Instrumenting Runtime Enforcement

François Hublet[1], David Basin[1], Linda Hu[1], Srđan Krstić[1], and Lennard Reese[2]

[1] ETH Zürich, Zurich, Switzerland
`{francois.hublet, basin, srdan.krstic}@inf.ethz.ch, lindhu@student.ethz.ch`
[2] University of 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 INSTRLIB, 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 [25]. 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

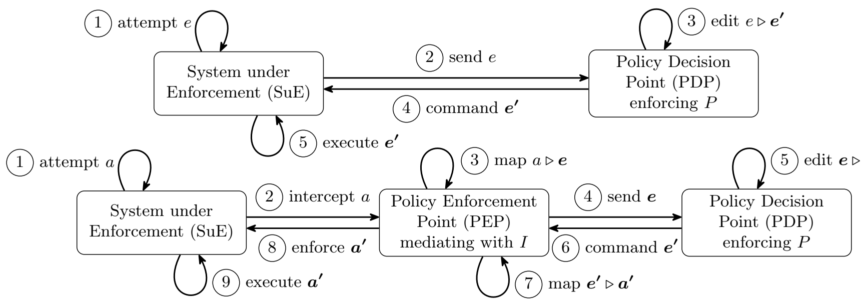

1 attempt e
System under Enforcement (SuE)
2 send e
4 command $\boldsymbol{e'}$
Policy Decision Point (PDP) enforcing P
3 edit $e \triangleright \boldsymbol{e'}$
5 execute $\boldsymbol{e'}$
1 attempt a
System under Enforcement (SuE)
2 intercept a
8 enforce $\boldsymbol{a'}$
9 execute $\boldsymbol{a'}$
3 map $a \triangleright \boldsymbol{e}$
Policy Enforcement Point (PEP) mediating with I
7 map $\boldsymbol{e'} \triangleright \boldsymbol{a'}$
4 send $\boldsymbol{e}$
6 command $\boldsymbol{e'}$
5 edit $\boldsymbol{e} \triangleright \boldsymbol{e'}$
Policy Decision Point (PDP) enforcing P

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

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 [20] the event e to a sequence of events $\boldsymbol{e'}$ compliant with P (Step (3)). The modification can involve removing, replacing, or inserting events. The events $\boldsymbol{e'}$ 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.

We propose an *extended* model, shown in Figure 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 $\boldsymbol{e}$ (Step (3)). This sequence is sent to the PDP (Step (4)), which edits $\boldsymbol{e}$ to some $\boldsymbol{e'}$ (Step (5)) and returns $\boldsymbol{e'}$ (Step (6)). The PEP maps $\boldsymbol{e'}$ back to a sequence of actions $\boldsymbol{a'}$ (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 ($\mathsf{APIAccess}(u)$ event), then u is authenticated," while the mediator I mapped both system actions `legacyAPIAccess`(u) and `modernAPIAccess`(u) to the event $\mathsf{APIAccess}(u)$. However, the PEP failed to map `legacyAPIAccess`(u) to $\mathsf{APIAccess}(u)$. As a result, although the PDP correctly enforces P, the composition of the SuE, PEP, and PDP does not provide the desired security guarantees.

Contributions. After reviewing the classical enforcement model (Section 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 (Section 3).
– We show how these conditions can be relaxed for properties where the occurrence of certain events can be soundly overapproximated (Section 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 (Section 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 (Section 6).

Related Work. Models of runtime enforcement mechanisms include security automata [11,26], edit automata [20], mandatory results automata (MRA) [21], and timed automata [3,23]. More recently, several frameworks for enforcing expressive first-order properties at runtime have also been developed [13,18,17]. 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 [4,24,14,15]. 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 (Section 2.1). We then introduce the 'classical' enforcement model and its associated notions of PDP soundness and transparency (Section 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., α, σ, ρ, ...) or bold Latin letters (e.g., $\boldsymbol{a}$, $\boldsymbol{e}$, $\boldsymbol{\ell}$, ...) to denote sequences; bold Greek letters denote sequences of sequences. We denote the empty sequence as ε and finite sequences as $\langle a_1, a_2, \ldots, a_n \rangle$. 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 $\mathsf{pre}(\sigma)$ 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 $\boldsymbol{\sigma} = \langle \sigma_1, \ldots, \sigma_n \rangle \in (A^*)^*$, we denote by $\circ \boldsymbol{\sigma} \triangleq \sigma_1 \cdot \sigma_2 \cdot \ldots \cdot \sigma_n$ 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 [1,6,2], 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 $\mathcal{L}_\tau$.*

We write $s \xrightarrow{\ell} s'$ for $(s, \ell, s') \in \dot{\rightarrow}$. An LTS execution is of the form $s^0 \xrightarrow{\ell_1} s^1 \xrightarrow{\ell_2} \ldots \xrightarrow{\ell_n} s^n$ and the trace of such an execution is $\sigma = \langle \ell_1, \ell_2, \ldots \rangle \setminus \tau$, 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) [20] 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 $\boldsymbol{\ell_2}$ from another set $\mathcal{L}^2$. If no $\boldsymbol{\ell_2}$ exists for a given ℓ_1, the execution of the LTS is terminated, similar to execution cutting in security automata [26].

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

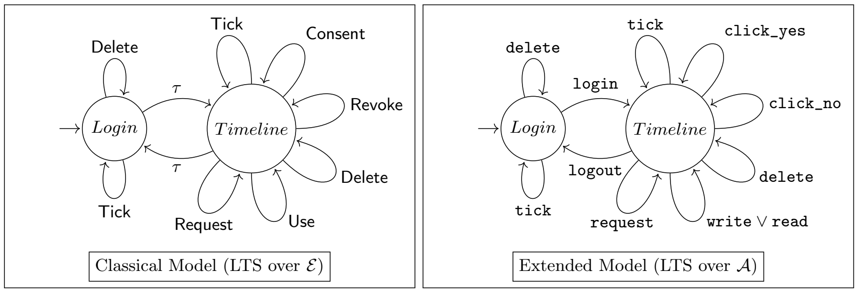


Fig. 2: Systems under Enforcement (SuE) in the Classical and Extended Model

An edit automaton is input-enabled [22] iff for any ℓ^1, the automaton can always edit ℓ^1 to some $\boldsymbol{\ell_2}$. 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 $\boldsymbol{\ell_2} \in \mathcal{L}^{2\,*}_\tau$ such that $s_\delta \xrightarrow{\ell_1 \triangleright \boldsymbol{\ell_2}} s'_\delta$.*

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 $\mathcal{L}^{2\,*}_\tau$). 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 $\boldsymbol{\ell_2}$. For $n \in \mathbb{N}$, $\xi = (x_i)_{1 \le i < n}$, and $\upsilon = (y_i)_{1 \le i < n}$, we denote by $\xi \triangleright \upsilon$ the zipped sequence $(x_i \triangleright y_i)_{1 \le i < n}$. For any EA δ, we abuse notation and write δ as a partial function such that $\delta(\sigma) = \circ\boldsymbol{\sigma'} \iff \exists s'_\delta.\ s^0_\delta \xrightarrow{\sigma \triangleright \boldsymbol{\sigma'}} s'_\delta$, reflecting the view of δ as a trace transducer.

2.2 The Classical Model of Runtime Enforcement

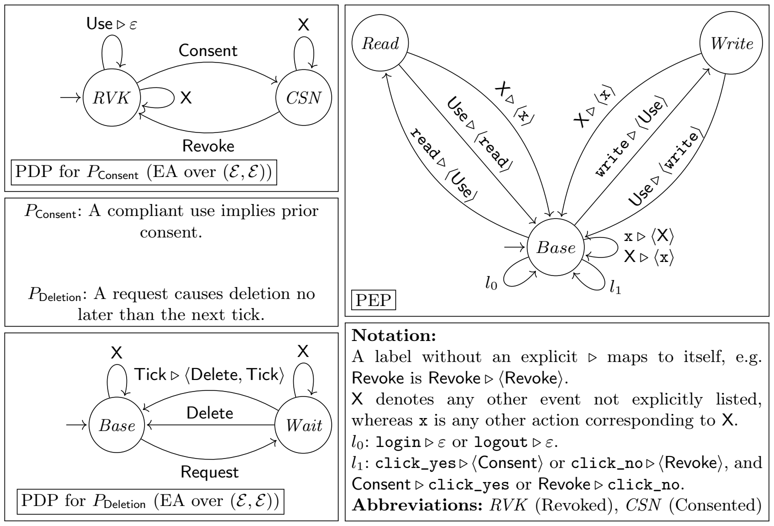
In the classical enforcement model [20,1,6], an SuE (LTS) and a PDP (edit automaton) over the same set of labels are composed, progressing in lock-step. This can be formalized as follows in terms of LTS composition:

