



Instrumenting Runtime Enforcement

François Hublet¹(✉), David Basin¹, Linda Hu¹, Srđan Krstić¹,
and Lennard Reese²

¹ ETH Zürich, Zurich, Switzerland

{francois.hublet,basin,srdan.krstic}@inf.ethz.ch, lindhu@student.ethz.ch

² University of Copenhagen, Copenhagen, Denmark
lere@di.ku.dk

Abstract. Runtime enforcement ensures that a running system complies with a property by observing and modifying the system’s actions. In practice, the property is often defined in terms of high-level, abstract events, while the system’s behavior consists of low-level, concrete actions. The relationship between actions and events is established in the *instrumentation* process, where developers must ensure that (i) system actions report the right events, and (ii) the necessary modifications to the system’s behavior are correctly enforced. However, the abstraction gap between a high-level property and low-level actions makes this process error-prone.

In this paper, we refine an existing formal model of runtime enforcement, which leaves instrumentation implicit, into a more precise model that explicitly accounts for instrumentation. We propose a correctness criterion for instrumentation and present a novel library, called INSTR-LIB, that instruments Python applications for runtime enforcement.

Keywords: Runtime Enforcement · Instrumentation · Edit Automata

1 Introduction

In 2022, personal data of roughly one third of Australia’s population was stolen from the telecommunication provider Optus [26]. The attack was unsophisticated, involving the attacker exploiting a coding error in the instrumentation of an internet-facing, legacy API. Due to this error, the access control (AC) mechanism, while in place, was not invoked to protect the legacy API, allowing for the easy retrieval of millions of user records.

This data breach highlights a recurring theme: even when appropriate security mechanisms are in place, incorrect instrumentation can allow attackers to bypass the mechanisms entirely. A rigorous approach to instrumentation is especially crucial in applications where the property is a set of desired sequences of abstract events, with each event reflecting many possible concrete system actions. The prominent example of this is when enforcing requirements derived from privacy law [14, 16]: if a regulation requires that “no user data is used without prior consent,” then being able to ensure that “data usage” is blocked whenever

“consent” has not been previously registered is insufficient to certify that an application complies with the law. The developers must also ensure that “data usage” is correctly identified by existing system instrumentation whenever low-level actions such as database reads and writes occur; furthermore, they must check that “consent” is only registered when users actually give consent in the UI.

Runtime enforcement generalizes AC by using execution monitors that observe the actions of a running system and modify these actions to ensure that only compliant behaviors are allowed. While, in recent years, increasingly powerful enforcement approaches and tools have been developed, much less attention has been devoted to the questions of how to properly instrument systems or how to audit existing instrumented systems to ensure correct enforcement.

Figure 1 (top) shows the idealized system model used in most previous work, which we call the *classical* model of runtime enforcement. In this model, the System under Enforcement (SuE) is a labeled transition system (LTS) with each transition labeled by some *event* e . When attempting to take a transition labeled by e (Step ①), the SuE sends the event e to a policy decision point (PDP) in charge of ensuring the SuE’s compliance with a property P (Step ②). The PDP edits [21] the event e to a sequence of events e' compliant with P (Step ③). The modification can involve removing, replacing, or inserting events. The events e' are then returned as a command to the SuE (Step ④), which takes the appropriate transitions (Step ⑤). As it assumes that all events are correctly sent to the PDP and that the SuE always follows the PDP’s commands, the classical model cannot capture non-compliance due to incorrect instrumentation.

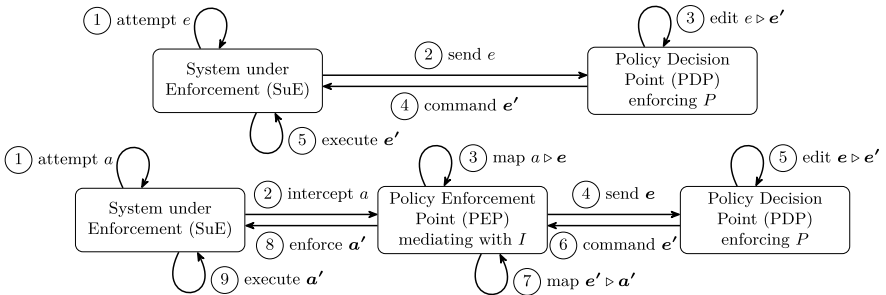


Fig. 1. The classical (top) and extended (bottom) enforcement models

We propose an *extended* model, shown in Fig. 1 (bottom), where the SuE is modeled as an LTS with transitions labeled by *actions* distinct from the events sent to the PDP. In addition to the SuE and PDP, our model introduces an explicit policy enforcement point (PEP) [9] that instruments the actions performed by the SuE, producing events processed by the PDP. Concretely, the PEP intercepts every action a attempted by the system (Step ②) and maps it to a sequence of events e (Step ③). This sequence is sent to the PDP (Step ④), which edits e to some e' (Step ⑤) and returns e' (Step ⑥). The PEP maps

e' back to a sequence of actions a' (Step (7)) that the PEP enforces in the SuE (Steps (8) and (9)).

In our extended model, the SuE's specification consists of two elements: the property P and a *mediator* I providing the desired interpretation of system actions in terms of PDP events. In the Optus data breach, the property P may have been “whenever a user u performs an API access ($\text{APIAccess}(u)$ event), then u is authenticated,” while the mediator I mapped both system actions $\text{legacyAPIAccess}(u)$ and $\text{modernAPIAccess}(u)$ to the event $\text{APIAccess}(u)$. However, the PEP failed to map $\text{legacyAPIAccess}(u)$ to $\text{APIAccess}(u)$. As a result, although the PDP correctly enforced P , the composition of the SuE, PEP, and PDP did not provide the desired security guarantees.

Contributions. After reviewing the classical enforcement model (Sect. 2), we make the following contributions:

- We formally introduce our extended enforcement model and define the notions of PEP correctness that provide necessary and sufficient conditions for correct instrumentation, independent of any specific PDP (Sect. 3).
- We show how these conditions can be relaxed for properties where the occurrence of certain events can be soundly overapproximated (Sect. 4).
- We further specialize our theory to support state-of-the-art PDPs that process events in finite sets (‘batches’). We provide a correct PEP algorithm for this setup and give auditing check-lists for ensuring compliance (Sect. 5).
- We implement our PEP algorithm in INSTRLIB, an open-source library for enforcing properties of Python applications using the state-of-the-art ENFGUARD [17] tool as the PDP. We illustrate our framework in a case study, enforcing privacy requirements in a micro-blogging application using INSTRLIB and then auditing our instrumentation's correctness (Sect. 6).

Related Work. Models of runtime enforcement mechanisms include security automata [11, 27], edit automata [21], mandatory results automata (MRA) [22], and timed automata [3, 24]. More recently, several frameworks for enforcing expressive first-order properties at runtime have also been developed [13, 17, 18]. Runtime enforcement can be performed both on high-level events and low-level system actions, e.g., through inlining [10]. Most models only consider the PDP's logic, without any guarantees of correct instrumentation. MRAs distinguish between system *actions* and enforced *results* and may fulfill the role of both a PDP and PEP, but do not provide for a clear separation between instrumentation and property enforcement. In contrast, several works from the security community discuss the composition of a PDP and PEP in the context of runtime enforcement without formally or precisely describing this composition [4, 14, 15, 25]. Our account of the ‘classical model’ builds on work by Aceto et al. [1] and Falcone et al. [7, 12], where an SuE modeled as an LTS and the PDP run in lockstep.

