User-Controlled Privacy: Taint, Track, and Control

François Hublet
francois.hublet@inf.ethz.ch

David Basin
basin@inf.ethz.ch

Srđan Krstić
srdan.krtic@inf.ethz.ch

Privacy Enhancing Technologies Symposium — Bristol, July 16, 2024

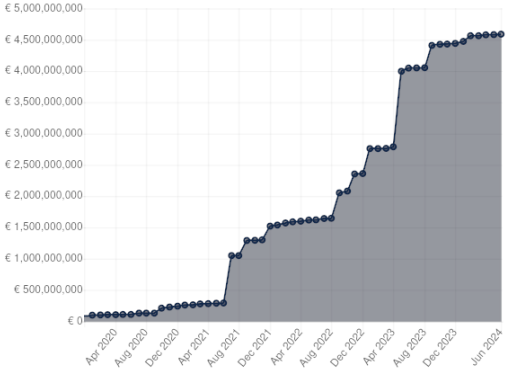
ETH zürich

Data protection: How things stand

Data protection: How things stand

Everything is fines...

Over €4.5B in GDPR fines since 2018

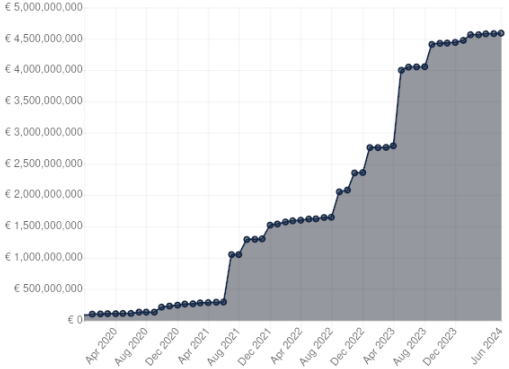


Source: `www.enforcementtracker.com/?insights`, retrieved 2024/07/10

Data protection: How things stand

Everything is fines...

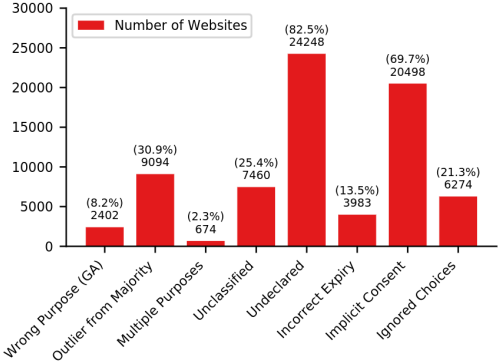
Over €4.5B in GDPR fines since 2018



Source: `www.enforcementtracker.com/?insights`, retrieved 2024/07/10

... but not everything is fine

A vast majority of websites are still non-compliant



Source: Bollinger et al. (2022)

Enforcing ex-ante rather than ex-post

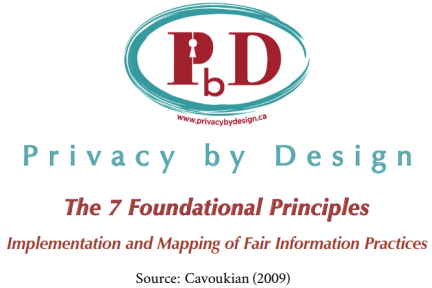
Enforcing ex-ante rather than ex-post

Privacy by Design...

Source: Cavoukian (2009)

Enforcing ex-ante rather than ex-post

Privacy by Design...



Source: Cavoukian (2009)

... is a legal obligation

Data protection by design and by default.

[T]he controller shall [...] implement appropriate technical and organisational measures [...] to meet the requirements of this Regulation.

— Art. 25(1) GDPR

Key privacy aspect: Consent- and purpose-based usage

Key privacy aspect: Consent- and purpose-based usage

Example: Data usage policy

Key privacy aspect: Consent- and purpose-based usage

Example: Data usage policy

Alice requires that, for any tweet she posts:

Key privacy aspect: Consent- and purpose-based usage

Example: Data usage policy

Alice requires that, for any tweet she posts:

(1) no data derived from that tweet is used for marketing purposes

Key privacy aspect: Consent- and purpose-based usage

Example: Data usage policy

Alice requires that, for any tweet she posts:

(1) no data derived from that tweet is used for marketing purposes

(2) for analytics purposes, such data can be sent only to `trustedanalytics.com`

Key privacy aspect: Consent- and purpose-based usage

Example: Data usage policy

Alice requires that, for any tweet she posts:

(1) no data derived from that tweet is used for marketing purposes

(2) for analytics purposes, such data can be sent only to `trustedanalytics.com`

(3) one week after she posts the tweet, data derived from it can only be used for service purposes

Key privacy aspect: Consent- and purpose-based usage

Example: Data usage policy

Alice requires that, for any tweet she posts:

(1) no data derived from that tweet is used for marketing purposes

(2) for analytics purposes, such data can be sent only to `trustedanalytics.com`

(3) one week after she posts the tweet, data derived from it can only be used for service purposes

We present the first approach supporting

Key privacy aspect: Consent- and purpose-based usage

Example: Data usage policy

Alice requires that, for any tweet she posts:

(1) no data derived from that tweet is used for marketing purposes

(2) for analytics purposes, such data can be sent only to `trustedanalytics.com`

(3) one week after she posts the tweet, data derived from it can only be used for service purposes

We present the first approach supporting

- Tracking of data in applications, through Information Flow Control (IFC)

Key privacy aspect: Consent- and purpose-based usage

Example: Data usage policy

Alice requires that, for any tweet she posts:

(1) no data derived from that tweet is used for marketing purposes

(2) for analytics purposes, such data can be sent only to `trustedanalytics.com`

(3) one week after she posts the tweet, data derived from it can only be used for service purposes

We present the first approach supporting

- Tracking of data in applications, through Information Flow Control (IFC) **and**
- Data usage policies that are

Key privacy aspect: Consent- and purpose-based usage

Example: Data usage policy

Alice requires that, for any tweet she posts:

(1) no data derived from that tweet is used for marketing purposes

(2) for analytics purposes, such data can be sent only to `trustedanalytics.com`

(3) one week after she posts the tweet, data derived from it can only be used for service purposes

We present the first approach supporting

- Tracking of data in applications, through Information Flow Control (IFC) **and**
- Data usage policies that are

R1. **User-specified**: users give and revoke consent

Key privacy aspect: Consent- and purpose-based usage

Example: Data usage policy

Alice requires that, for any tweet she posts:

(1) no data derived from that tweet is used for marketing purposes

(2) for analytics purposes, such data can be sent only to `trustedanalytics.com`

(3) one week after she posts the tweet, data derived from it can only be used for service purposes

We present the first approach supporting

- Tracking of data in applications, through Information Flow Control (IFC) **and**
- Data usage policies that are

R1. **User-specified**: users give and revoke consent

R2. **Per-input**: different inputs are subject to different restrictions

Key privacy aspect: Consent- and purpose-based usage

Example: Data usage policy

Alice requires that, for any tweet she posts:

(1) no data derived from that tweet is used for marketing purposes

(2) for analytics purposes, such data can be sent only to `trustedanalytics.com`

(3) one week after she posts the tweet, data derived from it can only be used for service purposes

We present the first approach supporting

- Tracking of data in applications, through Information Flow Control (IFC) **and**
- Data usage policies that are