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 $\langle \mathcal{S} | \delta \rangle = (\mathbb{S}_{\langle \mathcal{S}|\delta \rangle}, \mathcal{L}, s^0_{\langle \mathcal{S}|\delta \rangle}, \dot{\rightarrow}_{\langle \mathcal{S}|\delta \rangle})$ by*

$$\mathbb{S}_{\langle \mathcal{S}|\delta \rangle} \triangleq \mathbb{S}_\mathcal{S} \times \mathbb{S}_\delta \times \mathcal{L}^* \qquad s^0_{\langle \mathcal{S}|\delta \rangle} \triangleq (s^0_\mathcal{S}, s^0_\delta, \varepsilon)$$

$$\frac{s_\mathcal{S} \xrightarrow{\tau} s'_\mathcal{S}}{(s_\mathcal{S}, s_\delta, b) \xrightarrow{\tau}_{\langle \mathcal{S}|\delta \rangle} (s'_\mathcal{S}, s_\delta, b)}\ \mathit{step}_\tau \qquad \frac{s_\mathcal{S} \xrightarrow{\ell' \neq \tau} s''_\mathcal{S} \quad s_\delta \xrightarrow{\ell' \triangleright \varepsilon}_\delta s'_\delta}{(s_\mathcal{S}, s_\delta, \varepsilon) \xrightarrow{\tau}_{\langle \mathcal{S}|\delta \rangle} (s_\mathcal{S}, s'_\delta, \varepsilon)}\ \mathit{step}_0$$

$$\frac{s_\mathcal{S} \xrightarrow{\ell' \neq \tau} s''_\mathcal{S} \quad s_\delta \xrightarrow{\ell' \triangleright (\langle \ell \rangle \cdot \boldsymbol{\ell})}_\delta s'_\delta \quad s_\mathcal{S} \xrightarrow{\ell} s'_\mathcal{S}}{(s_\mathcal{S}, s_\delta, \varepsilon) \xrightarrow{\ell}_{\langle \mathcal{S}|\delta \rangle} (s'_\mathcal{S}, s'_\delta, \boldsymbol{\ell})}\ \mathit{step}_+ \qquad \frac{s_\mathcal{S} \xrightarrow{\ell} s'_\mathcal{S}}{(s_\mathcal{S}, s_\delta, \langle \ell \rangle \cdot \boldsymbol{\ell}) \xrightarrow{\ell}_{\langle \mathcal{S}|\delta \rangle} (s'_\mathcal{S}, s_\delta, \boldsymbol{\ell})}\ \mathit{buf}$$



P_{Consent}: A compliant use implies prior consent.

P_{Deletion}: A request causes deletion no later than the next tick.

Notation:
A label without an explicit ▷ maps to itself, e.g. Revoke is Revoke ▷ ⟨Revoke⟩.
X denotes any other event not explicitly listed, whereas x is any other action corresponding to X.
l_0: `login` ▷ ε or `logout` ▷ ε.
l_1: `click_yes` ▷ ⟨Consent⟩ or `click_no` ▷ ⟨Revoke⟩, and Consent ▷ `click_yes` or Revoke ▷ `click_no`.
Abbreviations: *RVK* (Revoked), *CSN* (Consented)

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

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:

- $step_\tau$ 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 $\langle \ell \rangle \cdot \boldsymbol{\ell}$. In this case, a first transition labeled by ℓ is performed immediately in $\mathcal{S}$, while $\boldsymbol{\ell}$ 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 $s_{\mathcal{S}} \xrightarrow{\ell'} s'_{\mathcal{S}}$ but there is no $\boldsymbol{\ell}$ such that $s_\delta \xrightarrow{\ell' \triangleright \boldsymbol{\ell}} s'_\delta$, the execution is blocked.

Example 2. We compose the SuE over $\mathcal{E}$ in Figure 2 with the PDPs P_{Consent} and P_{Deletion} in Figure 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 $RVK \xrightarrow{\mathsf{Use} \triangleright \varepsilon} RVK$; 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

$\mathit{Wait} \xrightarrow{\mathsf{Tick} \triangleright \langle \mathsf{Delete}, \mathsf{Tick} \rangle}_{\delta} \mathit{Base}$, inserting Delete; the SuE executes $\mathit{Timeline} \xrightarrow{\mathsf{Delete}} \mathit{Timeline}$ and buffers $\langle \mathsf{Tick} \rangle$ by rule step_{+}; finally, since the buffer is not empty, the SuE takes a transition $\mathit{Timeline} \xrightarrow{\mathsf{Tick}} \mathit{Timeline}$ by rule *buf*.

A PDP's correctness is expressed in terms of two well-known notions [20]: *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 $\sigma \in \mathbb{T}_{\mathcal{L}}$, $\mathsf{pre}(\delta(\sigma)) \subseteq P$.*

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 $\mathcal{S}_{\star} = (\mathbb{S}_{\star}, \mathcal{L}, s^0_{\star}, \dot{\rightarrow}_{\star})$ iff for any execution $s^0_{\star} \xrightarrow{\sigma}_{\star} s_{\star}$ of $\mathcal{S}_{\star}$ 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 $\mathsf{pre}(P)$, 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.*

Proof. $\Longrightarrow$ Assume that δ is prefix-sound with respect to P. Since δ is prefix-sound, $\mathsf{pre}(\delta(\mathbb{T}(\mathcal{S}))) \subseteq P$, hence $\mathbb{T}(\langle \mathcal{S} | \delta \rangle) \subseteq \mathsf{pre}(\delta(\mathbb{T}(\mathcal{S}))) \subseteq P$ and $\langle \mathcal{S} | \delta \rangle$ complies with P.

$\Longleftarrow$ Let σ such that $\mathsf{pre}(\delta(\sigma)) \nsubseteq P$. Let $\sigma' \in \mathsf{pre}(\delta(\sigma)) \setminus P$. Let $\mathcal{S} = \mathcal{U}_{\mathcal{L}}$ be the non-deterministic 'universal' system with state space $\{\bot\}$ that can take any transition in $\mathcal{L}$. If $\mathcal{S}$ attempts σ, then there is a partial execution of the composition $\langle \mathcal{S} | \delta \rangle$ that generates σ', showing that $\langle \mathcal{S} | \delta \rangle$ does not comply with P.

Let $\sqsubseteq$ denote prefix inclusion on traces, i.e., $\sigma \sqsubseteq \sigma' \triangleq \sigma \in \mathsf{pre}(\sigma')$.

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

Proof. Assume that δ is transparent with respect to $\mathsf{pre}(P)$, i.e., for any $\sigma \in \mathsf{pre}(P)$, $\sigma = \delta(\sigma)$. Let $s^0 \xrightarrow{\sigma} s_\star$ be an execution of $\mathcal{S}$ such that $\sigma \in P$. A straightforward induction on the prefixes $\sigma' \sqsubseteq \sigma$, which all satisfy $\sigma' = \delta(\sigma')$ by our assumption on δ, shows the existence of an execution $s^0_{\langle \mathcal{S} | \delta \rangle} \xrightarrow{\sigma} s_{\langle \mathcal{S} | \delta \rangle}$ for some $s_{\langle \mathcal{S} | \delta \rangle} \in \mathbb{S}_{\langle \mathcal{S} | \delta \rangle}$.

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 $\mathsf{pre}(P)$) 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 $P = \{\varepsilon, \langle a \rangle, \langle a, a \rangle, \langle a, a, a \rangle, \ldots\}$ and a PDP δ with a single state s^0_δ and transitions $s^0_\delta \xrightarrow{\ell \triangleright \langle a, a \rangle} s^0_\delta$ for all ℓ. Since δ only ever inserts more a's, any trace $\sigma = \langle a, a, \ldots \rangle \in P$ 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 $\mathsf{pre}(P) = P$, 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 (Section 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 (Section 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 $\textcircled{3}$ and $\textcircled{7}$ in Figure 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 $\textcircled{6}$ may depend on the edited events it received in Step $\textcircled{5}$, but not on the attempted actions it had mapped in Step $\textcircled{1}$. Moreover, (ii) if the sequence of edited actions at the end of Step $\textcircled{6}$ is empty, the PEP's state must remain the same as before Step $\textcircled{1}$. Formally:

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

$$s^1_\eta \xrightarrow{a'_1 \triangleright \boldsymbol{e}'_1}_{\eta,\mathsf{i}} s^{2,1}_\eta \xrightarrow{\boldsymbol{e}_1 \triangleright \boldsymbol{\alpha}_1}_{\eta,\mathsf{e}} s^{3,1}_\eta \qquad s^1_\eta \xrightarrow{a'_2 \triangleright \boldsymbol{e}'_2}_{\eta,\mathsf{i}} s^{2,2}_\eta \xrightarrow{\boldsymbol{e}_2 \triangleright \boldsymbol{\alpha}_2}_{\eta,\mathsf{e}} s^{3,2}_\eta,$$

with $\circ\boldsymbol{\alpha_1} = \circ\boldsymbol{\alpha_2}$, then (i) $s^{3,1}_\eta = s^{3,2}_\eta$, and (ii) if $\circ\boldsymbol{\alpha}_1 = \varepsilon$, then $s^{3,1}_\eta = s^1_\eta$.