2 Preliminaries

After introducing notation, we review labeled transition systems and edit automata (Sect. 2.1). We then introduce the ‘classical’ enforcement model and its associated notions of PDP soundness and transparency (Sect. 2.2).

Notation. Given a set A , the set of all finite sequences of elements of A is denoted by A^* . We use Greek letters (e.g., $\alpha, \sigma, \rho, \dots$) or bold Latin letters (e.g., $\mathbf{a}, \mathbf{e}, \mathbf{\ell}, \dots$) to denote sequences; bold Greek letters denote sequences of sequences. We denote the empty sequence as ε and finite sequences as $\langle a_1, a_2, \dots, a_n \rangle$. The sequence $\sigma \setminus a$ stands for σ with all occurrences of a removed, $\sigma_{..i}$ for the prefix of σ of length i , and $\text{pre}(\sigma)$ for the set of all prefixes of σ . Given $\sigma, \sigma' \in A^*$, the sequence $\sigma \cdot \sigma'$ is the concatenation of σ and σ' . Given a sequence of sequences $\boldsymbol{\sigma} = \langle \sigma_1, \dots, \sigma_n \rangle \in (A^*)^*$, we denote by $\circ\boldsymbol{\sigma} \triangleq \sigma_1 \cdot \sigma_2 \cdot \dots \cdot \sigma_n$ the concatenation of the σ_i .

For any set of labels \mathcal{L} , the set of *traces* over \mathcal{L} is $\mathbb{T}_{\mathcal{L}} \triangleq \mathcal{L}^*$. A *property* $P_{\mathcal{L}}$ over \mathcal{L} is a set of traces, i.e., a subset $P_{\mathcal{L}} \subseteq \mathbb{T}_{\mathcal{L}}$. To support *internal* (or ‘silent’) system actions, we use a distinguished label $\tau \notin \mathcal{L}$ and denote $\mathcal{L}_{\tau} \triangleq \mathcal{L} \cup \{\tau\}$.

2.1 Labeled Transition Systems and Edit Automata

As in previous work [1, 2, 6], we model systems as labeled transition systems:

Definition 1 (LTS). A labeled transition system (LTS) over \mathcal{L} is a quadruple $\mathcal{S} = (\mathbb{S}, \mathcal{L}, s^0, \dot{\rightarrow})$ such that \mathbb{S} is a set of states, \mathcal{L} is a set of labels, $s^0 \in \mathbb{S}$ is an initial state, and $\dot{\rightarrow} \subseteq \mathbb{S} \times \mathcal{L}_{\tau} \times \mathbb{S}$ is a transition relation labeled by \mathcal{L}_{τ} .

We write $s \xrightarrow{\ell} s'$ for $(s, \ell, s') \in \dot{\rightarrow}$. An LTS execution is of the form $s^0 \xrightarrow{\ell_1} s^1 \xrightarrow{\ell_2} \dots \xrightarrow{\ell_n} s^n$ and the trace of such an execution is $\sigma = \langle \ell_1, \ell_2, \dots \rangle \setminus \tau$, removing all internal τ actions. In this case, we also write $s^0 \xrightarrow{\sigma} s^n$.

Example 1. Figure 2 represents a social network application as an LTS over two different sets of labels, \mathcal{E} (‘events’, left) and \mathcal{A} (‘actions’, right) capturing the system’s behavior at two levels of abstraction. For simplicity, we model the system for a single user. At a high level (events), the system can be described in terms of user interaction (give Consent for data usage, Revoke consent, Request data deletion), backend operations (Use/Delete user data), and clock ticks (Tick). At a lower level (actions), one can observe UI interactions (click_yes, click_no in a consent banner, clicks on a request button), database (read, write, delete) or authentication (Login, Logout) operations, and clock ticks (tick).

Edit automata (EA) [21] are a general model for PDPs, providing an abstract model for a large class of practical enforcement mechanisms. Edit automata are a special kind of LTS which, in each step, read a label ℓ^1 from some set of labels \mathcal{L}^1 and edit it deterministically into a possibly empty sequence of labels ℓ^2 from another set \mathcal{L}^2 . If no ℓ^2 exists for a given ℓ^1 , the execution of the LTS is terminated, similar to execution cutting in security automata [27].

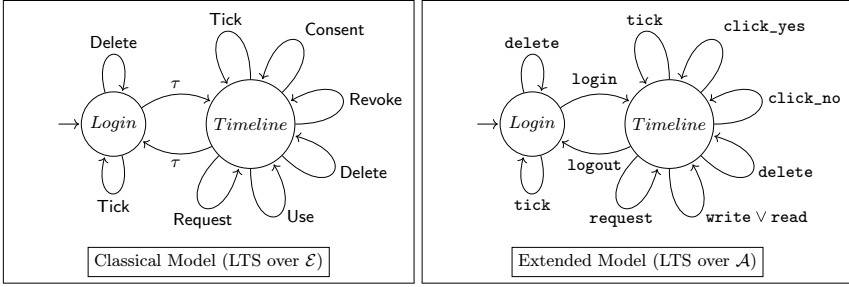


Fig. 2. Systems under Enforcement (SuE) in the Classical and Extended Model

Definition 2 (Edit Automaton). An edit automaton (EA) over $(\mathcal{L}^1, \mathcal{L}^2)$ is an LTS $\delta = (\mathbb{S}_\delta, \mathcal{L}^1 \times \mathcal{L}_\tau^{2*}, s_\delta^0, \xrightarrow{\cdot \triangleright} \delta)$ with a transition relation labeled with pairs of labels $(\ell_1, \ell_2) \in \mathcal{L}^1 \times \mathcal{L}_\tau^{2*}$, also denoted as $\ell_1 \triangleright \ell_2$, such that, for any $s_\delta \in \mathbb{S}_\delta$ and $\ell_1 \in \mathcal{L}^1$, there exists at most one pair $(s'_\delta, \ell_2) \in \mathbb{S}_\delta \times \mathcal{L}_\tau^{2*}$ such that $s_\delta \xrightarrow{\ell_1 \triangleright \ell_2} s'_\delta$.

An edit automaton is input-enabled [23] iff for any ℓ^1 , the automaton can always edit ℓ^1 to some ℓ_2 . Written more formally:

Definition 3. An edit automaton δ over $(\mathcal{L}^1, \mathcal{L}^2)$ is input-enabled iff for any $s_\delta \in \mathbb{S}_\delta$ and $\ell_1 \in \mathcal{L}^1$, there exists $s'_\delta \in \mathbb{S}_\delta$ and $\ell_2 \in \mathcal{L}_\tau^{2*}$ such that $s_\delta \xrightarrow{\ell_1 \triangleright \ell_2} s'_\delta$.

Note that there are two interpretations of an EA: the EA accepts a language of traces over pairs containing a label (from \mathcal{L}^1) and a sequence of labels (from \mathcal{L}_τ^{2*}). Alternatively, it is a transducer, translating a sequence of labels from \mathcal{L}^1 into a sequence of labels from \mathcal{L}^2 by concatenating the ℓ_2 . For $n \in \mathbb{N}$, $\xi = (x_i)_{1 \leq i < n}$, and $v = (y_i)_{1 \leq i < n}$, we denote by $\xi \triangleright v$ the zipped sequence $(x_i \triangleright y_i)_{1 \leq i < n}$. For any EA δ , we abuse notation and write δ as a partial function such that $\delta(\sigma) = \circ\sigma' \iff \exists s'_\delta. s_\delta^0 \xrightarrow{\sigma \triangleright \sigma'} s'_\delta$, reflecting the view of δ as a trace transducer.