R1. **User-specified**: users give and revoke consent

R2. **Per-input**: different inputs are subject to different restrictions

R3. **Time-dependent**: restrictions change over time

Key privacy aspect: Consent- and purpose-based usage

Example: Data usage policy

Alice requires that, for any tweet she posts:

(1) no data derived from that tweet is used for marketing purposes

(2) for analytics purposes, such data can be sent only to `trustedanalytics.com`

(3) one week after she posts the tweet, data derived from it can only be used for service purposes

We present the first approach supporting

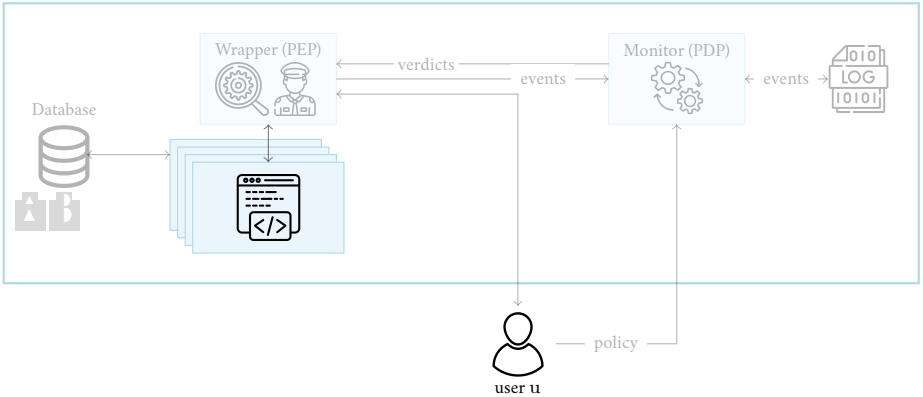
- Tracking of data in applications, through Information Flow Control (IFC) **and**
- Data usage policies that are

R1. **User-specified**: users give and revoke consent

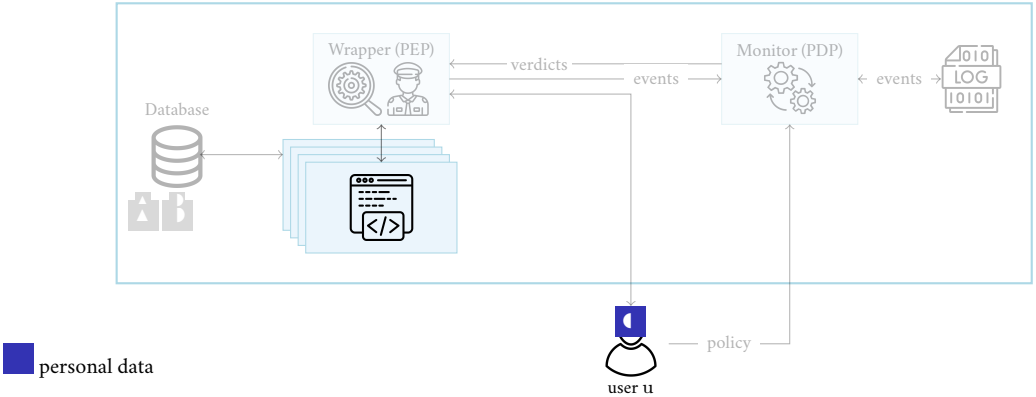
R2. **Per-input**: different inputs are subject to different restrictions

R3. **Time-dependent**: restrictions change over time

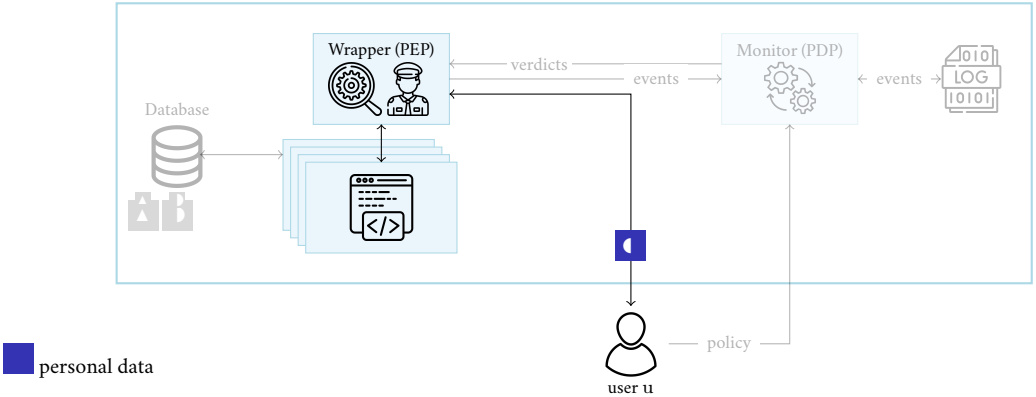
Taint, Track, and Control (TTC)



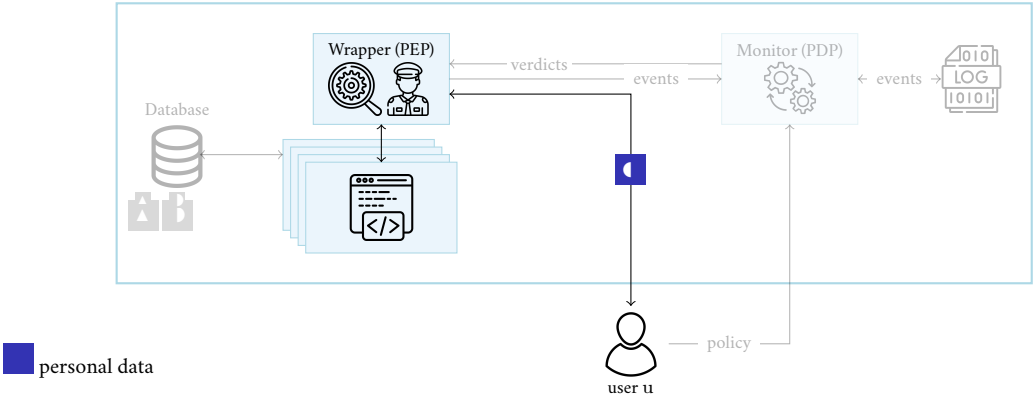
Taint, Track, and Control (TTC)



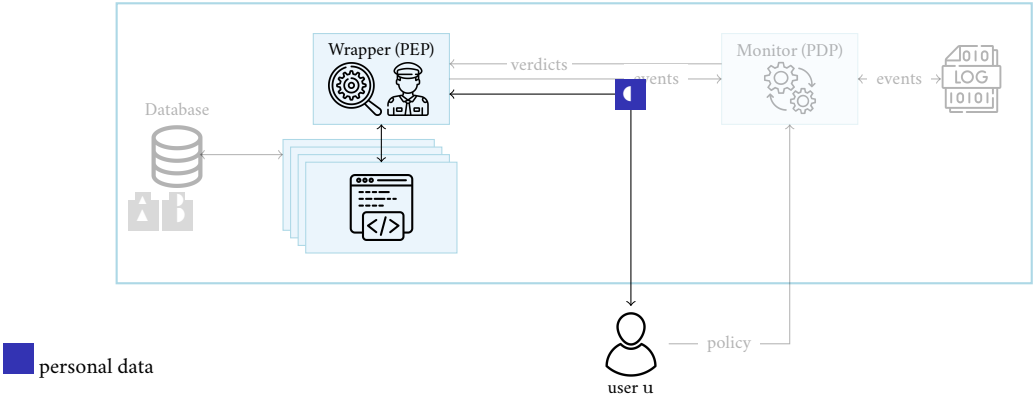
Taint, Track, and Control (TTC)