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 $s^0_\eta \xrightarrow{a'_1 \triangleright e'_1}_{\eta,\mathsf{i}} s^1_\eta \xrightarrow{\sigma_1 \triangleright \boldsymbol{\rho}_1}_{\eta,\mathsf{e}}$ $s'^1_\eta \xrightarrow{a'_2 \triangleright e'_2}_{\eta,\mathsf{i}} s^2_\eta \xrightarrow{\sigma_2 \triangleright \boldsymbol{\rho}_2}_{\eta,\mathsf{e}} \ldots \to s'^n_\eta$, we denote by $s^0_\eta \xrightarrow{\boldsymbol{\sigma} \triangleright \boldsymbol{\rho}_1 \cdot \ldots \cdot \boldsymbol{\rho}_n}_\eta s^n_\eta$ the sequence of transitions generating the action trace $\circ(\boldsymbol{\rho}_1 \cdot \ldots \cdot \boldsymbol{\rho}_n)$ and the event trace $\circ\boldsymbol{\sigma}$.

Example 3. The PEP in Figure 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 $\langle \mathcal{S}|\eta|\delta\rangle = (\mathbb{S}_{\langle \mathcal{S}|\eta|\delta\rangle}, \mathcal{L}, s^0_{\langle \mathcal{S}|\eta|\delta\rangle}, \dot{\rightarrow}_{\langle \mathcal{S}|\eta|\delta\rangle})$ by*

$$\mathbb{S}_{\langle \mathcal{S}|\eta|\delta\rangle} \triangleq \mathbb{S}_{\mathcal{S}} \times \mathbb{S}_{\eta} \times \mathbb{S}_{\delta} \times \mathcal{L}^* \qquad s^0_{\langle \mathcal{S}|\eta|\delta\rangle} \triangleq (s^0_{\mathcal{S}}, s^0_{\eta}, s^0_{\delta}, \varepsilon)$$

$$\frac{s_{\mathcal{S}} \xrightarrow{\tau} s'_{\mathcal{S}}}{(s_{\mathcal{S}}, s_\eta, s_\delta, b) \xrightarrow{\tau}_{\langle \mathcal{S}|\eta|\delta\rangle} (s'_{\mathcal{S}}, s_\eta, s_\delta, b)}\ \mathit{step}_\tau \qquad \frac{\begin{array}{c} s_{\mathcal{S}} \xrightarrow{a' \neq \tau} s''_{\mathcal{S}} \quad s_\eta \xrightarrow{a' \triangleright \boldsymbol{e}'}_{\eta,\mathsf{i}} s''_\eta \\ s_\delta \xrightarrow{\boldsymbol{e}' \setminus \tau \triangleright \boldsymbol{e}}_{\delta} s'_\delta \\ s''_\eta \xrightarrow{\boldsymbol{e} \setminus \tau \triangleright \varepsilon}_{\eta,\mathsf{e}} s'_\eta \quad \boldsymbol{e}' \setminus \tau \neq \varepsilon \end{array}}{(s_{\mathcal{S}}, s_\eta, s_\delta, \varepsilon) \xrightarrow{\tau}_{\langle \mathcal{S}|\eta|\delta\rangle} (s_{\mathcal{S}}, s'_\eta, s'_\delta, \varepsilon)}\ \mathit{step}_0$$

$$\frac{\begin{array}{c} s_{\mathcal{S}} \xrightarrow{a' \neq \tau} s''_{\mathcal{S}} \quad s_\eta \xrightarrow{a' \triangleright \boldsymbol{e}'}_{\eta,\mathsf{i}} s''_\eta \\ s_\delta \xrightarrow{\boldsymbol{e}' \setminus \tau \triangleright \boldsymbol{e}}_{\delta} s'_\delta \quad s''_\eta \xrightarrow{\boldsymbol{e} \setminus \tau \triangleright (\langle a \rangle \cdot \boldsymbol{a})}_{\eta,\mathsf{e}} s'_\eta \\ s_{\mathcal{S}} \xrightarrow{a} s'_{\mathcal{S}} \quad \boldsymbol{e}' \setminus \tau \neq \varepsilon \end{array}}{(s_{\mathcal{S}}, s_\eta, s_\delta, \varepsilon) \xrightarrow{a}_{\langle \mathcal{S}|\eta|\delta\rangle} (s'_{\mathcal{S}}, s'_\eta, s'_\delta, \boldsymbol{a})}\ \mathit{step}_+ \qquad \frac{\begin{array}{c} s_{\mathcal{S}} \xrightarrow{a' \neq \tau} s'_{\mathcal{S}} \\ s_\eta \xrightarrow{a' \triangleright \boldsymbol{e}'}_{\eta,\mathsf{i}} s'_\eta \quad \boldsymbol{e}' \setminus \tau = \varepsilon \end{array}}{(s_{\mathcal{S}}, s_\eta, s_\delta, \varepsilon) \xrightarrow{\tau}_{\langle \mathcal{S}|\eta|\delta\rangle} (s'_{\mathcal{S}}, s_\eta, s_\delta, \varepsilon)}\ \mathit{step}_\varepsilon$$

$$\frac{s_{\mathcal{S}} \xrightarrow{a} s'_{\mathcal{S}}}{(s_{\mathcal{S}}, s_\eta, s_\delta, \langle a \rangle \cdot \boldsymbol{a}) \xrightarrow{a}_{\langle \mathcal{S}|\eta|\delta\rangle} (s'_{\mathcal{S}}, s_\eta, s_\delta, \boldsymbol{a})}\ \mathit{buf}$$

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 $\mathit{step}_\varepsilon$ 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 Figure 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 $\mathit{Base} \xrightarrow{\mathtt{tick} \triangleright \langle \mathsf{Tick} \rangle}_{\eta,\mathsf{i}} \mathit{Base}$; the PDP takes a step $\mathit{Wait} \xrightarrow{\mathsf{Tick} \triangleright \langle \mathsf{Delete}, \mathsf{Tick} \rangle}_{\delta} \mathit{Base}$, inserting Delete; the PEP takes steps $\mathit{Base} \xrightarrow{\mathsf{Delete} \triangleright \mathtt{delete}}_{\eta,\mathsf{e}} \mathit{Base} \xrightarrow{\mathsf{Tick} \triangleright \mathtt{tick}}_{\eta,\mathsf{e}} \mathit{Base}$; the SuE executes $\mathit{Timeline} \xrightarrow{\mathtt{delete}} \mathit{Timeline}$ and buffers $\langle \mathtt{tick} \rangle$ by rule step_+; since the buffer is non-empty, the SuE takes a transition $\mathit{Timeline} \xrightarrow{\mathtt{tick}} \mathit{Timeline}$ 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 $\mathit{Base} \xrightarrow{\mathtt{write} \triangleright \mathsf{Use}} \mathit{Write}$; the PDP receives Use without prior consent and suppresses it, taking a step $\mathit{RVK} \xrightarrow{\mathsf{Use} \triangleright \varepsilon} \mathit{RVK}$; 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 $\sigma' \in \mathsf{pre}(\sigma)$.

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

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 *$I : \mathcal{A}^* \to \mathcal{E}^*$ iff whenever $s^0_\eta \xrightarrow{\boldsymbol{\sigma} \triangleright \boldsymbol{\rho}} s'_\eta$, then $I(\circ\boldsymbol{\rho}) \in \mathsf{pre}(\circ\boldsymbol{\sigma})$.*

Note that the above definition only requires the action trace $I(\circ\boldsymbol{\rho})$ to be a prefix of the event trace $\circ\boldsymbol{\sigma}$, 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 $\mathsf{pre}(\sigma) \subseteq P$ 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 $\boxed{4}$, 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 $s^0_\eta \xrightarrow{\boldsymbol{\sigma} \triangleright \boldsymbol{\rho}} s'_\eta \xrightarrow{a' \triangleright \boldsymbol{e}'}_{\eta,\mathsf{i}} s''_\eta \xrightarrow{\boldsymbol{e} \triangleright \boldsymbol{\alpha}}_{\eta,\mathsf{e}} s'''_\eta$, then $\circ\boldsymbol{\sigma} \cdot \boldsymbol{e}' \in \mathsf{pre}(I(\circ\boldsymbol{\rho} \cdot a'))$ and $\boldsymbol{e}' = \boldsymbol{e} \implies \circ\boldsymbol{\alpha} = \langle a' \rangle$.*

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 $\mathsf{pre}(P)$ 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 $I^{-1}(P) \triangleq \{\rho \in \mathcal{A}^* \mid I(\rho) \in P\}$.*

Proof. $\implies$ Assume that δ is prefix-sound with respect to P, η is sound, and let ρ be a trace of $\langle \mathcal{S} | \eta | \delta \rangle$, i.e. $(s^0_{\mathcal{S}}, s^0_\eta, s^0_\delta, \emptyset) \xrightarrow{\rho} (s^n_{\mathcal{S}}, s^n_\eta, s^n_\delta, b)$. By Definition 12, there exist σ and σ' such that $s^0_\delta \xrightarrow{\sigma' \triangleright \sigma} s^n_\delta$ and $s^0_\eta \xrightarrow{\boldsymbol{\sigma} \triangleright \boldsymbol{\rho}}_\eta s^n_\eta$, where $\sigma = \circ\boldsymbol{\sigma}$ and $\rho = \circ\boldsymbol{\rho}$. Expanding the definition of δ's soundness, we get $\mathsf{pre}(\sigma) \subseteq P$. By Definition 17, since η is sound, we get that, for any $i \leq |\boldsymbol{\rho}|$, $I(\circ\boldsymbol{\rho}_{..i}) \sqsubseteq \circ\boldsymbol{\sigma}_{..i}$. In particular, $I(\rho) \in \mathsf{pre}(\sigma) \subseteq P$. We conclude that $\rho \in I^{-1}(P)$.