2.2 The Classical Model of Runtime Enforcement

In the classical enforcement model [1, 6, 21], an SuE (LTS) and a PDP (edit automaton) over the same set of labels are composed, progressing in lock-step. This can be formalized as follows in terms of LTS composition:

Definition 4. An LTS \mathcal{S} over \mathcal{L} and an edit automaton δ over $(\mathcal{L}, \mathcal{L})$ serving as a PDP can be composed into an LTS $\langle \mathcal{S} | \delta \rangle = (\mathbb{S}_{\langle \mathcal{S} | \delta \rangle}, \mathcal{L}, s_{\langle \mathcal{S} | \delta \rangle}^0, \xrightarrow{\cdot} \langle \mathcal{S} | \delta \rangle)$ by

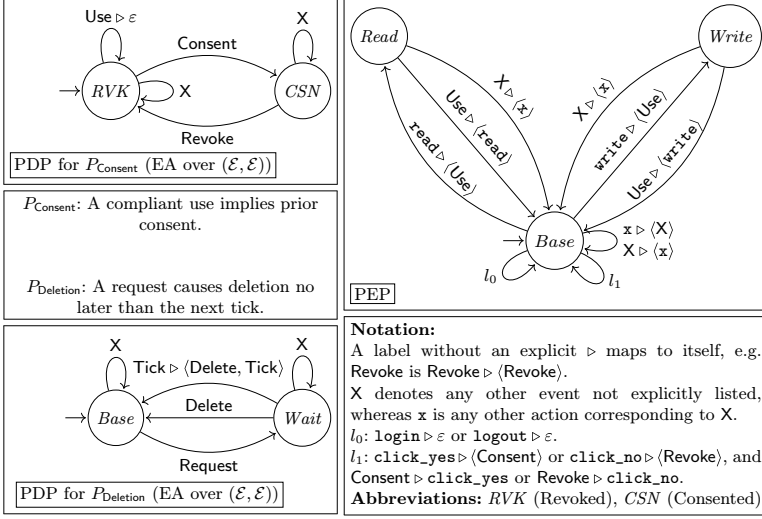


Fig. 3. Policy Decision Points (PDPs) and Policy Enforcement Point (PEP)

$$\begin{array}{c}
 \mathbb{S}_{\langle \mathcal{S} | \delta \rangle} \triangleq \mathbb{S}_{\mathcal{S}} \times \mathbb{S}_{\delta} \times \mathcal{L}^* \quad s_{\langle \mathcal{S} | \delta \rangle}^0 \triangleq (s_{\mathcal{S}}^0, s_{\delta}^0, \varepsilon) \\
 \frac{s_{\mathcal{S}} \xrightarrow{\tau} s'_{\mathcal{S}}}{(s_{\mathcal{S}}, s_{\delta}, b) \xrightarrow{\tau} \langle \mathcal{S} | \delta \rangle (s'_{\mathcal{S}}, s_{\delta}, b)} \text{step}_{\tau} \quad \frac{s_{\mathcal{S}} \xrightarrow{\ell' \neq \tau} s''_{\mathcal{S}} \quad s_{\delta} \xrightarrow{\ell' \triangleright \varepsilon} s'_{\delta}}{(s_{\mathcal{S}}, s_{\delta}, \varepsilon) \xrightarrow{\tau} \langle \mathcal{S} | \delta \rangle (s_{\mathcal{S}}, s'_{\delta}, \varepsilon)} \text{step}_0 \\
 \frac{s_{\mathcal{S}} \xrightarrow{\ell' \neq \tau} s''_{\mathcal{S}} \quad s_{\delta} \xrightarrow{\ell' \triangleright \langle \ell \rangle \cdot \ell} s'_{\delta} \quad s_{\mathcal{S}} \xrightarrow{\ell} s'_{\mathcal{S}}}{(s_{\mathcal{S}}, s_{\delta}, \varepsilon) \xrightarrow{\ell} \langle \mathcal{S} | \delta \rangle (s'_{\mathcal{S}}, s'_{\delta}, \ell)} \text{step}_+ \quad \frac{s_{\mathcal{S}} \xrightarrow{\ell} s'_{\mathcal{S}}}{(s_{\mathcal{S}}, s_{\delta}, \langle \ell \rangle \cdot \ell) \xrightarrow{\ell} \langle \mathcal{S} | \delta \rangle (s'_{\mathcal{S}}, s_{\delta}, \ell)} \text{buf}
 \end{array}$$

The states of the *enforced LTS* $\langle \mathcal{S} | \delta \rangle$ have three components: the state of \mathcal{S} , the state of δ , and a buffer in \mathcal{L}^* used to temporarily store the sequences of labels returned by the PDP δ until corresponding transitions are taken by the system. The enforced LTS has four kinds of transitions:

- step_{τ} transitions mirror internal transitions in $\langle \mathcal{S} | \delta \rangle$ without δ 's intervention;
- step_0 transitions are triggered by the EA erasing the original label ℓ' of a transition of \mathcal{S} , i.e., editing it to ε . In this case, the substate of $\langle \mathcal{S} | \delta \rangle$ corresponding to the state of \mathcal{S} does not change;
- step_+ transitions are triggered by the EA editing the original label ℓ' of a transition of \mathcal{S} to a non-empty sequence of labels $\langle \ell \rangle \cdot \ell$. In this case, a first transition labeled by ℓ is performed immediately in \mathcal{S} , while ℓ is buffered;
- buf transitions are performed while the buffer of the enforced LTS is not empty (step_0 and step_+ are disallowed in this case), performing one transition labeled with the first label in the buffer and removing it from the buffer.

This model allows PDPs to interrupt system execution: when the system can take a step $s_S \xrightarrow{\ell'} s'_S$ but there is no ℓ such that $s_\delta \xrightarrow{\ell' \triangleright \ell} s'_\delta$, the execution is blocked.

Example 2. We compose the SuE over \mathcal{E} in Fig. 2 with the PDPs P_{Consent} and P_{Deletion} in Fig. 3. Consider our LTS on \mathcal{E} in state *Timeline* (i.e., user is logged in) running together the PDP P_{Consent} in state *RVK* (i.e., user consent was revoked or never given). If the SuE attempts *Use*, the PDP receives *Use* in *RVK* and suppresses it by taking the step $RVK \xrightarrow{\text{Use} \triangleright \varepsilon} RVK$; by rule *step*₀, the state of the enforced LTS is left unchanged. Now, compose our LTS on \mathcal{E} in the state *Timeline* with the PDP P_{Delete} . Assume that the LTS just executed *Request*, which put the PDP in state *Wait*. If the SuE attempts *tick*, the PDP takes a step $Wait \xrightarrow{\text{Tick} \triangleright \langle \text{Delete}, \text{Tick} \rangle} \delta$ *Base*, inserting *Delete*; the SuE executes $Timeline \xrightarrow{\text{Delete}} Timeline$ and buffers $\langle \text{Tick} \rangle$ by rule *step*₊; finally, since the buffer is not empty, the SuE takes a transition $Timeline \xrightarrow{\text{Tick}} Timeline$ by rule *buf*.