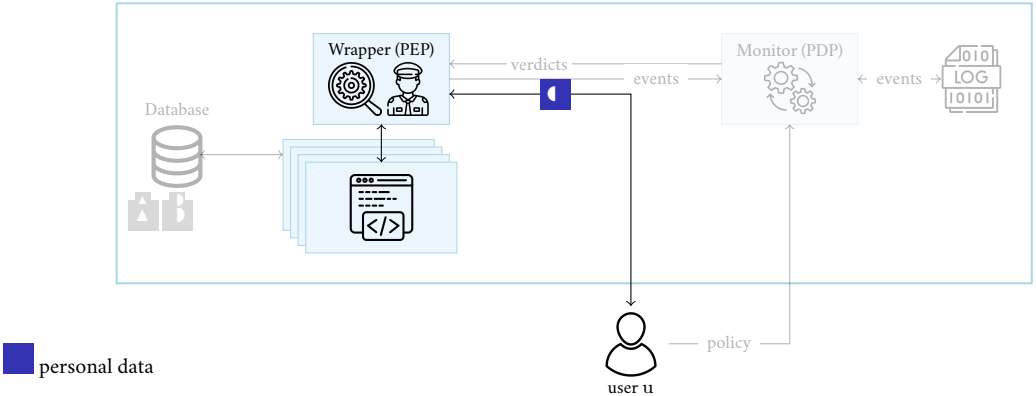
Taint, Track, and Control (TTC)



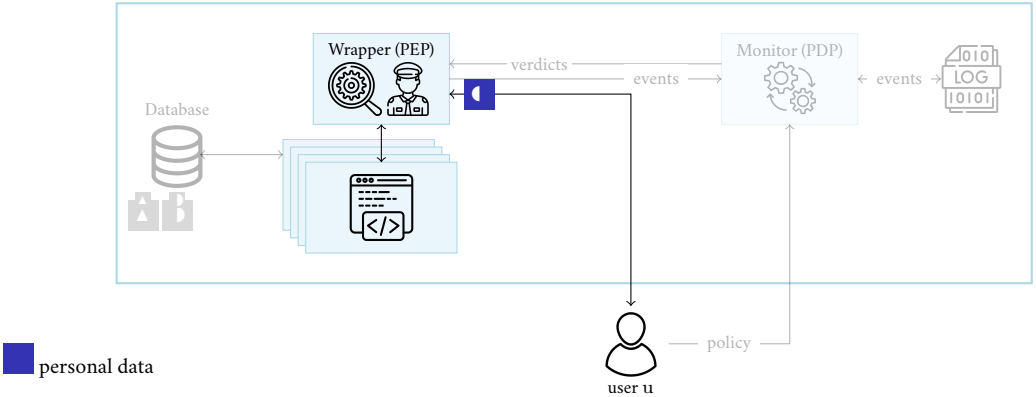
Taint, Track, and Control (TTC)



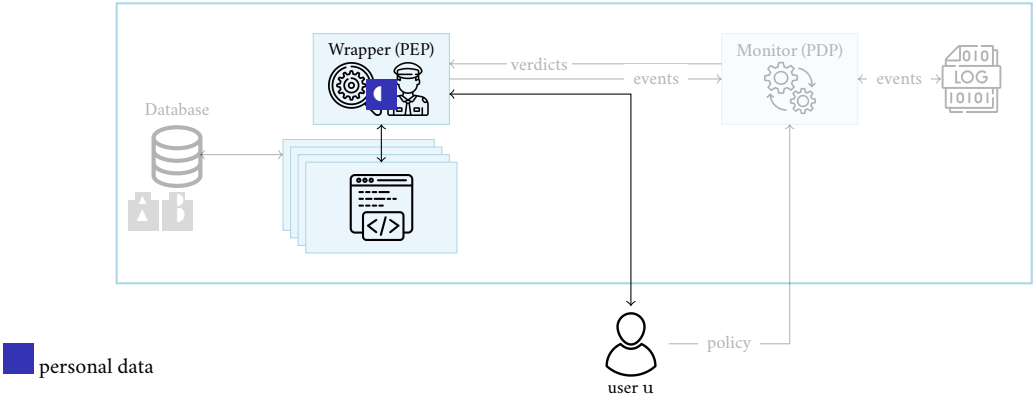
Taint, Track, and Control (TTC)



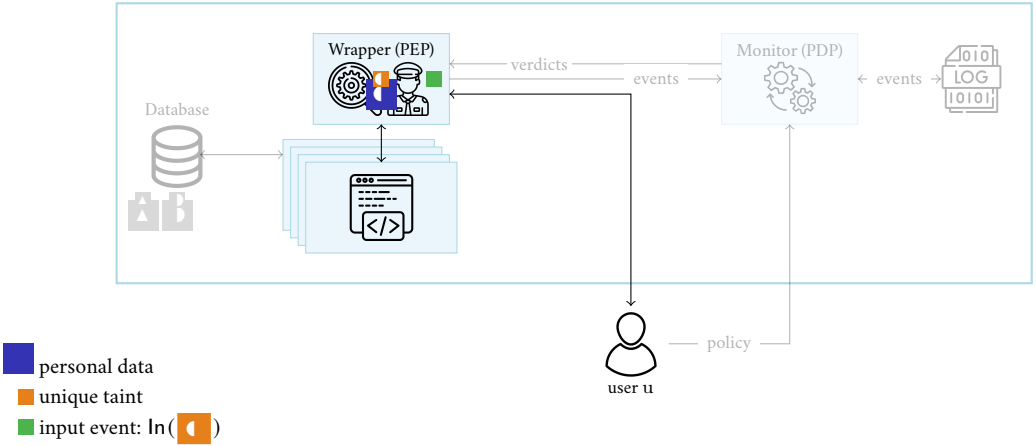
Taint, Track, and Control (TTC)



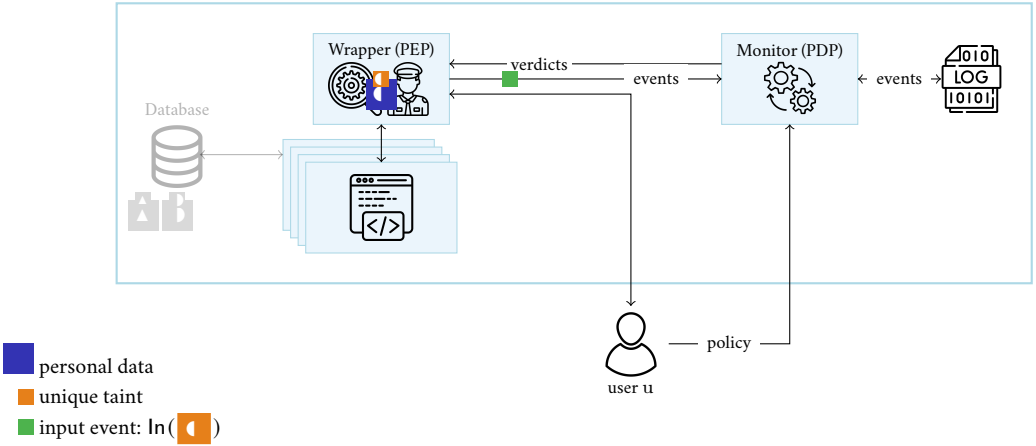
Taint, Track, and Control (TTC)