$\impliedby$ Assume that δ is not sound with respect to P, i.e., there exists an execution $s^0_\delta \xrightarrow{\sigma' \triangleright \sigma} s_\delta$ of δ such that $\mathsf{pre}(\sigma) \not\subseteq P$. Let $i \leq |\sigma|$ such that $\sigma_{..i} \notin P$. Fix $\mathcal{E} = \mathcal{A}$. Consider the identity PEP $\mathrm{id}_{\mathcal{A}}$ which (soundly) maps every action to itself with respect to $I = \mathrm{id}$ and let $\mathcal{S} = \mathcal{U}_{\mathcal{A}}$. Then, by Definition 12, there exists an execution $s^0_{\langle \mathcal{S} | \mathrm{id}_{\mathcal{A}} | \delta \rangle} \xrightarrow{\sigma_{..i}} s$. Since $\sigma_{..i} \notin P$, then $\langle \mathcal{S} | \mathrm{id}_{\mathcal{A}} | \delta \rangle$ does not comply with $P = I^{-1}(P)$.

Theorem 4. *For any edit automaton δ over $\mathcal{E}$ that is transparent with respect to $\mathsf{pre}(P)$, 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 $\mathcal{S}$.*

Proof. Assume that δ is transparent with respect to P, η is transparent, and let $\rho = (a_i)_{i \in \mathbb{N}} \in I^{-1}(P)$ such that $s^0 \xrightarrow{\rho} s$. By Definition 13, all prefixes of ρ are projected into $\mathsf{pre}(P)$ by I. Hence, they are not edited by δ, which is transparent with respect to $\mathsf{pre}(P)$. Using Definition 15, we build an execution $s^0_{\langle \mathcal{S}|\eta|\delta \rangle} \xrightarrow{\rho} s_{\langle \mathcal{S}|\eta|\delta \rangle}$.

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 $I^{-1}(P)$.*

Proof. $\Longrightarrow$ Identical to ($\Longrightarrow$) in the proof of Theorem 3.

$\Longleftarrow$ Assume that η is not sound. Let $(\boldsymbol{\sigma}, \boldsymbol{\rho})$ of minimal length such that $s^0_\eta \xrightarrow{\boldsymbol{\sigma} \triangleright \boldsymbol{\rho}} s'_\eta$, $s'_\eta \xrightarrow{a' \triangleright \boldsymbol{e}'}_{\eta,\mathsf{i}} s''_\eta$, $s''_\eta \xrightarrow{\boldsymbol{e} \triangleright \boldsymbol{\alpha}}_{\eta,\mathsf{e}} s'''_\eta$, and $\sigma^\dagger \triangleq I(\circ\boldsymbol{\rho} \cdot \circ\boldsymbol{\alpha}) \not\sqsubseteq \circ\boldsymbol{\sigma} \cdot \boldsymbol{e}$.

Let $P \triangleq \{\sigma' \in \mathcal{E}^* \mid \circ\boldsymbol{\sigma} \cdot \boldsymbol{e} \sqsubseteq \sigma' \vee \sigma' \sqsubseteq \circ\boldsymbol{\sigma} \cdot \boldsymbol{e}\}$ and $\mathcal{S} = \mathcal{U}_\mathcal{A}$. Clearly, the property P is not empty. Consider an edit automaton δ that enforces P by first editing the sequence of events to be exactly $\circ\boldsymbol{\sigma}$; then, when passed $\boldsymbol{e}'$, rewriting it to $\boldsymbol{e}$; and later not modifying the trace anymore. By Definition 12, there exists an execution $s^0_{\langle \mathcal{S}|\eta|\delta \rangle} \xrightarrow{\circ\boldsymbol{\rho}} (\emptyset, s'_\delta, s'_\eta, \varepsilon)$ for some s'_δ. There are two cases:

- If $\circ\boldsymbol{\sigma} \cdot \boldsymbol{e}$ is not a (proper) prefix of $\sigma^\dagger$, consider an execution of $\langle \mathcal{S} | \eta | \delta \rangle$ that performs one step from $(\emptyset, s'_\delta, s'_\eta, \varepsilon)$, generating $\sigma' = \circ\boldsymbol{\rho} \cdot \circ\boldsymbol{\alpha}$. Then $I(\sigma') \notin P$.
- If $\circ\boldsymbol{\sigma} \cdot \boldsymbol{e}$ is a proper prefix of $\sigma^\dagger$, let $f_1 \in \mathcal{E}$ and $\boldsymbol{f_2} \in \mathcal{E}^*$ such that $\sigma^\dagger = \circ\boldsymbol{\sigma} \cdot \boldsymbol{e} \cdot \langle f_1 \rangle \cdot \boldsymbol{f}_2$. Let $\boldsymbol{f} \triangleq \langle f_1 \rangle \cdot \boldsymbol{f}_2$ and $f_3 \in \mathcal{E} - \{f_1\}$. Modify P to $P' \triangleq P - \{\sigma' \in \mathbb{T}_\mathcal{E} \mid \sigma \cdot \boldsymbol{f} \sqsubseteq \sigma'\}$. Consider an edit automaton δ' that soundly enforces P' by first editing the sequence of events to be exactly $\circ\boldsymbol{\sigma}$; then, when passed $\boldsymbol{e}'$, rewriting it to $\boldsymbol{e}$; and later preventing reaching any suffix of $\circ\boldsymbol{\sigma} \cdot \boldsymbol{f}$ by editing $\boldsymbol{f}$ to $\boldsymbol{e} \cdot \langle f_3 \rangle \cdot \boldsymbol{f}_2$ if necessary. Again, consider an execution of $\langle \mathcal{S} | \eta | \delta \rangle$ that performs one step from $(\emptyset, s'_\delta, s'_\eta, \varepsilon)$, generating $\sigma' = \circ\boldsymbol{\rho} \cdot \circ\boldsymbol{\alpha}$. Then $\sigma' \notin I(P')$.

In both cases, we exhibited a system $\mathcal{S}$ and sound edit automaton δ for some property P such that $\langle \mathcal{S} | \eta | \delta \rangle$ does not comply 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 $\mathsf{pre}(P)$, and for any LTS $\mathcal{S}$, the LTS $\langle \mathcal{S} | \eta | \delta \rangle$ preserves $I^{-1}(P)$ with respect to $\mathcal{S}$.*

Proof. $\Longrightarrow$ Identical to ($\Longrightarrow$) in the proof of Theorem 4.

$\Longleftarrow$ For $\rho \in \mathcal{A}^*$, denote $\mathcal{S}_\rho \triangleq (\mathsf{pre}(\{\rho\}), \varepsilon, \dot{\rightarrow})$ the system with transitions $\rho' \xrightarrow{\ell} \rho' \cdot \ell$ for all $\rho' \cdot \ell \sqsubseteq \rho$. Let $\delta_{\mathcal{E}^*}$ be a trivial edit automaton with transitions $s^0_\delta \xrightarrow{\ell \triangleright \langle \ell \rangle} s^0_\delta$ that is transparent with respect to $\mathcal{E}^*$. If the PEP η is not input-enabled, then any execution when η blocks on a execution of some $\mathcal{S}_\rho$ under the trivial policy $\mathcal{E}^*$ and PDP $\delta_{\mathcal{E}^*}$ falsifies the preservation of $I^{-1}(P)$.