A PDP's correctness is expressed in terms of two well-known notions [21]: *soundness* and *transparency*. Soundness with respect to some property P states that any trace edited by the PDP is in P ; transparency states that traces that already adhere to P are not modified by the PDP.

Definition 5 (Soundness). *An edit automaton δ over $(\mathcal{L}, \mathcal{L})$ is sound with respect to a property $P \subseteq \mathbb{T}_{\mathcal{L}}$ iff for any $\sigma \in \mathbb{T}_{\mathcal{L}}$, $\delta(\sigma) \subseteq P$.*

Definition 6 (Transparency). *An edit automaton δ over $(\mathcal{L}, \mathcal{L})$ is transparent with respect to a property $P \subseteq \mathbb{T}_{\mathcal{L}}$ iff for any $\sigma \in P$, $\delta(\sigma) = \sigma$.*

Next, we define the following variant of soundness: we say a PDP is *prefix-sound* if all prefixes of a trace edited by that PDP are in P . Prefix-soundness is stronger than standard soundness.

Definition 7 (Prefix-Soundness). *An edit automaton δ over $(\mathcal{L}, \mathcal{L})$ is prefix-sound with respect to $P \subseteq \mathbb{T}_{\mathcal{L}}$ iff for any $\sigma \in \mathbb{T}_{\mathcal{L}}$, $\text{pre}(\delta(\sigma)) \subseteq P$.*

Runtime enforcement aims to ensure the compliance of the enforced LTS with some P . Another common requirement is that, if an execution of the original LTS already fulfills P , then this execution is not altered by the enforcement mechanism. The corresponding properties of the *enforced system* are as follows:

Definition 8 (Compliance). *An LTS $\mathcal{S} = (\mathbb{S}, \mathcal{L}, s^0, \dot{\rightarrow})$ complies with the property P iff for any execution $s^0 \xrightarrow{\sigma} s$, we have $\sigma \in P_{\mathcal{L}}$.*

Definition 9 (Preservation). *An LTS $\mathcal{S} = (\mathbb{S}, \mathcal{L}, s^0, \dot{\rightarrow})$ preserves the property $P_{\mathcal{L}}$ with respect to an LTS $\mathcal{S}_* = (\mathbb{S}_*, \mathcal{L}, s_*^0, \dot{\rightarrow}_*)$ iff for any execution $s_*^0 \xrightarrow{\sigma_*} s_*$ of \mathcal{S}_* such that $\sigma \in P_{\mathcal{L}}$, we have an execution $s^0 \xrightarrow{\sigma} s$ of \mathcal{S} .*

The following theorems provide sufficient conditions for compliance and preservation. The first theorem shows that if a PDP δ is prefix-sound with respect to P , then any enforced system $\langle \mathcal{S} | \delta \rangle$ complies with P . The second theorem shows that if a PDP δ is transparent with respect to $\text{pre}(P)$, then any enforced system $\langle \mathcal{S} | \delta \rangle$ preserves P with respect to \mathcal{S} . These theorems confirm that PDP soundness and transparency are useful properties as they guarantee that any system composed with a PDP complies to its specification. The first condition is necessary and sufficient. The second condition is sufficient, but not necessary.

Theorem 1. *The edit automaton δ is prefix-sound with respect to property P iff for any LTS \mathcal{S} , the LTS $\langle \mathcal{S} | \delta \rangle$ complies with P .*

Theorem 2. *Let $P \subseteq \mathbb{T}_{\mathcal{L}}$. If an automaton δ is transparent with respect to $\text{pre}(P)$, then for any LTS \mathcal{S} , the LTS $\langle \mathcal{S} | \delta \rangle$ preserves P with respect to \mathcal{S} .*

As a corollary of Theorem 1, there exist sound (but not *prefix-sound*) edit automata that fail to ensure compliance when composed with certain systems. Such failures occur when an automaton inserts labels to restore compliance with P , but in doing so, creates intermediate trace prefixes that are not in P .

For example, let $\mathcal{L} = \{a, b\}$ and $P = \{\langle a, b \rangle\}$. Consider an edit automaton δ that (1) replaces the first symbol of any trace with $\langle a, b \rangle$, and then (2) blocks further execution. This automaton is sound with respect to P . When composed with a system, δ forces the system to execute a followed by b . However, after the first step, the system produces the intermediate trace $\langle a \rangle$, which is not in P . Thus, the enforced LTS does not comply with P .

Similarly, there exist edit automata that are transparent with respect to P (but not with respect to $\text{pre}(P)$) and fail to ensure preservation. Using the same \mathcal{L} and P , consider an edit automaton δ such that $\delta(\langle a \rangle) = \varepsilon$, $\delta(\langle b \rangle) = \varepsilon$, and $\delta(\langle a, b \rangle) = \langle a, b \rangle$. This automaton is transparent with respect to P . However, when composed with a system that has only two transitions $s_0 \xrightarrow{a} s_1 \xrightarrow{b} s_2$, this edit automaton prevents the generation of $\langle a, b \rangle$ by the enforced LTS, as it suppresses any initial a and thus prevents the LTS from reaching s_1 .

To see why Theorem 2 is not an equivalence, consider the property $P = \{\varepsilon, \langle a \rangle, \langle a, a \rangle, \langle a, a, a \rangle, \dots\}$ and a PDP δ with a single state s_δ^0 and transitions $s_\delta^0 \xrightarrow{\ell \triangleright \langle a, a \rangle} s_\delta^0$ for all ℓ . Since δ only ever inserts more a 's, any trace $\sigma = \langle a, a, \dots \rangle \in P$ of a system \mathcal{S} is also a trace of $\langle \mathcal{S} | \delta \rangle$. However, δ is not transparent since, for example, $\delta(\langle a \rangle) = \langle a, a \rangle \neq \langle a \rangle$.

We conclude this discussion by noting that for prefix-closed properties, i.e., properties P such that $\text{pre}(P) = P$, Theorems 1 and 2 can be expressed in terms of standard soundness and transparency with respect to P .

3 The Extended Enforcement Model

We now extend the classical model just presented to a model that explicitly distinguishes between low-level system actions and high-level PDP events. To

this end, we first define a formal notion of a PEP that mediates between these two levels and introduce new notions of PEP soundness and transparency (Sect. 3.1); then, we describe the three-way composition of an SuE, PEP, and PDP and provide analogues of Theorems 1 and 2 in this extended setting (Sect. 3.2).