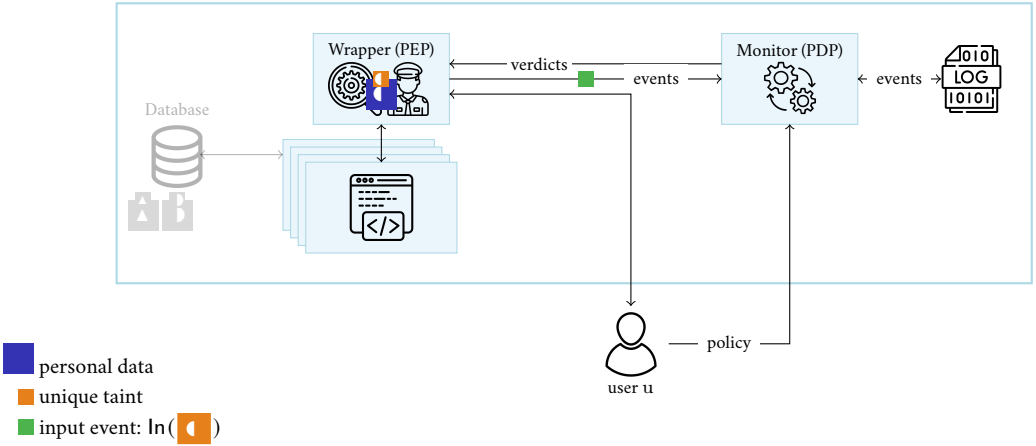
Taint, Track, and Control (TTC)



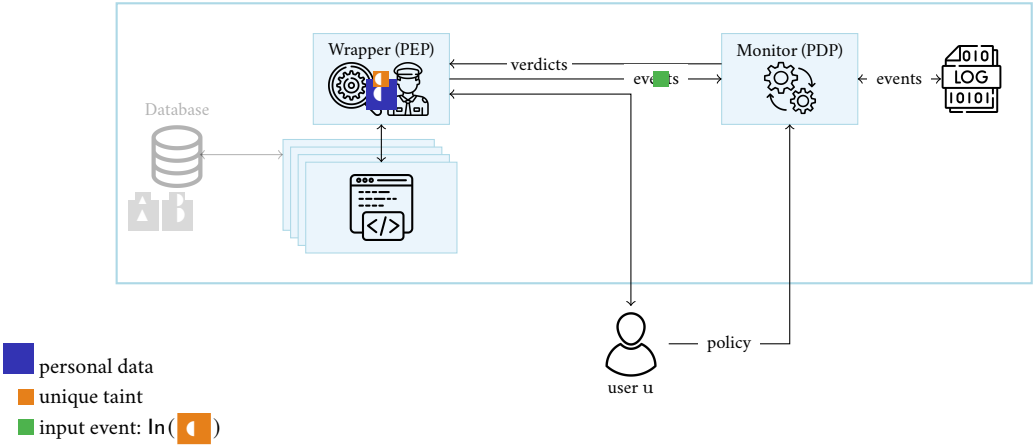
Taint, Track, and Control (TTC)



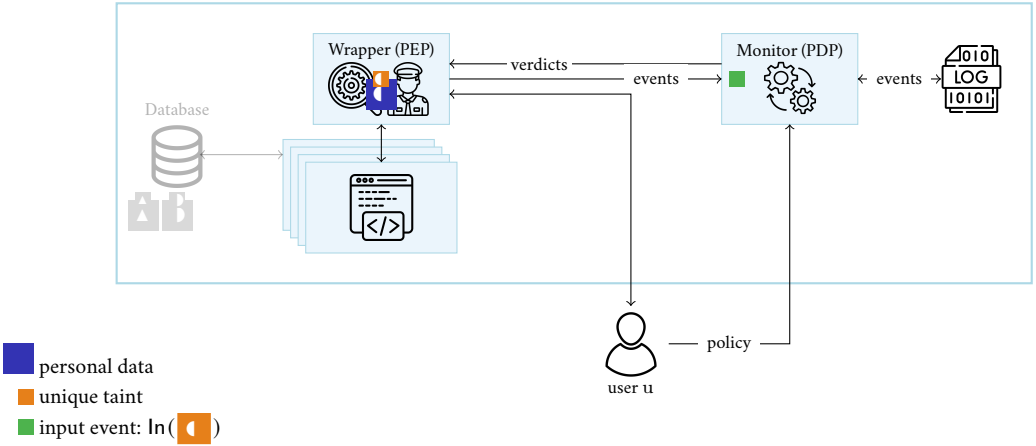
Taint, Track, and Control (TTC)



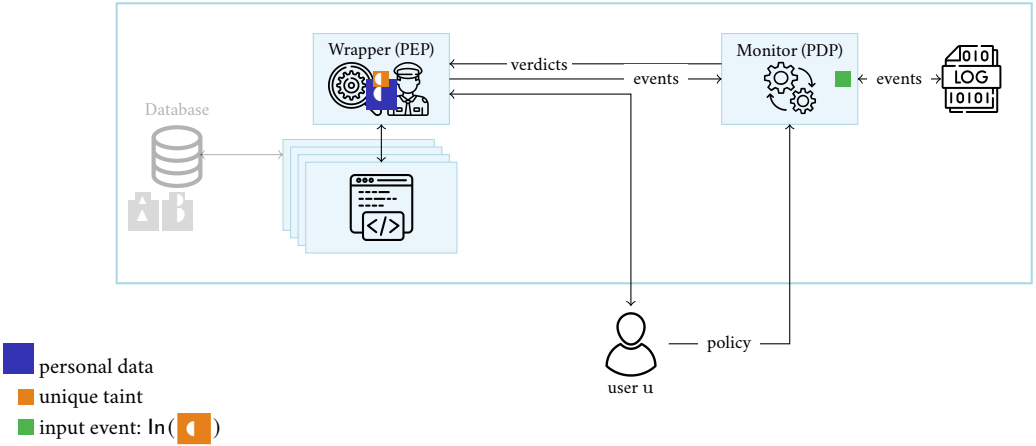
Taint, Track, and Control (TTC)



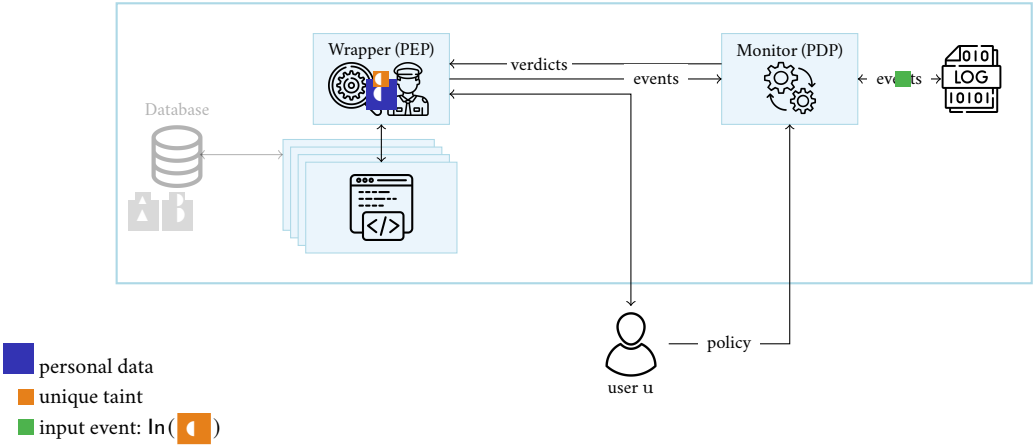
Taint, Track, and Control (TTC)



