While the communication overhead in the centralized approach was the same as in the scenarios described above, in the decentral approach it dropped to approximately 60% of the above values for ECA rule 1a: While in the central approach still *every* event must be signaled to the central PDP, in our decentralized approach four out of three PDPs are not aware of any copy of the protected data and thus they neither enforce the policy nor participate in the corresponding Cassandra keyspace.

PDP evaluation times. Fig. 8 shows how many milliseconds it takes for an event to be decided upon by the PDP for different event frequencies and percentages of relevant events. For each event, this includes (i) the time to send the event from the PEP to the PDP, (ii) the PDP's evaluation process, and (iii) the time to send the decision to the PEP.

For the *centralized infrastructure*, we observe that the evaluation times increase as the event frequency increases. Clearly, higher event frequencies push the central PDP towards its limits, since more events must be processed by the single component. Also, more events cause more load on the network an thus slightly higher network latency. For the same reasons as discussed above, the percentage of relevant events is irrelevant. Overall, the PEP usually gets responses from the PDP after 3 to 10ms.

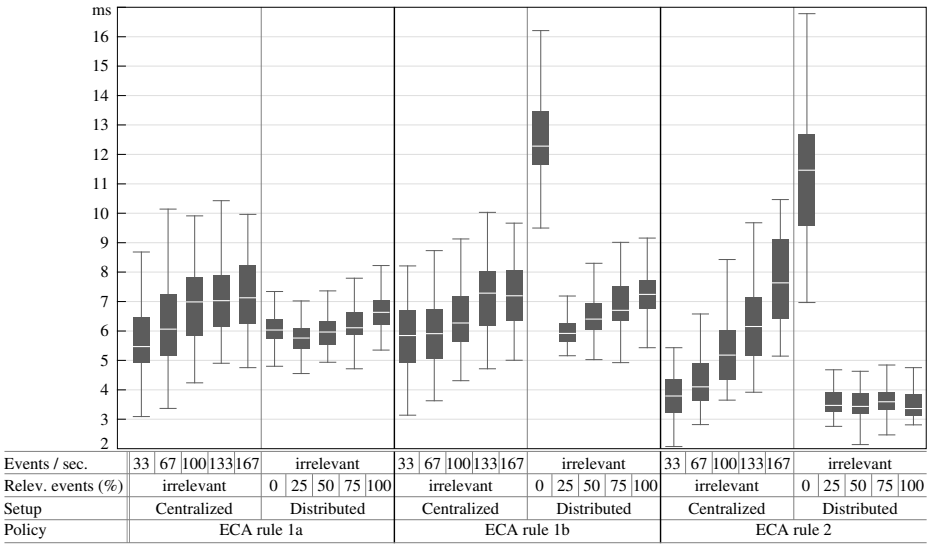

Fig. 8: PDP evaluation times when enforcing ECA rules 1a, 1b, and 2.

For the *distributed infrastructure*, we observed that the event frequency does *not* influence the PDP's evaluation times. However, we observe an anomaly when enforcing traces with 0% relevant events. This is in correspondence with the communication overhead and can be explained by the fact that in this case the PDPs can never conclusively evaluate locally. Hence, for each event lookups within the database are required. By using the *Quorum* consistency level, this results in remote requests to other nodes of the cluster, decreasing performance of the evaluation process. In these cases, the PEP may need to wait up to 16ms for the PDP's response. In contrast, if an event trace contains at least some relevant events, then the distributed decision process is capable of outperforming the centralized approach by providing responses between 2 to 9ms.

Wrap-Up. Considering the bare numbers, we realize that a fully decentralized enforcement infrastructure is not unconditionally superior to a centralized one. According to our case studies, the adoption of a decentralized approach is particularly beneficial if (i) event frequencies are high, (ii) the percentage of events relevant for policy evaluation and/or data flow tracking is within a range of approximately 3% to 60%, and/or if (iii) the policy being enforced allows for many locally conclusive evaluations.