3.1 Formal Model of the Policy Enforcement Point (PEP)

Let \mathcal{A} be a set of actions and \mathcal{E} a set of events. A PEP η is a pair of edit automata, the *instrumentor* η_i and the *enforcer* η_e , that perform Steps ③ and ⑦ in Fig. 1 (bottom). The *instrumentor* η_i maps actions to sequences of events, while the *enforcer* η_e maps events back to sequences of actions. For the PEP to serve as a transducer between traces of actions and traces of events as they occur in the system, (i) the PEP's state after Step ⑥ may depend on the edited events it received in Step ⑤, but not on the attempted actions it had mapped in Step ①. Moreover, (ii) if the sequence of edited actions at the end of Step ⑥ is empty, the PEP's state must remain the same as before Step ①. Formally:

Definition 10. A PEP over $(\mathcal{A}, \mathcal{E})$ is a pair $\eta = (\eta_i, \eta_e)$, where $\eta_i = (\mathbb{S}_\eta, \mathcal{A} \times \mathcal{E}_\tau^*, s_\eta^0, \xrightarrow{\cdot} \rightarrow_{\eta,i})$ is an edit automaton over $(\mathcal{A}, \mathcal{E})$ and $\eta_e = (\mathbb{S}_\eta, \mathcal{E} \times \mathcal{A}_\tau^*, s_\eta^0, \xrightarrow{\cdot} \rightarrow_{\eta,e})$ is an edit automaton over $(\mathcal{E}, \mathcal{A})$, with the same state space, such that for any

$$s_\eta^1 \xrightarrow{a'_1 \triangleright e'_1} \eta_i s_\eta^{2,1} \xrightarrow{e_1 \triangleright \alpha_1} \eta_e s_\eta^{3,1} \quad s_\eta^1 \xrightarrow{a'_2 \triangleright e'_2} \eta_i s_\eta^{2,2} \xrightarrow{e_2 \triangleright \alpha_2} \eta_e s_\eta^{3,2},$$

with $\circ \alpha_1 = \circ \alpha_2$, then (i) $s_\eta^{3,1} = s_\eta^{3,2}$, and (ii) if $\circ \alpha_1 = \varepsilon$, then $s_\eta^{3,1} = s_\eta^1$.

Definition 11. A PEP η is input-enabled iff both η_i and η_e are input-enabled.

For each sequence of alternating η_i and η_e transitions $s_\eta^0 \xrightarrow{a'_1 \triangleright e'_1} \eta_i s_\eta^1 \xrightarrow{\sigma_1 \triangleright \rho_1} \eta_e s_\eta'^1 \xrightarrow{a'_2 \triangleright e'_2} \eta_i s_\eta^2 \xrightarrow{\sigma_2 \triangleright \rho_2} \eta_e \dots \rightarrow s_\eta'^n$, we denote by $s_\eta^0 \xrightarrow{\sigma \triangleright \rho_1 \dots \rho_n} \eta s_\eta^n$ the sequence of transitions generating the action trace $\circ(\rho_1 \cdot \dots \cdot \rho_n)$ and the event trace $\circ \sigma$.

Example 3. The PEP in Fig. 3 mediates between \mathcal{A} and \mathcal{E} , mapping the intercepted SuE actions into sequences of PDP events and the events edited by the PDP back into SuE actions. Note that (1) the PEP maps `login` and `logout`, which are irrelevant for enforcement, to ε ; (2) the instrumentor non-injectively maps `read` and `write` to `Use`, going into state *Read* or *Write*, which allows the enforcer to transparently generate the original action if the PDP does not modify it.

3.2 A More Realistic Enforcement Model

Having formalized the PEP, we now compose it with an LTS and PDP:

Definition 12. An LTS \mathcal{S} over \mathcal{A} , a PEP η over $(\mathcal{A}, \mathcal{E})$, and a PDP δ over \mathcal{E} can be composed into an LTS $\langle \mathcal{S} | \eta | \delta \rangle = (\mathbb{S}_{\langle \mathcal{S} | \eta | \delta \rangle}, \mathcal{L}, s_{\langle \mathcal{S} | \eta | \delta \rangle}^0, \dot{\rightarrow}_{\langle \mathcal{S} | \eta | \delta \rangle})$ by

$$\begin{array}{c}
\mathbb{S}_{\langle \mathcal{S} | \eta | \delta \rangle} \triangleq \mathbb{S}_{\mathcal{S}} \times \mathbb{S}_{\eta} \times \mathbb{S}_{\delta} \times \mathcal{L}^* \quad s_{\langle \mathcal{S} | \eta | \delta \rangle}^0 \triangleq (s_{\mathcal{S}}^0, s_{\eta}^0, s_{\delta}^0, \varepsilon) \\
\\
\frac{s_{\mathcal{S}} \xrightarrow{\tau} s'_{\mathcal{S}}}{(s_{\mathcal{S}}, s_{\eta}, s_{\delta}, b) \xrightarrow{\tau}_{\langle \mathcal{S} | \eta | \delta \rangle} (s'_{\mathcal{S}}, s_{\eta}, s_{\delta}, b)} \text{step}_{\tau} \quad \frac{s_{\mathcal{S}} \xrightarrow{a' \neq \tau} s''_{\mathcal{S}} \quad s_{\eta} \xrightarrow{a' \triangleright e'} \eta_i \quad s''_{\eta} \quad s_{\delta} \xrightarrow{e' \setminus \tau \triangleright e} \delta \quad s'_{\delta}}{s''_{\eta} \xrightarrow{e' \setminus \tau \triangleright e} \eta_e \quad s'_{\eta} \quad e' \setminus \tau \neq \varepsilon} \text{step}_0}{(s_{\mathcal{S}}, s_{\eta}, s_{\delta}, \varepsilon) \xrightarrow{\tau}_{\langle \mathcal{S} | \eta | \delta \rangle} (s_{\mathcal{S}}, s'_{\eta}, s'_{\delta}, \varepsilon)} \text{step}_0 \\
\\
\frac{s_{\mathcal{S}} \xrightarrow{a' \neq \tau} s''_{\mathcal{S}} \quad s_{\eta} \xrightarrow{a' \triangleright e'} \eta_i \quad s''_{\eta} \quad s_{\delta} \xrightarrow{e' \setminus \tau \triangleright e} \delta \quad s'_{\delta} \quad s''_{\eta} \xrightarrow{e' \setminus \tau \triangleright ((a), a)} \eta_e \quad s'_{\eta}}{s_{\mathcal{S}} \xrightarrow{a} s'_{\mathcal{S}} \quad e' \setminus \tau \neq \varepsilon} \text{step}_{+} \quad \frac{s_{\mathcal{S}} \xrightarrow{a' \neq \tau} s'_{\mathcal{S}} \quad s_{\eta} \xrightarrow{a' \triangleright e'} \eta_i \quad s'_{\eta} \quad e' \setminus \tau = \varepsilon}{(s_{\mathcal{S}}, s_{\eta}, s_{\delta}, \varepsilon) \xrightarrow{\tau}_{\langle \mathcal{S} | \eta | \delta \rangle} (s'_{\mathcal{S}}, s_{\eta}, s_{\delta}, \varepsilon)} \text{step}_{\varepsilon} \\
\\
\frac{s_{\mathcal{S}} \xrightarrow{a} s'_{\mathcal{S}}}{(s_{\mathcal{S}}, s_{\eta}, s_{\delta}, (a) \cdot a) \xrightarrow{a}_{\langle \mathcal{S} | \eta | \delta \rangle} (s'_{\mathcal{S}}, s_{\eta}, s_{\delta}, a)} \text{buf}
\end{array}$$

The states of the enforced LTS $\langle \mathcal{S} | \eta | \delta \rangle$ have four components, and now also include the state of the PEP η . Rules step_{τ} and buf are similar to before, while step_0 and step_{+} incorporate the mediation through η_i and η_e . We add a fifth rule $\text{step}_{\varepsilon}$ that covers the case where the instrumentor generates an empty sequence of events in response to an action, leading the enforced system to take the attempted transition without any intervention from δ .