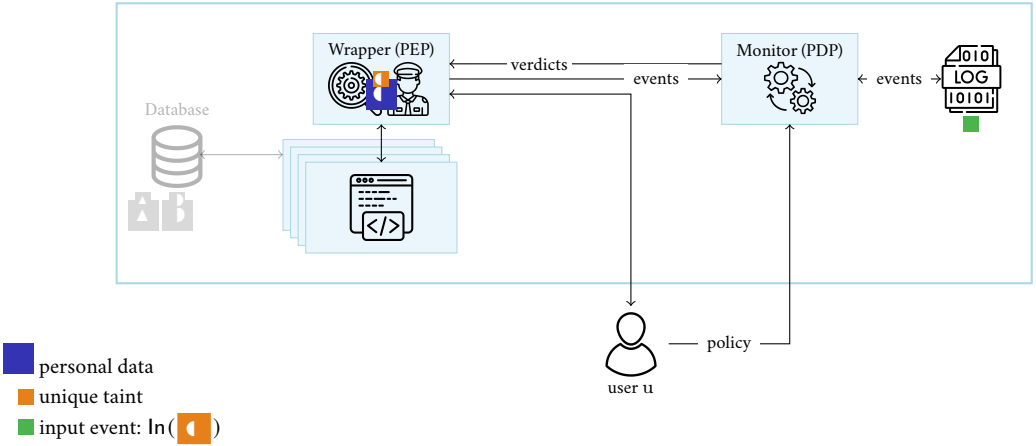
Taint, Track, and Control (TTC)



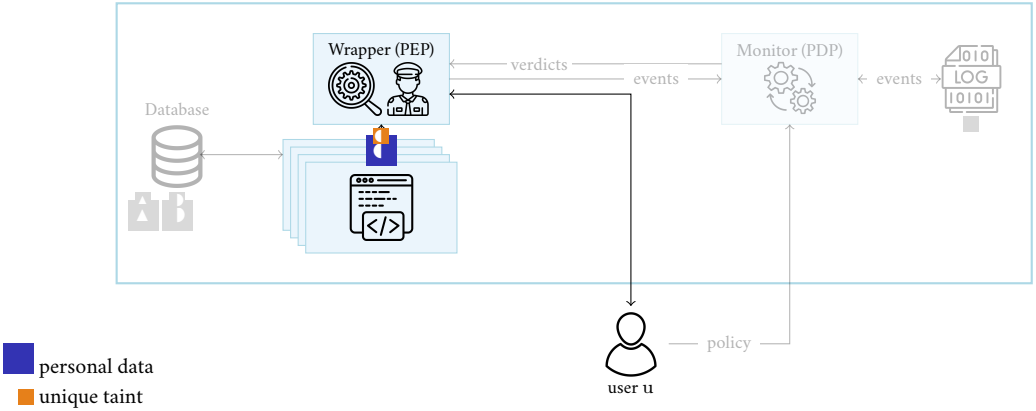
Taint, Track, and Control (TTC)



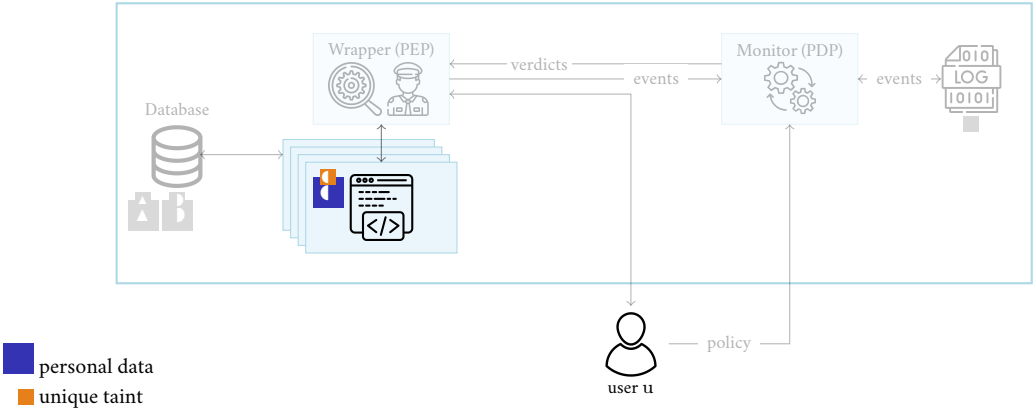
Taint, Track, and Control (TTC)



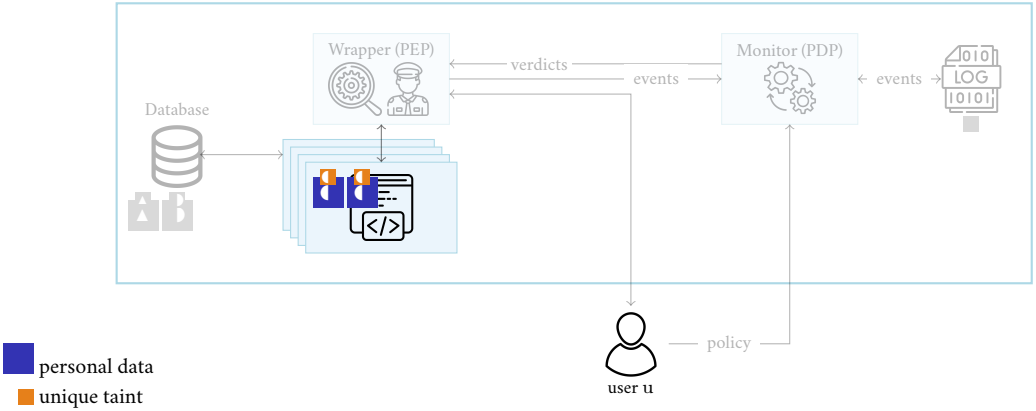
Taint, Track, and Control (TTC)



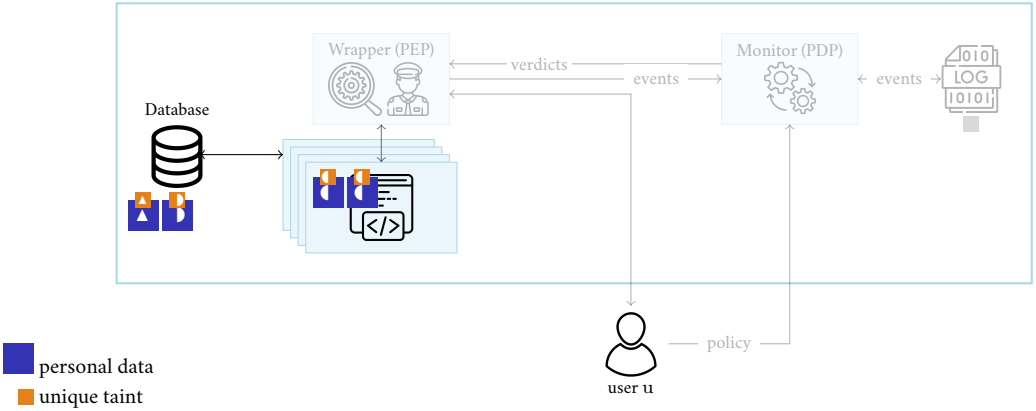
Taint, Track, and Control (TTC)