Now, assume that there exist $s^0_\eta \xrightarrow{\boldsymbol{\sigma} \triangleright \boldsymbol{\rho}} s'_\eta$, $s'_\eta \xrightarrow{a' \triangleright \boldsymbol{e}}_{\eta,\mathsf{i}} s''_\eta$, and $s''_\eta \xrightarrow{\boldsymbol{e} \triangleright \boldsymbol{a}}_{\eta,\mathsf{e}} s'''_\eta$, such that $\circ\boldsymbol{\sigma} \cdot \boldsymbol{e} \sqsubseteq I(\circ\boldsymbol{\rho} \cdot a')$, but $\boldsymbol{a} \neq \langle a' \rangle$. Let $\rho = \circ\boldsymbol{\rho}$ and consider the LTS $\mathcal{S}_{\rho \cdot a'}$. Consider an execution of $\langle \mathcal{S}_{\rho \cdot a'} | \eta | \delta_{\mathcal{E}^\omega} \rangle$ that continues from $(\rho \cdot \boldsymbol{a}, \emptyset, s'''_\eta, \varepsilon)$. There are three cases:

- If none of $\boldsymbol{a}$ and $\langle a' \rangle$ is a prefix of the other, then the system is blocked after s'''_η, preventing it to ever produce $\rho \cdot a'$.
- If $\boldsymbol{a}$ is a proper prefix of $\langle a' \rangle$, i.e., $\boldsymbol{a} = \varepsilon$, then $s'''_\eta = s'_\eta$ and the system loops, yielding the same conclusion.
- If $\langle a' \rangle$ is a proper prefix of $\boldsymbol{a}$, i.e., $\boldsymbol{a} = \langle a' \rangle \cdot \langle b_1 \rangle \cdot \boldsymbol{b_2}$ for some $b_1 \in \mathcal{A}$, $\boldsymbol{b_2} \in \mathcal{A}^*$, then consider $c \in \mathcal{A} - \{b_1\}$. Let $\rho' \triangleq \rho \cdot \langle a', c \rangle$ and consider $\langle \mathcal{S}_{\rho \cdot a' \cdot c} | \eta | \delta_{\mathcal{E}^*} \rangle$. The enforced LTS cannot generate $\rho \cdot \langle a', c \rangle$, since after generating $\rho \cdot \langle a' \rangle$, it is forced by the PEP to execute $\rho \cdot \boldsymbol{a}$.

Alternatively, assume that there exist $s^0_\eta \xrightarrow{\boldsymbol{\sigma} \triangleright \boldsymbol{\rho}} s'_\eta$, $s'_\eta \xrightarrow{a' \triangleright \boldsymbol{e}'}_{\eta,\mathsf{i}} s''_\eta$, and $s''_\eta \xrightarrow{\boldsymbol{e} \triangleright \boldsymbol{a}}_{\eta,\mathsf{e}} s'''_\eta$, $\circ\boldsymbol{\sigma} \cdot \boldsymbol{e}' \not\sqsubseteq I(\rho \cdot a')$. Let $P \triangleq I(\mathsf{pre}(\{\rho \cdot a'\}))$. Consider the LTS $\mathcal{S}_{\rho \cdot a'}$ and a transparent EA δ_P that accepts all traces in $\mathsf{pre}(P)$, blocking otherwise.

First, we show that $\circ\boldsymbol{\sigma} \cdot \boldsymbol{e}' \notin \mathsf{pre}(I(\mathsf{pre}(\{\rho \cdot a'\})))$. Towards a contradiction, assume that $\circ\boldsymbol{\sigma} \cdot \boldsymbol{e}' \in \mathsf{pre}(I(\mathsf{pre}(\{\rho \cdot a'\})))$. This means that there exists $\rho' \sqsubseteq \rho \cdot a'$ such that $\circ\boldsymbol{\sigma} \cdot \boldsymbol{e}' \sqsubseteq I(\rho')$. But then $\circ\boldsymbol{\sigma} \cdot \boldsymbol{e}' \sqsubseteq I(\rho') \sqsubseteq I(\rho \cdot a')$ since I is a trace mediator, contradicting $\circ\boldsymbol{\sigma} \cdot \boldsymbol{e}' \not\sqsubseteq I(\rho \cdot a')$. Hence, $\circ\boldsymbol{\sigma} \cdot \boldsymbol{e}' \notin \mathsf{pre}(I(\mathsf{pre}(\{\rho \cdot a'\})))$.

Since $\circ\boldsymbol{\sigma} \cdot \boldsymbol{e}' \notin P$, then the execution of $\langle \mathcal{S}_{\rho \cdot a'} | \eta | \delta_P \rangle$ generates at most the trace ρ before being blocked by δ_P, thus failing to generate $\rho \cdot a'$. Since $\rho \cdot a' \in I^{-1}(P)$, this concludes the proof.

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 $\preceq_{\mathsf{Use}-}$ such that $\sigma \preceq_{\mathsf{Use}-} \sigma'$ 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 $\forall \sigma, \sigma'.\ \sigma' \in P \wedge \sigma \preceq \sigma' \Longrightarrow \sigma \in P$.*

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 $S_\star$, it preserves $I(\rho)$ with respect to $S_\star$.*

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* $\mathsf{Consent}$ events than I, then any $\langle S|\eta|\delta_{\mathsf{Consent}}\rangle$ satisfies $I^{-1}(P_{\mathsf{Consent}})$. This is because the property P_{Consent} is downward-closed with respect to the relation $\preceq_{\mathsf{Consent}+}$ such that $\sigma \preceq_{\mathsf{Consent}+} \sigma'$ iff σ is obtained from σ' by adding $\mathsf{Consent}$ 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 $s_\eta^0 \xrightarrow{\boldsymbol{\sigma} \triangleright \boldsymbol{\rho}} s'_\eta$, then $\circ\boldsymbol{\sigma}$ is a prefix of some σ' with $\sigma' \preceq I(\circ\boldsymbol{\rho})$.*

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 $\langle \mathcal{S}|\eta|\delta\rangle$ complies with $I^{-1}(P)$.*

Proof. Analogously to Theorem 3 ($\Longrightarrow$), we prove that $\langle \mathcal{S}|\eta|\delta\rangle$ complies with $I^{-1}(P)$ using our downward-closure assumption. Let ρ be a trace of $\langle \mathcal{S}|\eta|\delta\rangle$. Since $\langle \mathcal{S}|\eta|\delta\rangle$ complies with $I^{-1}(P)$, then $\rho \in I^{-1}(P)$, i.e., $\sigma \triangleq I(\rho) \in P$. By our first assumption, $I^\star(\rho) \preceq I(\rho)$. By our second assumption applied to σ and $\sigma' \triangleq I(\rho)$, we get $I^\star(\rho) \in P$, i.e., $\rho \in {I^\star}^{-1}(P)$. Hence, $\langle \mathcal{S}|\eta|\delta\rangle$ complies with ${I^\star}^{-1}(P)$.

Theorem 9. *Let η be a PEP over $(\mathcal{A}, \mathcal{E})$ and δ an edit automaton over $\mathcal{E}$. Assume that $\langle \mathcal{S}|\eta|\delta\rangle$ preserves $I^{-1}(P)$ with respect to $\mathcal{S}$. If $\forall \rho.\ I(\rho) \preceq I^\star(\rho)$ and P is downward-closed under $\preceq$, then $\langle \mathcal{S}|\eta|\delta\rangle$ preserves ${I^\star}^{-1}(P)$ with respect to $\mathcal{S}$.*

Proof. Let ${I^\star}^{-1}(P)$ be a trace of $\langle \mathcal{S}|\eta|\delta\rangle$. By our first assumption, $I(\rho) \preceq I^\star(\rho)$. By our second assumption, $I(\rho) \in P$, i.e., $\rho \in I^{-1}(P)$. Since $\langle \mathcal{S}|\eta|\delta\rangle$ preserves $I^{-1}(P)$ with respect to $\mathcal{S}$, then ρ is the trace of an execution of $\mathcal{S}$. Hence, $\langle \mathcal{S}|\eta|\delta\rangle$ preserves ${I^\star}^{-1}(P)$ with respect to $\mathcal{S}$.

5 Practical Enforcement with Batches and Capabilities

The general model that we have described in Sections 3–4 must be instantiated for specific classes of events, PDPs, and downward-closed relations to be used with real-world tools. In this section, we specialize our theory to enforcement mechanisms that work with sets of simultaneous events ('batches', sometimes also called *timepoints* [5] in the literature) and distinguish between events that the PDP can cause or suppress [13]. This will allow us to implement our framework with EnfGuard [17] as PDP in Section 6. We first define batch edit automata, batch PEPs, and their properties, and introduce a relation $\preceq_{\mathsf{mono}}$ whose downward closure contains properties that cannot be violated by generating 'more' or 'fewer' of certain events (Section 5.1). Then, we give a concrete batch PEP algorithm and prove sufficient ($\preceq$-)soundness conditions (Section 5.2).

5.1 Batch Edit Automata and Batch PEPs

In this section, we consider the case where each $a \in \mathcal{A}$ and $e \in \mathcal{E}$ is a finite set, i.e., $\mathcal{A} \triangleq \mathcal{P}(A)_f$ and $\mathcal{E} \triangleq \mathcal{P}(E)_f$, where $\mathcal{P}(\cdot)_f$ denotes the set of all finite subsets. In contrast to the previous sections, we now refer to elements of $\mathcal{A}$ and $\mathcal{E}$ as *action batches* and *event batches*, and reserve the words *actions* and *events* for elements of A and E, respectively. As in previous work [17], we assume that the edit automaton serving as a PDP cannot insert or delete event batches, but only edit the *content* of the sets, and that which events can be caused or suppressed is known in advance: there is a set $C \subseteq E$ of *causable* events and a set $S \subseteq E$ of *suppressable* events. We call such automata *batch automata*; the pair (C, S) is called the automaton's *capabilities*.