Example 4. We compose our LTS on \mathcal{A} with the PDP P_{Delete} and the PEP in Fig. 3. Assume that the SuE is in state *Timeline* and just executed a transition labeled by **request**, putting the PDP in state *Wait*. The PEP is in state *Base*. If the SuE attempts **tick**, the PEP takes a step $\text{Base} \xrightarrow{\text{tick} \triangleright \langle \text{Tick} \rangle} \eta_i \text{Base}$; the PDP takes a step $\text{Wait} \xrightarrow{\text{Tick} \triangleright \langle \text{Delete}, \text{Tick} \rangle} \delta \text{Base}$, inserting **delete**; the PEP takes steps $\text{Base} \xrightarrow{\text{Delete} \triangleright \text{delete}} \eta_e \text{Base} \xrightarrow{\text{Tick} \triangleright \text{tick}} \eta_e \text{Base}$; the SuE executes *Timeline* $\xrightarrow{\text{delete}} \text{Timeline}$ and buffers $\langle \text{tick} \rangle$ by rule step_{+} ; since the buffer is non-empty, the SuE takes a transition $\text{Timeline} \xrightarrow{\text{tick}} \text{Timeline}$ by rule buf .

Now, we compose our LTS with the PDP P_{Consent} and the PEP as above, with the PDP in state *RVK*. If, in state *Timeline*, the SuE attempts **write**, then the PEP maps $\text{Base} \xrightarrow{\text{write} \triangleright \text{Use}} \text{Write}$; the PDP receives use without prior consent and suppresses it, taking a step $\text{RVK} \xrightarrow{\text{Use} \triangleright \varepsilon} \text{RVK}$; consequently, according to the rule step_0 , the SuE does not take any transition.

The PEP mediates between traces of system actions and traces of PDP events. Hence, its correctness can only be assessed with regard to a mapping between these two sets of traces, which is part of the system's specification. We call such a mapping a *trace mediator*. We require trace mediators to be monotonic with respect to the prefix relation $\sigma' \in \text{pre}(\sigma)$.

Definition 13. A trace mediator is a function $I : \mathcal{A}^* \rightarrow \mathcal{E}^*$ such that $\forall \rho, \rho' \in \mathcal{A}^* . \rho \in \text{pre}(\rho') \implies I(\rho) \in \text{pre}(I(\rho'))$.

In practice, this is usually desirable, since otherwise I could arbitrarily map extensions of a given action trace to shorter or incomparable event traces.

We adopt the following definition of soundness. We later show that this definition ensures compliance of the three-way composition:

Definition 14. A PEP η over $(\mathcal{A}, \mathcal{E})$ is sound with respect to a trace mediator $I : \mathcal{A}^* \rightarrow \mathcal{E}^*$ iff whenever $s_\eta^0 \xrightarrow{\sigma \triangleright \rho} s'_\eta$, then $I(\circ \rho) \in \text{pre}(\circ \sigma)$.

Note that the above definition only requires the action trace $I(\circ \rho)$ to be a prefix of the event trace $\circ \sigma$, rather than equal to it. Intuitively, we observe that, if the mapped action trace $I(\rho)$ is always a prefix of the event trace σ , then the fact that $\text{pre}(\sigma) \subseteq P$ guarantees $I(\rho) \in P$.

Similarly, for transparency, we require that the instrumentor maps the action trace to a prefix of its image by I and that, whenever the edit automaton serving as a PDP does not modify the sequence of events in Step (4), the enforcer returns the same action that was originally passed to the instrumentor.

Definition 15. A PEP η is transparent with respect to a trace mediator I iff it is input-enabled and, whenever $s_\eta^0 \xrightarrow{\sigma \triangleright \rho} s'_\eta \xrightarrow{a' \triangleright e'}_{\eta, i} s''_\eta \xrightarrow{e \triangleright \alpha}_{\eta, e} s'''_\eta$, then $\circ \sigma \cdot e' \in \text{pre}(I(\circ \rho \cdot a'))$ and $e' = e \implies \circ \alpha = \langle a' \rangle$.

The following theorems provide analogues for the correctness results in Theorems 1–2 for our three-way composition. They show that an edit automaton δ is prefix-sound with respect to P if and only if its composition with a sound PEP η with respect to a trace mediator I and an arbitrary LTS complies with $I^{-1}(P)$, and, similarly, that an edit automaton δ that is transparent with respect to $\text{pre}(P)$ guarantees that such an enforced system preserves $I^{-1}(P)$.

Theorem 3. An edit automaton δ over \mathcal{E} is prefix-sound with respect to P iff for any PEP η over $(\mathcal{A}, \mathcal{E})$ that is sound with respect to I and LTS \mathcal{S} over \mathcal{A} , the LTS $\langle \mathcal{S} | \eta | \delta \rangle$ complies with $I^{-1}(P) \triangleq \{\rho \in \mathcal{A}^* \mid I(\rho) \in P\}$.

Theorem 4. For any edit automaton δ over \mathcal{E} that is transparent with respect to $\text{pre}(P)$, for any PEP η over $(\mathcal{A}, \mathcal{E})$ that is transparent with respect to I , and for any LTS \mathcal{S} over \mathcal{A} , the LTS $\langle \mathcal{S} | \eta | \delta \rangle$ preserves $I^{-1}(P)$ with respect to \mathcal{S} .

Note that Theorems 3–4 do not show that our definition of PEP soundness is ‘minimal.’ Under Theorem 3, there could in theory exist weaker notions of PEP soundness that would still ensure that any three-way composition is sound with respect to the PDP’s property and PEP’s trace mediator. Similarly, there could exist weaker notions of PEP transparency that still guarantee preservation. The following theorems show that such weaker definitions cannot exist:

Theorem 5. Assume that $|\mathcal{E}| \geq 2$. A PEP η over $(\mathcal{A}, \mathcal{E})$ is sound with respect to I iff for any property P , for any edit automaton δ over \mathcal{E} prefix-sound with respect to P , and for any LTS \mathcal{S} over \mathcal{A} , the LTS $\langle \mathcal{S} | \eta | \delta \rangle$ complies with $I^{-1}(P)$.

Theorem 6. *Assume that $|\mathcal{A}| \geq 2$. A PEP η over $(\mathcal{A}, \mathcal{E})$ is transparent with respect to I iff for any property P , for any edit automaton δ transparent with respect to $\text{pre}(P)$, and for any LTS S , the LTS $\langle S|\eta|\delta \rangle$ preserves $I^{-1}(P)$ with respect to S .*

4 Enforcement of Downward-Closed Properties

In the previous section, we have described how using a PEP that is sound with respect to a trace mediator I and a prefix-sound PDP for P ensures system compliance with $I^{-1}(P)$. In practice, however, PEPs that are not generally sound can still ensure compliance with more restricted classes of properties. For example, for property P_{Consent} above, a PEP preventing *too many read* actions can still guarantee compliance. In this case, what allows compliant behavior despite an unsound PEP is that P_{Consent} is *downward-closed* with respect to the relation $\preceq_{\text{Use-}}$ such that $\sigma \preceq_{\text{Use-}} \sigma'$ iff σ is obtained from σ' by removing **Use** events.