Instead of blindly deploying either of those infrastructures, experiments as the ones above should be performed, considering the concrete parameters, i.e. the policies, the amount of systems, and the expected event traces, of a given application scenario. We also envision that such experiments can be performed at runtime, and that the technology in use (i.e. central or decentral) may be switched dynamically in correspondence with those live observations. While Cassandra simplified the implementation of our infrastructure, it comes at the cost of performance and communication overhead. It stands to reason that a tailored solution would improve upon those overheads.

5 Related Work

Service Automata [17] address the problem of enforcing policies that cannot be decided locally. For this, local "service automatons", roughly equivalent to PEP, PDP and PIP, monitor the execution of programs within a distributed system. If an automaton's knowledge is insufficient to take a policy decision, it delegates the decision to some other automaton. For this, each security-relevant event is statically mapped to one single responsible automaton; possibly conflicting events must be mapped to the same automaton. In contrast, our approach does not rely on such a static mapping, but allows each PDP to take the corresponding decisions. Further, Service Automata do not cater to the fact that the data to be protected might be copied both within and across systems.

Lazouski et al. [8] provide a framework that enforces usage control policies if data copies are distributed. Besides access and usage control rules, so-called PDP/PIP allocation policies are embedded into the protected data, specifying which PDPs and PIPs are involved in the decision process. Subject and object attributes required by the PDP are stored at different PIP locations. Different to our approach, the proposed allocation policies effectively introduce central components: for each protected data, the responsible PDP is fixed. Also, each attribute is under the responsibility of one single PIP. Failure of any of those components will break policy enforcement.

Bauer et al. [18] monitor LTL formulas in distributed systems using rewriting techniques. Whenever a local monitor observes an event that influences policy evaluation, the policy is rewritten according to predefined rules and then exchanged with the other local monitors. Hence, the local monitors are capable of detecting violation or satisfaction of the formula. The approach is different from ours in that it requires a synchronous system bus. Further, our approach is more expressive by also considering state-based usage control policies and by integrating data flow tracking.

Basin et al. [6, 25] are capable of detectively enforcing data usage policies in distributed systems. For this, log files are decentrally collected and a-posteriori (i.e. offline) merged and evaluated against data usage policies. In contrast, our approach also allows for the preventive enforcement of data usage policies.

6 Conclusions

We presented the first fully decentralized infrastructure for the preventive enforcement of global data usage policies if the data to be protected, as well as the corresponding data usage events, happen within multiple distributed systems.

We based the implementation of our infrastructure onto the distributed database Cassandra. Local monitors, PEPs, observe data usage events within the distributed system, and signal those events to local decision points, PDPs, which decide whether the event complies with the data usage policies. Since remote PDPs might also observe events that influence the local PDP's decision, the PDPs exchange relevant information via Cassandra. This way, we achieve consistent policy enforcement across multiple PDPs without any central components. To minimize the amount of database queries, we optimized our implementation using formal results from the literature.

We evaluated our infrastructure by comparing it with a centralized approach, in which one single PDP is responsible for taking all policy decisions for all events being

observed by all distributed PEPs. Our case studies revealed that the adoption of a decentralized infrastructure is particularly beneficial in case the frequency of the observed system events is high and if approximately 3% to 60% of all events are of relevance for policy evaluation and/or data flow tracking. In terms of PDP evaluation times, our results revealed that the centralized and the decentralized approach perform similarly. For our decentralized infrastructure, the PEP usually gets policy evaluation results from the PDP within 2 to 9ms. While performing our experiments, we also realized that all of the above evaluation results highly dependent on the policy being enforced. Notably, there also exist policies (cf. ECA rule 2) for which the decentralized approach performs tremendously better than the centralized one for most situations.

In any case, a decentralized infrastructure overcomes many problems omnipresent in a centralized approach. By deploying all components locally and by replicating data to different locations, there is no single point of failure and no need for a central component to be always available for all clients.

In terms of future work, we plan to experiment with the different consistency levels provided by Cassandra, which allow to trade consistency with performance. While we will likely be able to improve performance and communication overhead, it would be interesting to understand to which extent a non-strongly consistent database influences the consistency of the distributed policy evaluations. Clearly, it would depend on the considered scenario whether any such inconsistencies are acceptable. Depending on those results, a further option is to abandon off-the-shelf databases and to implement mechanisms specifically tailored to usage control requirements.