We first specialize our definitions of edit automata and PEPs:

Definition 18. *A* batch edit automaton *(BEA) with capabilities* (C, S) *is an EA with transitions* $s_\delta \xrightarrow{\ell \triangleright \langle \ell' \rangle} s'_\delta$ *or* $s_\delta \xrightarrow{\ell \triangleright \tau} s'_\delta$ *only, where* $\ell' - \ell \subseteq C$ *and* $\ell - \ell' \subseteq S$.

Definition 19. *A* batch PEP *(BPEP) with capabilities* (C, S) *is an input-enabled PEP with transitions* $s_\eta \xrightarrow{a \triangleright \langle e \rangle}_{\eta,\mathsf{i}} s'_\eta$ *and* $s'_\eta \xrightarrow{e \triangleright \langle a \rangle}_{\eta,\mathsf{i}} s''_\eta$ *where, whenever* $s^0_\eta \xrightarrow{\boldsymbol{\sigma} \triangleright \boldsymbol{\rho}}$ $s_\eta \xrightarrow{a' \triangleright \langle e' \rangle}_{\eta,\mathsf{i}} s'_\eta \xrightarrow{e \triangleright \langle a \rangle}_{\eta,\mathsf{e}} s''_\eta$, *there exists* $e'' \in \mathcal{E}$ *and* s'''_η *such that* $s_\eta \xrightarrow{a \triangleright \langle e'' \rangle} s'''_\eta$, $e'' - e' \subseteq C$, *and* $e' - e'' \subseteq S$.

Next, we define classes of properties that are common in practice, namely those where adding or removing a given event e to any batch of a compliant trace can never yield a non-compliant trace. When a property P can never be violated by just adding e events to an already compliant trace, we call it *e-monotonic*; when it can never be violated by just removing e events from such a trace, we call it *e-antimonotonic*.

Definition 20. *A property* $P \subseteq \mathcal{E}^*$ *is* monotonic *with respect to* $e \in E$ *iff for all* $\sigma, \sigma' \in \mathcal{E}^*$ *such that for all* $i \in \mathbb{N}$, $\sigma'_i \in \{\sigma_i, \sigma_i \cup \{e\}\}$, $\sigma \in P \Longrightarrow \sigma' \in P$.

Similarly, a property $P \subseteq \mathcal{E}^*$ *is* antimonotonic *with respect to* $e \in E$ *iff for all* $\sigma, \sigma' \in \mathcal{E}^*$ *such that for all* $i \in \mathbb{N}$, $\sigma'_i \in \{\sigma_i, \sigma_i - \{e\}\}$, $\sigma \in P \Longrightarrow \sigma' \in P$.

For the rest of this section, we fix two sets $M^+, M^- \subseteq E$ of events and consider the following relation $\preceq_{\mathsf{mono}}$:

$$\sigma \preceq_{\mathsf{mono}} \sigma' \iff |\sigma| = |\sigma'| \land \forall i \in \{1, \dots, |\sigma|\}.\ \sigma'_i - \sigma_i \subseteq M^+ \land \sigma_i - \sigma'_i \subseteq M^-.$$

The following proposition directly follows from Definition 20:

Theorem 10. *A property P is downward-closed with respect to $\preceq_{\mathsf{mono}}$ iff it is monotonic in every $e \in M^+$ and antimonotonic in every $e \in M^-$.*

5.2 A Batch PEP Algorithm

In this subsection, we present a concrete batch PEP algorithm and give sufficient conditions for it to be (i) sound and transparent or at least (ii) $\preceq_{\mathsf{mono}}$-sound.

The algorithm. Our algorithm is defined via two functions $\mathsf{i}_\eta : \mathcal{A}^* \times \mathcal{A} \to \mathcal{E}$ and $\mathsf{e}_\eta : \mathcal{A}^* \times \mathcal{A} \times \mathcal{E} \to \mathcal{A}$ such that $s^0_\eta \xrightarrow{\boldsymbol{\sigma} \triangleright \boldsymbol{\rho}} s_\eta \xrightarrow{a' \triangleright \langle e' \rangle}_{\eta,\mathsf{i}} s'_\eta$ iff $e' = \mathsf{i}_\eta(\circ\boldsymbol{\rho}, a')$ and $s^0_\eta \xrightarrow{\boldsymbol{\sigma} \triangleright \boldsymbol{\rho}} s_\eta \xrightarrow{a' \triangleright \langle e' \rangle}_{\eta,\mathsf{i}} s'_\eta \xrightarrow{e \triangleright \langle a \rangle}_{\eta,\mathsf{e}} s''_\eta$ iff $a = \mathsf{e}_\eta(\circ\boldsymbol{\rho}, a', e)$. We call the resulting PEP $\mathcal{B}(\mathsf{i}_\eta, \mathsf{e}_\eta)$. The function i_η, called the *instrumentation mapping*, can be freely chosen. It maps an action to a sequence of events, which may depend on the history of past actions. The function e_η, called the *enforcement mapping*, is defined in terms of the instrumentation mapping as well as three *handlers*: the causation handler h_C, the preservation handler h_K ('K' stands for 'keep'), and the suppression handler h_S. Each handler is a partial map describing how the enforcer should handle the PDP's edits: when the original set of actions is a' and the PDP causes some set of events e, the handler generates the set of actions $\mathsf{h}_C(e, a')$; when the PDP does not modify some events e, the handler generates the actions $\mathsf{h}_K(e, a')$; and when it suppresses some events e, the handler generates the actions $\mathsf{h}_S(e, a')$. Since the first argument of each handler is a set of events, there may be several ways to use the same handler to cause, preserve, or suppress the same events. For instance, to cause $\{e_1, e_2, e_3\}$, one can either call $\mathsf{h}_C(\{e_1\}, a')$, $\mathsf{h}_C(\{e_2\}, a')$, and $\mathsf{h}_C(\{e_3\}, a')$ consecutively and collect the generated actions, call $\mathsf{h}_C(\{e_1, e_2\}, a')$ and $\mathsf{h}_C(\{e_2, e_3\}, a')$, call only $\mathsf{h}_C(\{e_1, e_2, e_3\})$, etc. For the PEP to be input-enabled, the domain of h_C (respectively, h_K, h_S) must *generate* C (respectively, E, S):

Definition 21. *Let Ω be an element set. A set $X \in \mathcal{P}(\mathcal{P}(\Omega))$ is a generator of another set $Y \in \mathcal{P}(\mathcal{P}(\Omega))$ iff for any $y \in Y$, there exists a finite subset $X' \subseteq X$ such that $y = \bigcup_{x \in X'} x$.*

In the case where several decompositions of the set of events into a union of elements of the domain exists, we can use any function $\mathsf{choose}_{Y;X} : Y \to \mathcal{P}(X)_f$ such that $\bigcup \mathsf{choose}_{X;Y}(y) = y$ to select a valid decomposition. Given such a choice function, the handlers provide a convenient way for developers to define

Algorithm 1 Batch PEP Algorithm

```
$\mathsf{e}_\eta(\rho, a', e) =$ let $e' = \mathsf{i}_\eta(\rho, a')$ in
    (if $e' = e$ then $a'$ else let $c_1, \ldots, c_{n_C} = \mathsf{choose}_{\mathrm{dom}(\mathsf{h}_C);C}(e - e')$;
                                    $k_1, \ldots, k_{n_K} = \mathsf{choose}_{\mathrm{dom}(\mathsf{h}_K);K}(e \cap e')$;
                                    $s_1, \ldots, s_{n_S} = \mathsf{choose}_{\mathrm{dom}(\mathsf{h}_S);S}(e' - e)$
                                in $\bigcup_{i=1}^{n_C} \mathsf{h}_C(c_i, a') \cup \bigcup_{i=1}^{n_K} \mathsf{h}_K(k_i, a') \cup \bigcup_{i=1}^{n_S} \mathsf{h}_S(s_i, a')$ )
```

(sound) PEP reactions to arbitrary PDP edits, without having to explicitly define the PEP's reaction to every combination of edits.

Algorithm 1 gives the pseudocode of e_η. First, the enforcer computes the events e' corresponding to the original actions and compares them to the edited events e returned by the PDP. If the PDP did not modify the events, the enforcer returns the original actions a'. Otherwise, it decomposes the sets of caused actions $e' - e$, preserved actions $e \cap e'$, and suppressed actions $e - e'$ over $\mathrm{dom}(\mathsf{h}_C)$, $\mathrm{dom}(\mathsf{h}_K)$, and $\mathrm{dom}(\mathsf{h}_S)$ respectively, and calls the corresponding handlers, returning the union of all generated actions.