An “overapproximating” PEP such as the above has at several potential advantages: it may induce less runtime overhead because it produces fewer events; its implementation might be easier to audit for soundness as some checks (e.g., checking that the PEP does not prevent too many **read** actions) may become unnecessary. Next, we study this overapproximation by considering all properties $P \subseteq \mathcal{E}^*$ that are downward-closed with respect to some fixed relation \preceq on \mathcal{E}^* :

Definition 16 (Downward Closure). *Let $\preceq \subseteq \mathcal{E}^* \times \mathcal{E}^*$. A property $P \subseteq \mathcal{E}^*$ is downward-closed under \preceq iff $\forall \sigma, \sigma'. \sigma' \in P \wedge \sigma \preceq \sigma' \implies \sigma \in P$.*

The following theorem follows straightforwardly from this definition:

Theorem 7. *Let I and I' be two trace mediators such that $\forall \rho. I'(\rho) \preceq I(\rho)$. If a system S complies with $I(\rho)$, it complies with $I'(\rho)$. If S preserves $I'(\rho)$ with respect to some S_* , it preserves $I(\rho)$ with respect to S_* .*

In the above example, one consequence of this theorem is that if we have a PEP η that is sound with respect to some I' such that I' produces *fewer consent* events than I , then any $\langle S|\eta|\delta_{\text{Consent}} \rangle$ satisfies $I^{-1}(P_{\text{Consent}})$. This is because the property P_{Consent} is downward-closed with respect to the relation $\preceq_{\text{Consent+}}$ such that $\sigma \preceq_{\text{Consent+}} \sigma'$ iff σ is obtained from σ' by adding **Consent** events.

Next, we consider the following notion of \preceq -soundness for PEPs:

Definition 17. *A PEP η is \preceq -sound with respect to I iff whenever $s_\eta^0 \xrightarrow{\sigma \triangleright \rho} s'_\eta$, then $\circ\sigma$ is a prefix of some σ' with $\sigma' \preceq I(\circ\rho)$.*

The next theorem formalizes the overapproximation discussed above in the case of **read**: if a PEP η generates sequences of actions that are smaller (under \preceq) than the sequences of events edited by the PDP, then compliance is guaranteed.

Theorem 8. *Let η be a PEP over $(\mathcal{A}, \mathcal{E})$ and δ an edit automaton over \mathcal{E} sound with respect to P . If η is \preceq -sound with respect to I and P is downward-closed under \preceq , then $\langle S|\eta|\delta \rangle$ complies with $I^{-1}(P)$.*

reads and writes, user inputs and outputs, and calls to class members. The library allows developers to specify the PEP logic as in Algorithm 1 by providing an instrumentation mapping and handlers. Its architecture is shown in extended report [19]. It has bindings to the state-of-the-art ENFGUARD tool [17].

Minitwitter. We now showcase the usage of INSTRLIB by instrumenting a [micro-blogging app](#) for compliance with two privacy requirements. The target app has 435 lines of Python code and allows users to view their and other users’ timeline, follow other users, and post short messages. Additionally, the app shows one of two advertisement messages on the user’s timeline depending on the content of their posts. It also displays a privacy banner that prompts the user to accept or reject the use of their data for marketing purposes.

We enforce the two following requirements: (1) whenever data is used for marketing purposes, the user has given (and not revoked) consent; (2) if the user requests the deletion of all of their data, their data is deleted within one minute. Here, the events of interest are $\text{use}(u, p)$, denoting “user u ’s data is used for purpose p ,” $\text{consent}(u, p)$ (respectively, $\text{revoke}(u, p)$), denoting “user u gives (respectively, revokes) consent to use their data for purpose p ,” $\text{request}(u)$, denoting “user u requests deletion of their data;” and $\text{delete}(u)$, denoting “user u ’s data is deleted.” The PDP is assumed to be able to cause delete and suppress use. The trace mediator I , which is part of the system specification, is shown in Table 1. As in most practical instances, this trace mediator is informal.

Table 1. The trace mediator I in Minitwitter

Actions	Event	Sets
Any reading or writing of some of user u ’s data for purpose p	$\text{Use}(u, p)$	S, M^-
Any user input from u giving consent for purpose p	$\text{Consent}(u, p)$	M^+
Any user input from u revoking consent for purpose p	$\text{Revoke}(u, p)$	M^-
Any user input from u containing a deletion request	$\text{Request}(u)$	M^-
Any call to a function that deletes all of user u ’s data	$\text{Delete}(u)$	C, M^+

To instrument Minitwitter, developers proceed in three steps. *First*, they provide the property of interest in ENFGUARD’s [17] property specification language: Metric First-Order Temporal Logic (MFOTL) [8]. Here, the property is

$$\square(\forall u. (\text{Use}(u, \text{“marketing”}) \rightarrow (\neg \text{Revoke}(u, \text{“marketing”}) \text{ S } \text{Consent}(u, \text{“marketing”})))) \wedge (\text{Request}(u) \rightarrow \diamond_{[0,60]} \text{Delete}(u))).$$

Second, the developers use the built-in Django bindings for INSTRLIB to associate actions to database reads and writes (actions `read` and `write`), user inputs

to views (action `input`), and function calls (action `execute`). Functions with a specific processing can be marked with that purpose (here, “marketing”). At runtime, based on the current call stack, INSTRLIB injects the current purposes of processing into `read` and `write` actions. This step requires about 40 lines of code in the Python files describing the app’s database models and URLs.

Third, the developers describe the instrumentation mapping and handlers. This requires about 50 lines of code in a single Python file. The instrumentation mapping maps database `read` or `writes` to `Use` events, consent banner clicks (captured by specific `input` actions) to either `consent` or `revoke`, clicks on a special ‘Delete My Data’ button (captured by other `input` actions) to `request`, and executions of a special function `delete_data(u)` that erases all of a user’s data (captured by an `execute` action) to `Delete`. Two handlers are implemented: a suppression handler for `use` that returns `None` instead of the actual content of object fields and prevents their overwriting; and a causation handler for `delete` that calls `delete_data`. By default, INSTRLIB provides a simple preservation handlers as described in Sect. 5.2, which are sufficient when i_η maps each action to at most one event. All enforcement-related code is showed in extended report [19].

In Table 2, we report the latency of four of Minitwitter’s views with and without instrumentation with INSTRLIB: viewing the `timeline`, `posting` a message, giving `consent`, and `requesting` deletion of one’s data, for different values of the number n of posts in the database. The runtime overhead is < 15 ms per request.

Table 2. Runtime latency (ms) over 20 repetitions

View	Baseline ($\propto n$)						Instrumented ($\propto n$)			
	10^2	10^3	10^4	10^5	10^6	10^2	10^3	10^4	10^5	10^6
timeline	54	54	58	58	66	56 +2	58 +4	69 +11	70 +12	75 +9
post	64	62	63	62	61	67 +4	68 +6	66 +3	67 +5	67 +6
consent	47	46	47	47	47	53 +6	54 +8	55 +8	54 +7	54 +7
request	–	–	–	–	–	58	59	59	58	72

Auditing Minitwitter’s Implementation. In the property above, `consent` and `delete` are monotonic, whereas `request`, `revoke`, and `use` are antimonotonic. We can now go through the checklist provided by Theorem 11. For (1–3), we check the existence of a causation handler for `delete`, a suppression handler for `use`, and preservation handlers for all events. Regarding preservation handlers, we note that, as described above, INSTRLIB’s default implementation provides simple preservation handlers that are sufficient with our choice of i_η —this also allows us to check (6). Condition (4) is implemented by INSTRLIB by design. For the

delete causation handler, we answer (5a) positively by checking that the handler does call the `delete_data` function, whose behavior matches the informal description in Table 1. Condition (5b) is vacuous since `delete` is monotonic and causable. Condition (7) is trivially fulfilled since our suppression handlers return `None`. For (8a), we must check that `request`, `use`, and `revoke` events are always emitted when actions occur that map to them according to Table 1. Inspecting the interface of the application, we control that the buttons for revocation of consent and deletion requests map to the views whose `inputs` we have instrumented. Similarly, we check that all fields that contain personal data emit `read` and `write` events and that all functions performing marketing are marked as such. Finally, for (8b), we must check that `consent` and `delete` are only logged when the corresponding system actions as described in Table 1 happen. To this end, we control that `consent` is only generated by the `input` corresponding to clicking ‘yes’ in the banner and, similarly, that the `delete` event is only generated by the `execute` action of function `delete_data`.