Acknowledgements. This work was supported by the DFG Priority Programme 1496 “Reliably Secure Software Systems - RS3”, grant PR-1266/3.

References

[1] A. Pretschner, M. Hilty, and D. Basin. “Distributed Usage Control”. In: *Communications of the ACM* 49.9 (Sept. 2006), pp. 39–44.

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

[3] J. Park and R. Sandhu. “The UCONABC Usage Control Model”. In: *ACM Transactions on Information and System Security* 7.1 (Feb. 2004), pp. 128–174.

[4] X. Zhang, F. Parisi-Presicce, R. Sandhu, and J. Park. “Formal Model and Policy Specification of Usage Control”. In: *ACM Transactions on Information and System Security* 8.4 (Nov. 2005), pp. 351–387.

[5] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter. “A Policy Language for Distributed Usage Control”. In: *Computer Security – ESORICS 2007*. Vol. 4734. LNCS. Springer Berlin Heidelberg, 2007, pp. 531–546.

[6] D. Basin, M. Harvan, F. Klaedtke, and E. Zălinescu. “Monitoring Data Usage in Distributed Systems”. In: *IEEE Transactions on Software Engineering* 39.10 (Oct. 2013), pp. 1403–1426.

[7] A. Pretschner, E. Lovat, and M. Büchler. "Representation-Independent Data Usage Control". In: *Data Privacy Management and Autonomous Spontaneus Security*. Vol. 7122. LNCS. Springer Berlin Heidelberg, 2012, pp. 122–140.

[8] A. Lazouski, G. Mancini, F. Martinelli, and P. Mori. "Architecture, Workflows, and Prototype for Stateful Data Usage Control in Cloud". In: *IEEE Security and Privacy Workshops*. May 2014, pp. 23–30.

[9] A. Fromm, F. Kelbert, and A. Pretschner. "Data Protection in a Cloud-Enabled Smart Grid". In: *Smart Grid Security*. Ed. by J. Cuellar. Vol. 7823. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 96–107.

[10] M. Harvan and A. Pretschner. "State-Based Usage Control Enforcement with Data Flow Tracking using System Call Interposition". In: *3rd International Conference on Network and System Security*. Oct. 2009, pp. 373–380.

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

[12] F. Kelbert and A. Pretschner. "Data Usage Control Enforcement in Distributed Systems". In: *Proc. 3rd ACM Conference on Data and Application Security and Privacy*. 2013, pp. 71–82.

[13] F. Kelbert and A. Pretschner. "Decentralized Distributed Data Usage Control". In: *Cryptology and Network Security*. Vol. 8813. LNCS. Springer International Publishing, 2014, pp. 353–369.

[14] D. Basin, F. Klaedtke, S. Müller, and E. Zălinescu. "Monitoring Metric First-order Temporal Properties". In: *Journal of the ACM* (2015). To appear.

[15] Adobe Systems Incorporated. *Adobe Content Server*. 2015. URL: `http://www.adobe.com/solutions/ebook/content-server.html` (visited on 04/02/2015).

[16] H. Janicke, A. Cau, F. Siewe, and H. Zedan. "Concurrent Enforcement of Usage Control Policies". In: *IEEE Workshop on Policies for Distributed Systems and Networks*. June 2008, pp. 111–118.

[17] R. Gay, H. Mantel, and B. Sprick. "Service Automata". In: *Formal Aspects of Security and Trust*. Vol. 7140. LNCS. Springer Berlin Heidelberg, 2012.

[18] A. Bauer and Y. Falcone. "Decentralised LTL Monitoring". In: *FM 2012: Formal Methods*. Vol. 7436. LNCS. Springer Berlin Heidelberg, 2012, pp. 85–100.

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

[20] A. Lakshman and P. Malik. "Cassandra: A Decentralized Structured Storage System". In: *ACM SIGOPS Operating Systems Review* 44.2 (Apr. 2010), pp. 35–40.

[21] The Apache Software Foundation. *The Apache Cassandra Project*. 2014. URL: `http://cassandra.apache.org/` (visited on 04/02/2015).