Correctness. The following theorem provides a set of conditions that together guarantee that $\mathcal{B}(\mathsf{i}_\eta, \mathsf{e}_\eta)$ is a sound and transparent PEP with capabilities (C, S). We first state the theorem and then give an intuitive interpretation of each of the conditions in terms of instrumentation auditing:

Theorem 11. *Let $\mathsf{i}_\eta : \mathcal{A}^* \times \mathcal{A} \to \mathcal{E}$, $\mathsf{h}_C : \mathcal{P}(C)_f \rightharpoonup \mathcal{A} \to \mathcal{A}$, $\mathsf{h}_S : \mathcal{P}(S)_f \rightharpoonup \mathcal{A} \to \mathcal{A}$, $\mathsf{h}_K : \mathcal{P}(E)_f \rightharpoonup \mathcal{A} \to \mathcal{A}$ be given. Suppose that (1) $\mathrm{dom}(\mathsf{h}_C)$ is a generator of $\mathcal{P}(C)_f$ under $\cup$, (2) $\mathrm{dom}(\mathsf{h}_K)$ is a generator of $\mathcal{P}(E)_f$ under $\cup$, (3) $\mathrm{dom}(\mathsf{h}_S)$ is a generator of $\mathcal{P}(S)_f$ under $\cup$, (4) $\forall \rho, a, b.\ \mathsf{i}_\eta(\rho, a) \cup \mathsf{i}_\eta(\rho, b) = \mathsf{i}_\eta(\rho, a \cup b)$, where we set $\tau \cup s \triangleq s \cup \tau \triangleq s$, (5) $\forall \rho, c, a.\ \mathsf{i}_\eta(\rho, \mathsf{h}_C(c, a)) = c$, (6) $\forall \rho, k, a.\ \mathsf{i}_\eta(\rho, \mathsf{h}_K(k, a)) = k$, (7) $\forall \rho, s, a.\ \mathsf{i}_\eta(\rho, \mathsf{h}_S(s, a)) = \emptyset$, and (8) $\forall \rho, a.\ \mathsf{i}_\eta(\rho, a) = (I(\rho \cdot \langle a \rangle))_{|\rho|+1}$. Then $\mathcal{B}(\mathsf{i}_\eta, \mathsf{e}_\eta)$ is a batch PEP with capabilities (C, S) that is sound and transparent with respect to I.*

Proof. Since (i) $\mathrm{dom}(h_C)$ is a generator of $\mathcal{P}(E)_f$ under $\cup$ and (ii) $\mathrm{dom}(h_S)$ is a generator of $\mathcal{P}(S)_f$, then the function e_η is total. Transparency is straightforward. For soundness, we compute

$$\begin{aligned}
&\mathsf{i}_\eta(a_C \cup a_K \cup a_S) \\
&= \mathsf{i}_\eta\left(\bigcup_{i=1}^{n_C} \mathsf{h}_C(c_i, a') \cup \bigcup_{i=1}^{n_K} \mathsf{h}_K(k_i, a') \cup \bigcup_{i=1}^{n_S} \mathsf{h}_S(s_i, a')\right) \\
&= \left(\bigcup_{i=1}^{n_C} \mathsf{i}_\eta(\mathsf{h}_C(c_i, a'))\right) \cup \left(\bigcup_{i=1}^{n_K} \mathsf{i}_\eta(\mathsf{h}_K(k_i, a'))\right) \cup \left(\bigcup_{i=1}^{n_S} \mathsf{i}_\eta(\mathsf{h}_S(s_i, a'))\right) && \text{by (iii)} \\
&= \left(\bigcup_{i=1}^{n_C} c_i\right) \cup \left(\bigcup_{i=1}^{n_K} k_i\right) \cup \left(\bigcup_{i=1}^{n_S} \emptyset\right) && \text{by (iv)–(v)}
\end{aligned}$$

$= e - e' \cup e \cap e' = e.$ by def. e_η

This theorem provides a checklist (1)–(8) that can be used to audit the correctness of the system's instrumentation when our PEP algorithm is used together with a sound PDP. The checks to be performed are as follows:

(1–3) Is the causation handler (respectively, preservation, suppression handler) defined for all causable events (respectively, for all events, for all suppressable events)?

(4) Does the instrumentation mapping map single actions to events independently, i.e., does a set of n actions map to the same events as the union of the events that each of the n actions maps to?

(5-6) When the causation (respectively, preservation) handler receives events, does it generate actions that map exactly (through i_η) to the events it received?

(7) Does the suppression handler only generate actions that map to $\emptyset$?

(8) Does the instrumentation mapping i_η implement exactly I?

For all events $e \in \mathcal{E}$ such that $e \in \mathsf{i}_\eta(\rho, \{x\}) \Longrightarrow \mathsf{i}_\eta(\rho, \{x\}) = \{e\}$ (i.e., events that are never generated together with another event), note that the part of the preservation handler related to e can be straightforwardly defined as $a' \mapsto h_K(\{e\}, a') = \{x \in a' \mid e \in \mathsf{i}_\eta(\rho, \{x\})\}$, triggering all actions in a' that map to e.

In practice, auditing these requirements, especially (6) and (8), can be challenging, as it requires checking correct instrumentation and enforcement for *all possible events*: the instrumentation must *always* provide *exactly* the right events *whenever necessary*; the enforcer must *always* generate *exactly* the right actions.

For $\preceq_{\mathsf{mono}}$ soundness, we can significantly weaken conditions (5)–(8):

Theorem 12. *Let i_η, h_C, h_S, and h_K be given. Suppose that (1)–(4) and (7) are as in Theorem 11, (5) $\forall\, c, a.\ \mathsf{i}_\eta(\rho, \mathsf{h}_C(c, a)) - c \subseteq C \cap M^+ \wedge c - \mathsf{i}_\eta(\rho, \mathsf{h}_C(c, a)) \subseteq M^-$, (6) $\forall \rho, k, a.\ \mathsf{i}_\eta(\rho, \mathsf{h}_K(k, a)) - k \subseteq C \cap M^+ \wedge k - \mathsf{i}_\eta(\rho, \mathsf{h}_K(k, a)) \subseteq S \cap M^-$, (8) $\forall \rho, a.\ \mathsf{i}_\eta(\rho, a) \preceq (I(\rho \cdot \langle a \rangle))_{|\rho|+1}$. Then $\mathcal{B}(\mathsf{i}_\eta, \mathsf{e}_\eta)$ is $\preceq_{\mathsf{mono}}$-sound with respect to I.*

Proof. By Theorems 8 and 11.

The questions to be answered for (5), (6), and (8) are now the following:

(5-6) a. When a handler receives an event that is *not both* antimonotonic *and* suppressable, does it always generate an action that maps to this event?
b. When a handler generates an action that maps to an event that is *not both* monotonic *and* causable, did it always receive this event originally?

(8) a. Does the implementation mapping i_η ensure that every event that is *not* monotonic is always logged when an action mapping to it occurs?
b. Does the implementation mapping i_η ensure that every event that is *not* antimonotonic is only ever logged when an action mapping to it occurs?

6 Case Study

InstrLib. We have implemented our batch PEP algorithm (Section 5) in a Python library called InstrLib [19]. The library consists of 1,500 lines of Python

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

Table 1: The trace mediator I in Minitwitter

code with two instrumentation layers: a low-level layer, which allows for instrumenting arbitrary Python function calls and attributes, and a high-level layer providing off-the-shelf enforcement hooks for Django web applications. The latter uses a fixed set of actions (`read`, `write`, `input`, `output`, `execute`) to capture database 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 the appendix. It has bindings to the state-of-the-art EnfGuard tool [17].

Minitwitter. We now showcase the usage of InstrLib by instrumenting a microblogging 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 $\mathsf{Use}(u,p)$, denoting "user u's data is used for purpose p;" $\mathsf{Consent}(u,p)$ (respectively, $\mathsf{Revoke}(u,p)$), denoting "user u gives (respecively, revokes) consent to use their data for purpose p;" $\mathsf{Request}(u)$, denoting "user u requests deletion of their data;" and $\mathsf{Delete}(u)$, 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.

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

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

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

	Baseline ($\propto n$)					Instrumented ($\propto n$)				
View	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

Table 2: Runtime latency (ms) over 20 repetitions

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 `write`s 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 Section 5.2, which are sufficient when i_η maps each action to at most one event. All enforcement-related code is showed in the appendix.

In Table 2, we report the latency of four of Minitwitter's views with and without instrumentation with InstrLib: viewing the **timeline**, **post**ing a message, giving **consent**, and **request**ing 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.

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 12. 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 `input`s 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. Luca Aceto, Ian Cassar, Adrian Francalanza, and Anna Ingólfsdóttir. On Runtime Enforcement via Suppressions. In Sven Schewe and Lijun Zhang, editors, *Conference on Concurrency Theory (CONCUR 2018)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 34:1–34:17, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
2. Luca Aceto, Ian Cassar, Adrian Francalanza, and Anna Ingolfsdottir. Bidirectional runtime enforcement of first-order branching-time properties. *Logical Methods in Computer Science*, 19, 2023.
3. Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.
4. Guangdong Bai, Liang Gu, Tao Feng, Yao Guo, and Xiangqun Chen. Context-aware usage control for Android. In *Security and Privacy in Communication Networks (SecureComm)*, volume 6, pages 326–343. Springer, 2010.
5. David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. Monitoring metric first-order temporal properties. *Journal of the ACM (JACM)*, 62(2):15:1–15:45, 2015.
6. Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. Developing theoretical foundations for runtime enforcement. *CoRR*, abs/1804.08917, 2018.