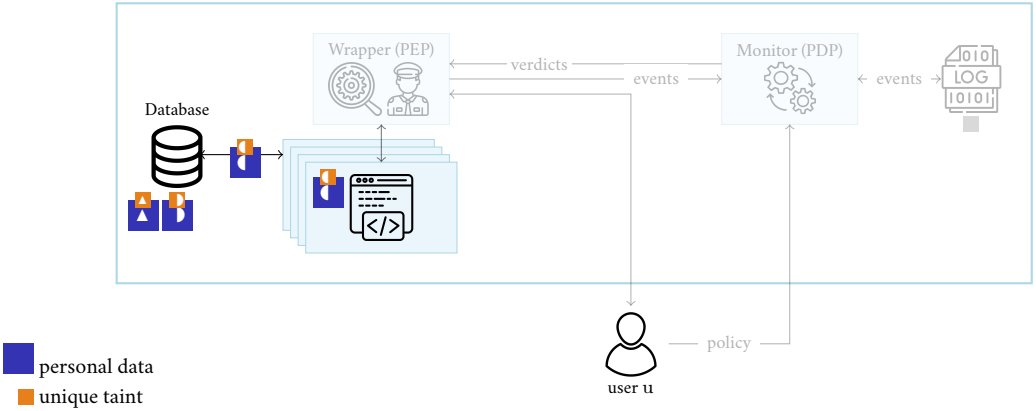
Taint, Track, and Control (TTC)



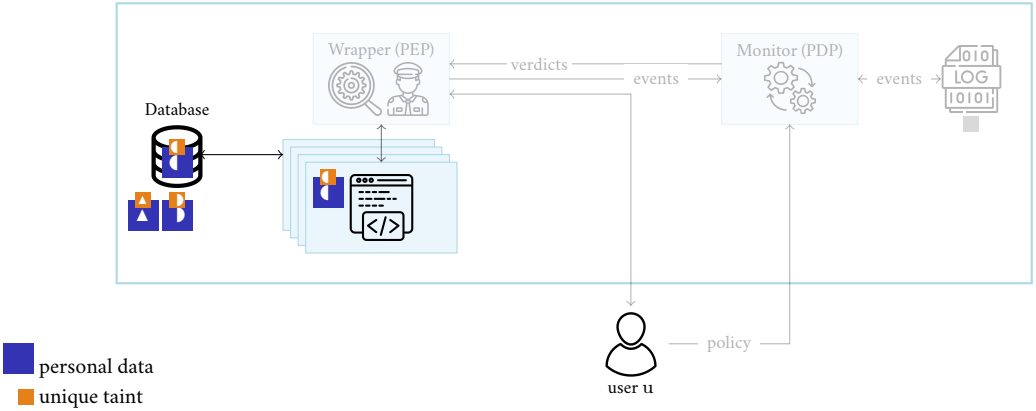
Taint, Track, and Control (TTC)



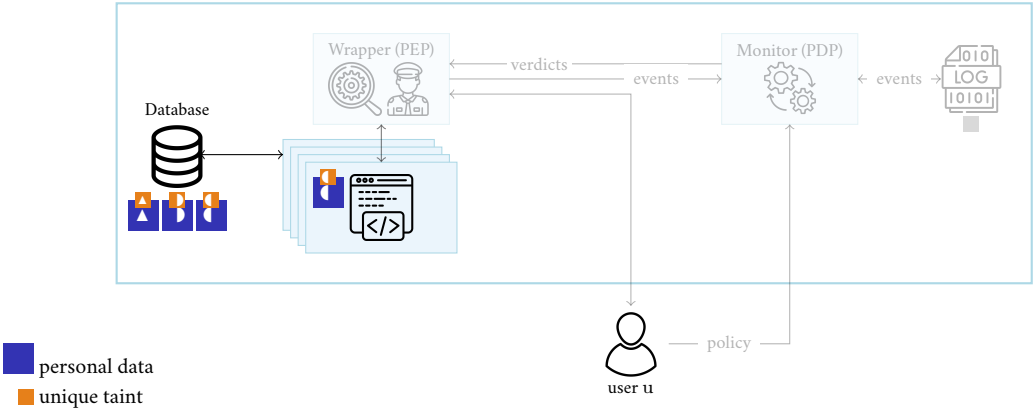
Taint, Track, and Control (TTC)



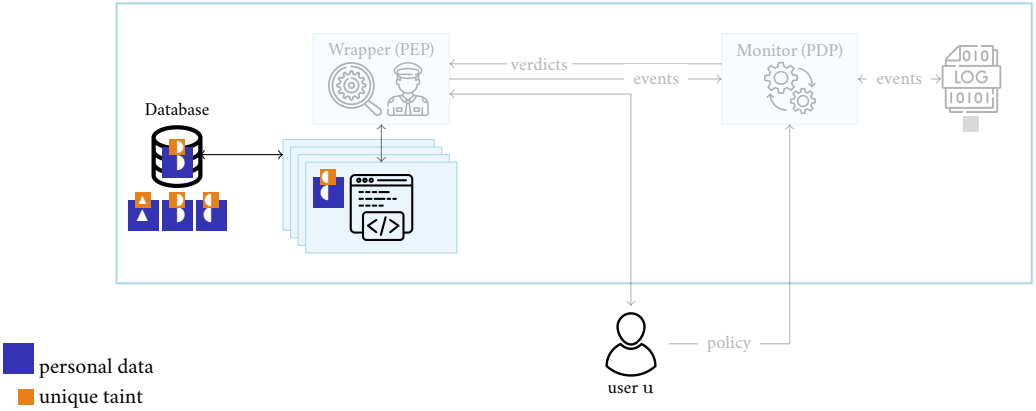
Taint, Track, and Control (TTC)



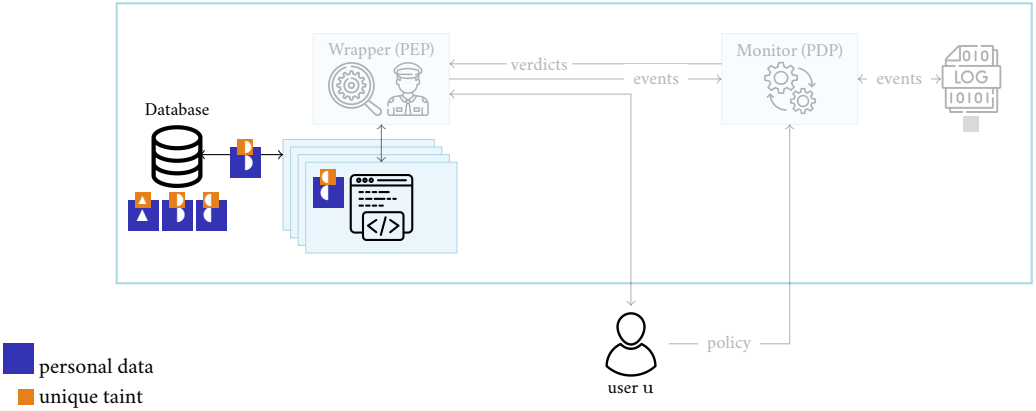
Taint, Track, and Control (TTC)