7 Conclusions and Future Work

To the best of our knowledge, we have provided the first formal account of instrumentation in runtime enforcement. Besides a policy decision point (PDP), our extended enforcement model features a policy enforcement point (PEP) as an explicit component. Our model is general, independent of any specific PDP, and provides necessary and sufficient conditions for the correctness of the composition of a system, a PDP, and a PEP. We have demonstrated the applicability of our approach by implementing in the INSTRLIB instrumentation library and using it to enforce privacy requirements in a micro-blogging application.

Future work includes extending our auditing methodology to validate the implementation of runtime enforcement mechanisms in large applications with complex specifications and further optimizing INSTRLIB.

Acknowledgments. François Hublet is supported by the Swiss National Science Foundation grant “Model-driven Security & Privacy” (204796). We thank the anonymous RV reviewers for their insightful feedback.

References

1. Aceto, L., Cassar, I., Francalanza, A., Ingólfssdóttir, A.: On runtime enforcement via suppressions. In: Schewe, S., Zhang, L. (edis.) Conference on Concurrency Theory (CONCUR 2018), Leibniz International Proceedings in Informatics (LIPIcs), pp. 34:1–34:17, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2018)
2. Aceto, L., Cassar, I., Francalanza, A., Ingólfssdóttir, A.: Bidirectional runtime enforcement of first-order branching-time properties. *Logical Methods Comput. Sci.* **19** (2023)

3. Alur, R., Dill, D.L.: A theory of timed automata. *Theor. Comput. Sci.* **126**(2), 183–235 (1994)
4. Bai, G., Gu, L., Feng, T., Guo, Y., Chen, X.: Context-aware usage control for android. In: Jajodia, S., Zhou, J. (eds.) *SecureComm 2010*. LNCS, vol. 50, pp. 326–343. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16161-2_19
5. Basin, D., Klaedtke, F., Müller, S., Zalinescu, E.: Monitoring metric first-order temporal properties. *J. ACM (JACM)* **62**(2), 15:1–15:45 (2015)
6. Cassar, I., Francalanza, A., Aceto, L., Ingólfssdóttir, A.: Developing theoretical foundations for runtime enforcement. *CoRR* [arxiv:1804.08917](https://arxiv.org/abs/1804.08917) (2018)
7. Charafeddine, H., El-Harake, K., Falcone, Y., Jaber, M.: Runtime enforcement for component-based systems. In: *Symposium on Applied Computing*, pp. 1789–1796. ACM (2015)
8. Chomicki, J.: Efficient checking of temporal integrity constraints using bounded history encoding. *Trans. Datab. Syst. (TODS)* **20**(2), 149–186 (1995)
9. OASIS XACML Technical Committee. eXtensible Access Control Markup Language (XACML) Version 1.0. Technical Report oasis-xacml-1.0, OASIS (2003)
10. Erlingsson, Ú.: The inlined reference monitor approach to security policy enforcement. PhD thesis, Cornell University (2004)
11. Erlingsson, U., Schneider, F.B.: Sasi enforcement of security policies: a retrospective. In: *New Security Paradigms*, pp. 87–95 (1999)
12. Falcone, Y., Jaber, M.: Fully automated runtime enforcement of component-based systems with formal and sound recovery. *J. Softw. Tools Technol. Transf. (STTT)* **19**(3), 341–365 (2017)
13. Hublet, F., Basin, D., Krstić, S.: Real-time policy enforcement with metric first-order temporal logic. In: *European Symposium on Research in Computer Security (ESORICS)*, pp.211–232. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-17146-8_11
14. Hublet, F., Basin, D., Krstić, S.: Enforcing the GDPR. In: *European Symposium on Research in Computer Security (ESORICS)*, pp. 400–422. Springer, Heidelberg (2023). https://doi.org/10.1007/978-3-031-51476-0_20
15. Hublet, F., Basin, D., Krstić, S.: User-controlled privacy: Taint, track, and control. *Proc. Priv. Enhancing Technol.* **2024**(1), 597–616 (2024)
16. François Hublet, Alexander Kvamme, and Srdan Krstic. Towards an enforceable GDPR specification. *CoRR*, abs/2402.17350, 2024
17. Hublet, F., Lima, L., Basin, D., Krstić, S., Traytel, D.: Scaling up proactive enforcement. In: Piskac, R., Rakamarić, Z. (eds.) *Computer Aided Verification (CAV)*, pp. 370–392. Springer, Heidelberg (2025). https://doi.org/10.1007/978-3-031-98682-6_19
18. Hublet, F., Lima, L., Basin, D., Krstić, S., Traytel, D.: Proactive real-time first-order enforcement. In: Gurfinkel, A., Ganesh, V. (eds.) *Computer Aided Verification (CAV)*, vol. 14682 of LNCS, pp. 156–181. Springer, Heidelberg (2024). https://doi.org/10.1007/978-3-031-65630-9_8
19. Hublet, F., Basin, D., Hu, L., Krstić, S., Reese, L.: Instrumenting runtime enforcement. Technical report, ETH Zürich (2025). Extended version. <https://doi.org/10.5281/zenodo.16530943>
20. Hublet, F., Basin, D., Hu, L., Krstić, S., Reese, L.: *InstrLib* (2025). <https://github.com/runtime-enforcement/instrlib>
21. Ligatti, J., Bauer, L., Walker, D.: Edit automata: enforcement mechanisms for run-time security policies. *J. Inf. Secur.* **4**, 2–16 (2005)

22. Ligatti, J., Reddy, S.: A theory of runtime enforcement, with results. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 87–100. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15497-3_6
23. Lynch, N.A., Tuttle, M.R.: An introduction to input/output automata. Laboratory for Computer Science, Massachusetts Institute of Technology (1988)
24. Pinisetty, S., Falcone, Y., Jéron, T., Marchand, H., Rollet, A., Nguena Timo, O.L.: Runtime enforcement of timed properties. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 229–244. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35632-2_23
25. Rasthofer, S., Arzt, S., Lovat, E., Bodden, E.: Droidforce: enforcing complex, data-centric, system-wide policies in android. In: Availability, Reliability and Security, pp. 40–49. IEEE (2014)
26. Sarraf, S.: Optus breach occurred due to a coding error, alleges ACMA. CSO Online (2022)
27. Schneider, F.: Enforceable security policies. *Trans. Inf. Syst. Sec.* **3**(1), 30–50 (2000)