7. Hadil Charafeddine, Khalil El-Harake, Yliès Falcone, and Mohamad Jaber. Runtime enforcement for component-based systems. In *Symposium on applied computing*, pages 1789–1796. ACM, 2015.
8. Jan Chomicki. Efficient checking of temporal integrity constraints using bounded history encoding. *Transactions on Database Systems (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. Úlfar Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, 2004.
11. Ulfar Erlingsson and Fred B Schneider. Sasi enforcement of security policies: A retrospective. In *New security paradigms*, pages 87–95, 1999.
12. Yliès Falcone and Mohamad Jaber. Fully automated runtime enforcement of component-based systems with formal and sound recovery. *Journal on Software Tools for Technology Transfer (STTT)*, 19(3):341–365, 2017.
13. François Hublet, David Basin, and Srđan Krstić. Real-time policy enforcement with metric first-order temporal logic. In *European Symposium on Research in Computer Security (ESORICS)*, pages 211–232. Springer, 2022.
14. François Hublet, David Basin, and Srđan Krstić. Enforcing the GDPR. In *European Symposium on Research in Computer Security (ESORICS)*, pages 400–422. Springer, 2023.
15. François Hublet, David Basin, and Srđan Krstić. 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. François Hublet, Leonardo Lima, David Basin, Srđan Krstić, and Dmitriy Traytel. Scaling up proactive enforcement. In Ruzica Piskac and Zvonimir Rakamarić, editors, *Computer Aided Verification (CAV)*, pages 370–392. Springer, 2025.
18. François Hublet, Leonardo Lima, David Basin, Srđan Krstić, and Dmitriy Traytel. Proactive real-time first-order enforcement. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification (CAV)*, volume 14682 of *LNCS*, pages 156–181. Springer, 2024.
19. François Hublet, David Basin, Linda Hu, Srđan Krstić, and Lennard Reese. InstrLib, 2025. `https://github.com/runtime-enforcement/instrlib`.
20. Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *Journal of Information Security*, 4:2–16, 2005.
21. Jay Ligatti and Srikar Reddy. A theory of runtime enforcement, with results. In *European Symposium on Research in Computer Security (ESORICS)*, volume 15, pages 87–100. Springer, 2010.
22. Nancy A. Lynch and Mark R. Tuttle. *An introduction to input/output automata*. Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.
23. Srinivas Pinisetty, Ylies Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet, and Omer Landry Nguena Timo. Runtime enforcement of timed properties. In *Runtime Verification (RV)*, pages 229–244. Springer, 2012.
24. Siegfried Rasthofer, Steven Arzt, Enrico Lovat, and Eric Bodden. Droidforce: Enforcing complex, data-centric, system-wide policies in android. In *Availability, Reliability and Security*, pages 40–49. IEEE, 2014.
25. Samira Sarraf. Optus breach occurred due to a coding error, alleges ACMA. *CSO Online*, 2022.
26. Fred Schneider. Enforceable security policies. *Trans. Inf. Syst. Sec.*, 3(1):30–50, 2000.

A Enforcement code for Minitwitter

```python
...

from instrlib.django.orm import InstrumentORM

from .enforcer import logger

def info_user(user):
  try:
    return str(user)
  except:
    return ""

@InstrumentORM(logger,
             {"User.first_name", "User.last_name", "User.is_staff", "User.
                 email", "User.save", "User.last_login"},
             info = info_user, events = {'read', 'write', 'execute'})
class User(AbstractUser):

  def delete_data(self):
    self.twit.all().delete()
    self.follower.all().delete()
    self.following.all().delete()

def info_follow(follow):
  try:
    return str(object.__getattribute__(follow, 'follower'))
  except:
    return ""

@InstrumentORM(logger,
               {"Follow.follower", "Follow.following", "Follow.date", "Follow
                   "},
               info = info_follow, events = {'read', 'write'})
class Follow(CommonInfo):
  ...

def info_twit(twit):
  try:
    return str(object.__getattribute__(twit, 'author'))
  except:
    return ""

@InstrumentORM(logger,
               {"Twit.content", "Twit.posted_on", "Twit.updated_on", "Twit.
                   author", "Twit.save"},
               info = info_twit, events = {'read', 'write'})
class Twit(CommonInfo):
  ...
```

models.py

```python
...

from instrlib.django.url import InstrumentURL
from twitt.enforcer import logger

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('twitt.urls')),
]

to_instrument = {
    "twitt.views.SetCookieConsentView",
    "twitt.views.DeleteAllView",
}

urlpatterns = InstrumentURL(logger, to_instrument, events = {'input'})(
    urlpatterns)
```

urls.py

```python
class HomeView(LoginRequiredMixin, TemplateView):
  template_name = 'index.html'

  @with_purpose('marketing')
  def generate_advertisement(self):
    ...
```

views.py

```python
from typing import Any

from instrlib.event import Event
from instrlib.pdp import EnfGuard
from instrlib.logger import Logger
from instrlib.pep import InstrumentationMapping, PEP
from instrlib.schema import Schema

from Twitter.settings import INSTRLIB_EXE, INSTRLIB_FORMULA, INSTRLIB_LOG,
    INSTRLIB_SIG

# Handlers

def none_handler(event_name, event_args, response, *args, **kwargs):
    return None

def delete_handler(events):
    from twitt.models import User

    for event in events:

        user_name = event['args'][0]
        try:
            user = User.objects.get(username=user_name)
        except User.DoesNotExist:
            return

        user.delete_data()

# Schema

schema = Schema()
schema.add('Use', [str, str])
schema.add('Consent', [str, str])
schema.add('Request', [str])
schema.add('Delete', [str])

# PDP

pdp = EnfGuard(INSTRLIB_EXE, INSTRLIB_SIG, INSTRLIB_FORMULA, log_file =
    INSTRLIB_LOG)

# PEP

suppression_handlers : dict[str | tuple[str, ...], Any] = {
    ('Use')    : none_handler,
}
causation_handlers : dict[str | tuple[str, ...], Any] = {
    ('Delete') : delete_handler,
}
```

```

def read_mapping(action):
    return Event('Use', action.args[4], action.args[5])

def write_mapping(action):
    return Event('Use', action.args[5], action.args[6])

def input_mapping(action):
    if action.args[:3] == ('SetCookieConsentView', 'accept', 'true'):
        return Event('Consent', action.args[3], 'marketing')
    elif action.args[:3] == ('DeleteAllView', 'delete', 'true'):
        return Event('Request', action.args[3])
    else:
        return None

def execute_mapping(action):
    if action.args[:2] == ('User', 'delete_data'):
        return Event('Delete', action.args[3])
    else:
        return None

instrumentation_mapping = InstrumentationMapping({
    'read':    read_mapping,
    'write':   write_mapping,
    'input':   input_mapping,
    'execute': execute_mapping,
})

pep = PEP(
    suppression_handlers    = suppression_handlers,
    causation_handlers      = causation_handlers,
    instrumentation_mapping = instrumentation_mapping
)

# Logger

logger = Logger(pep, schema, pdp)
```

enforcer.py

B Architecture of Instrlib

The architecture of InstrLib, shown in Figure 4, involves six different kinds of parallel threads: worker threads; writer and reader threads; proactive worker threads; the PDP; and a timer thread. These threads interact according to two main flows. The first, *reactive* [18] flow (in green in Figure 4) starts with an instrumented function being run by a worker serving a user request; the events to be logged are computed and placed into a queue (1) that is read by the writer thread (2). The writer thread passes the events to the PDP (3) and places the original worker request in another queue (4). The reader thread reads the outputs of the PDP (5) and realigns them with the content of the second queue (6) before passing it back to the worker (7). Additionally, the reader thread can start *proactive* workers (8) to perform causation. The second, *proactive* [18] flow (in red) is initiated by a clock tick. It proceeds as before (2–6) but can only perform causation (8). This second flow is used when, e.g., an action has to be performed before a deadline irrespective of the presence of user inputs.

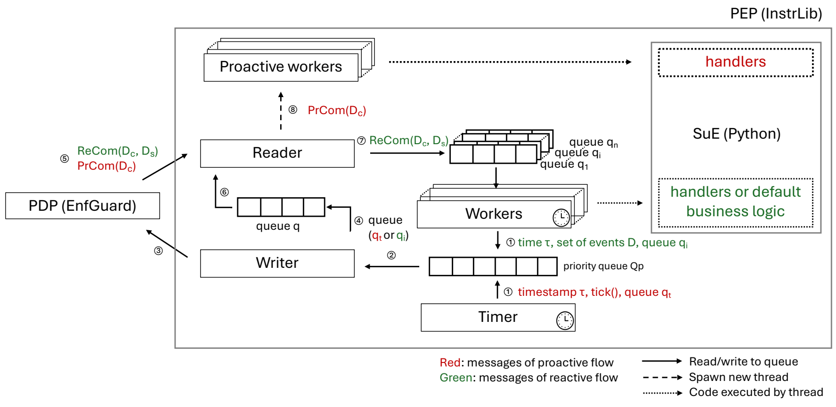


Fig. 4: Architecture of InstrLib