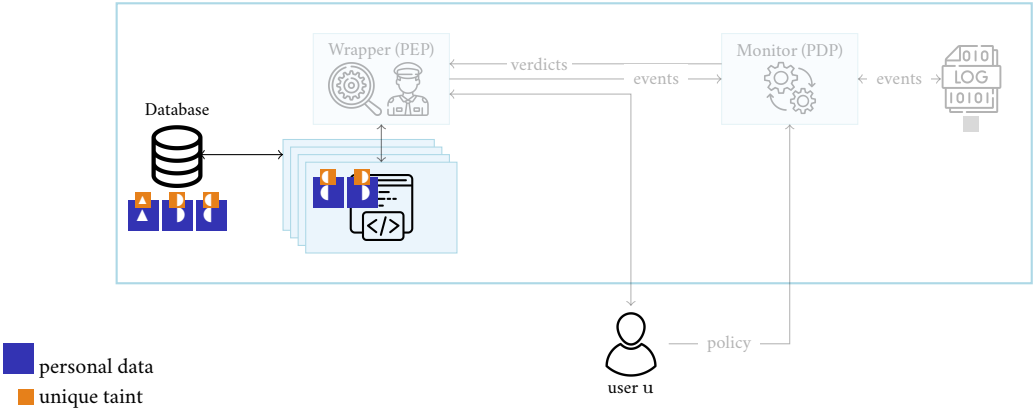
Taint, Track, and Control (TTC)



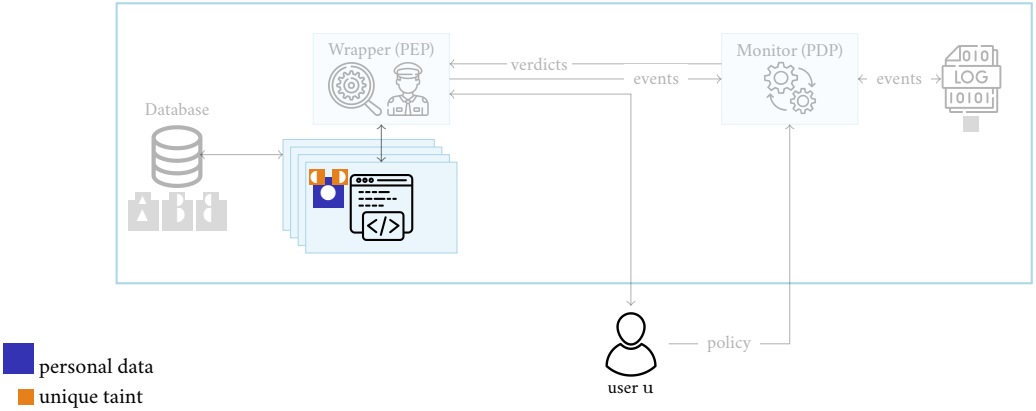
Taint, Track, and Control (TTC)



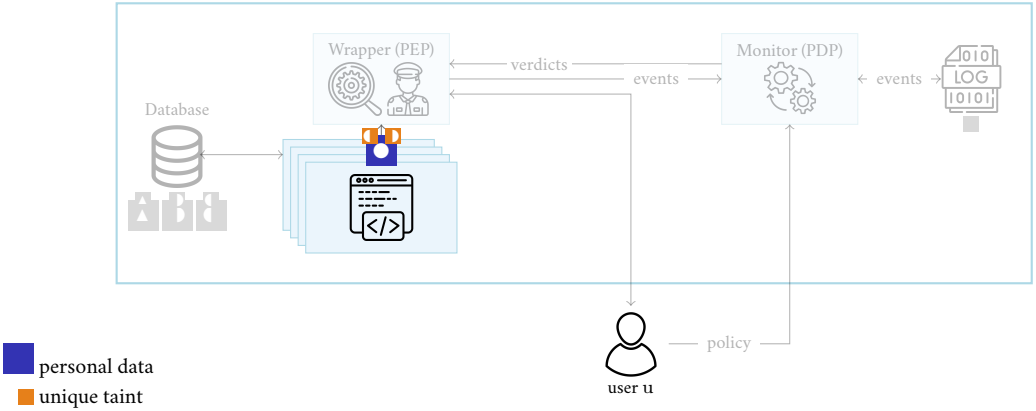
Taint, Track, and Control (TTC)



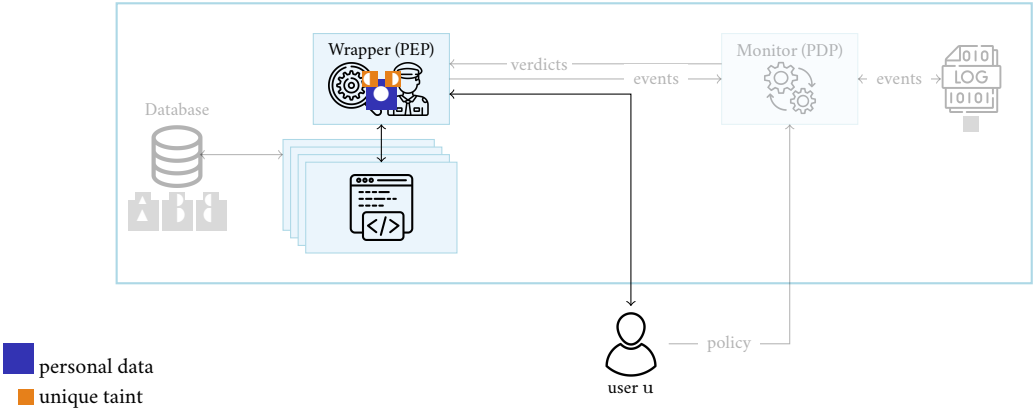
Taint, Track, and Control (TTC)



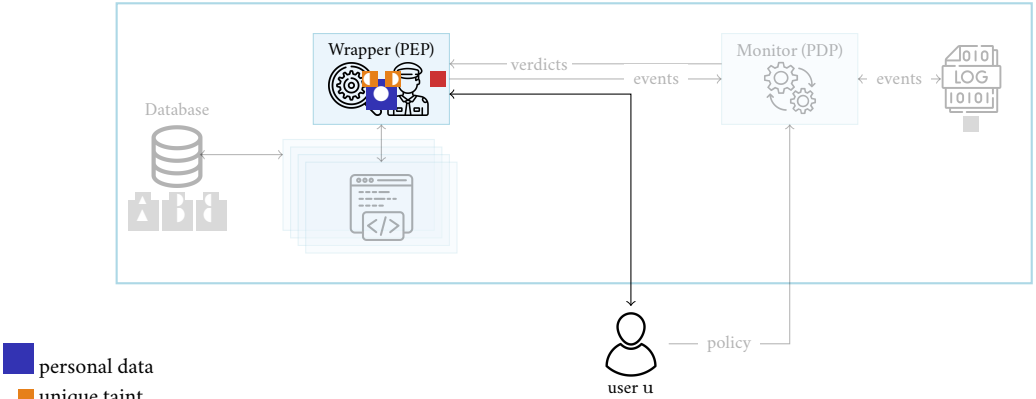
Taint, Track, and Control (TTC)



Taint, Track, and Control (TTC)