[22] E. A. Brewer. "Towards Robust Distributed Systems". In: *Proc. of the 19th Annual ACM Symposium on Principles of Distributed Computing*. Keynote. 2000.

[23] L. Lamport. "The Part-time Parliament". In: *ACM Transactions on Computer Systems* 16.2 (May 1998), pp. 133–169.

[24] The Apache Software Foundation. *Apache Thrift*. 2014. URL: `https://thrift.apache.org/` (visited on 04/02/2015).

[25] D. Basin, G. Caronni, S. Ereth, M. Harvan, F. Klaedtke, and H. Mantel. "Scalable Offline Monitoring". In: *Runtime Verification*. Vol. 8734. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 31–47.

A Further Evaluation Results

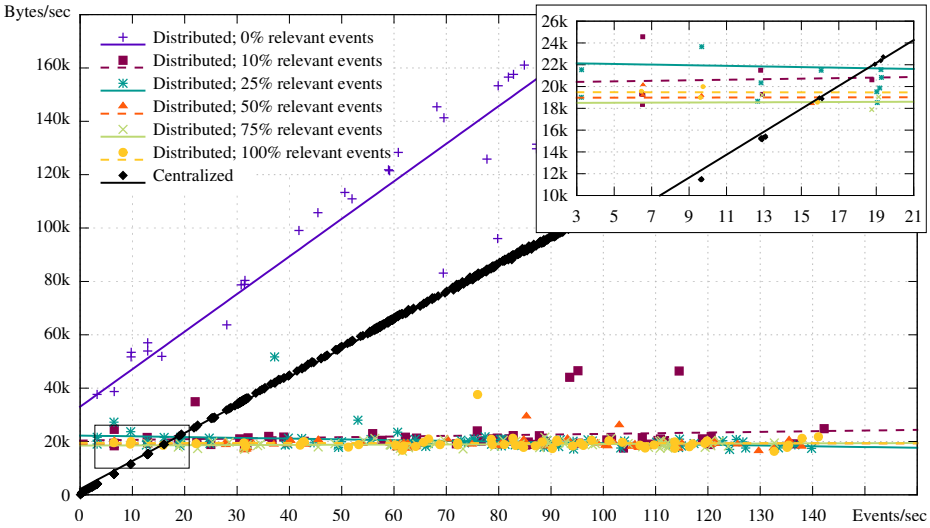

Fig. 9: Communication overhead when enforcing Policy 2 on seven systems.

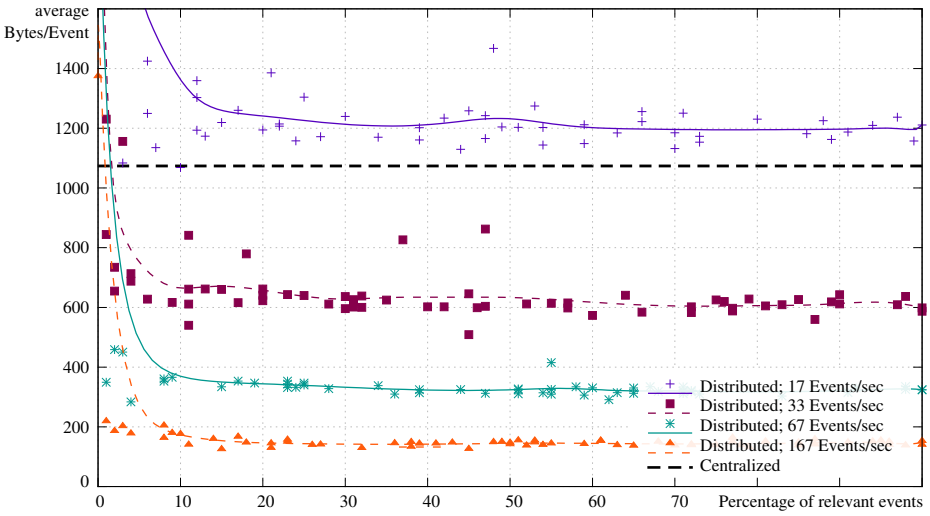

Fig. 10: Communication overhead when enforcing Policy 2 on seven systems.