Taint, Track, and Control (TTC)

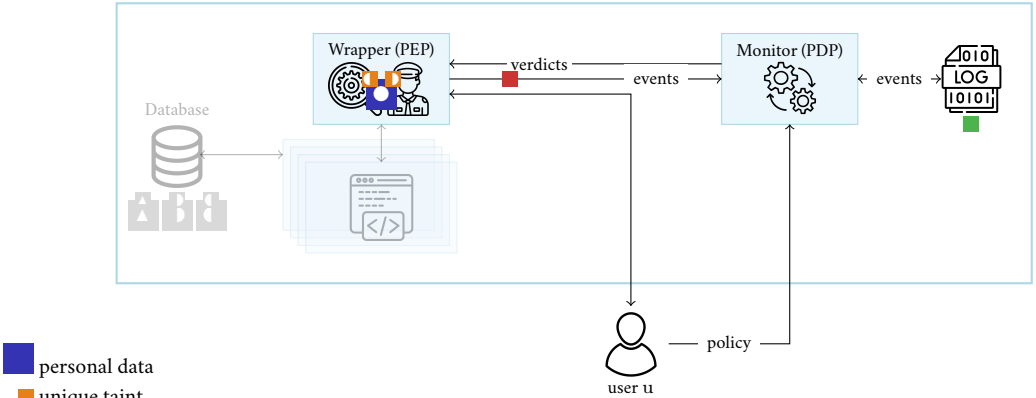


personal data

unique taint

output + interference events: Out(u, p,), Itf(,), Itf(,)

Taint, Track, and Control (TTC)



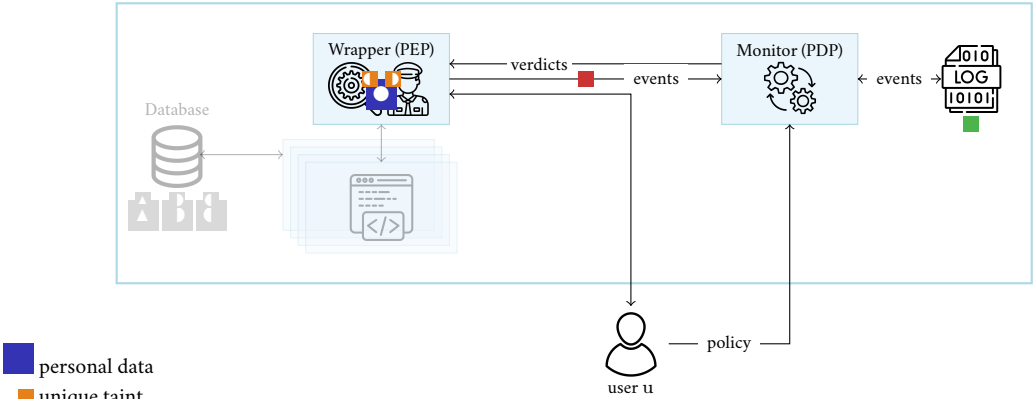
personal data

unique taint

input event: In( )

output + interference events: Out(u, p,), Itf(,), Itf(,)

Taint, Track, and Control (TTC)



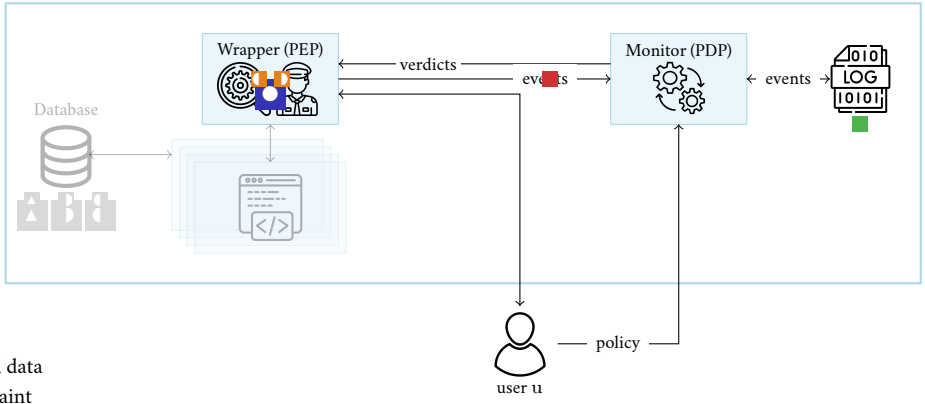
personal data

unique taint

input event: In( )

output + interference events: Out(u, p,), Itf(,), Itf(,)

Taint, Track, and Control (TTC)



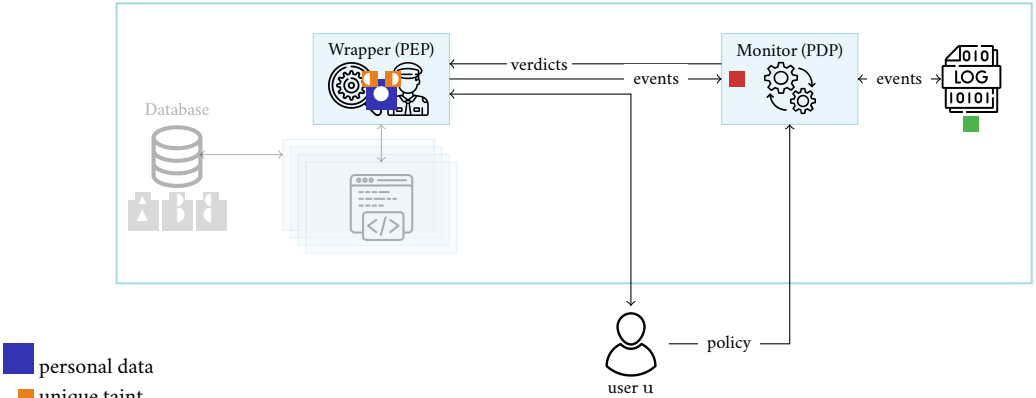
personal data

unique taint

input event: In( )

output + interference events: Out(u, p,), Itf(,), Itf(,)

Taint, Track, and Control (TTC)



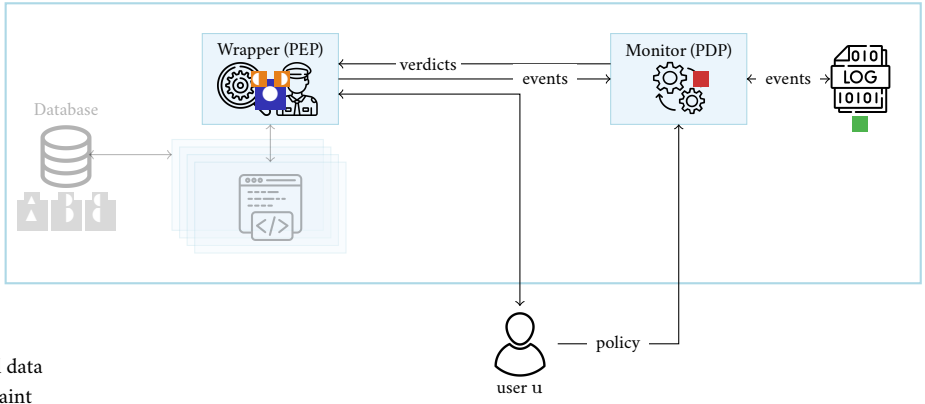
personal data

unique taint

input event: In( )

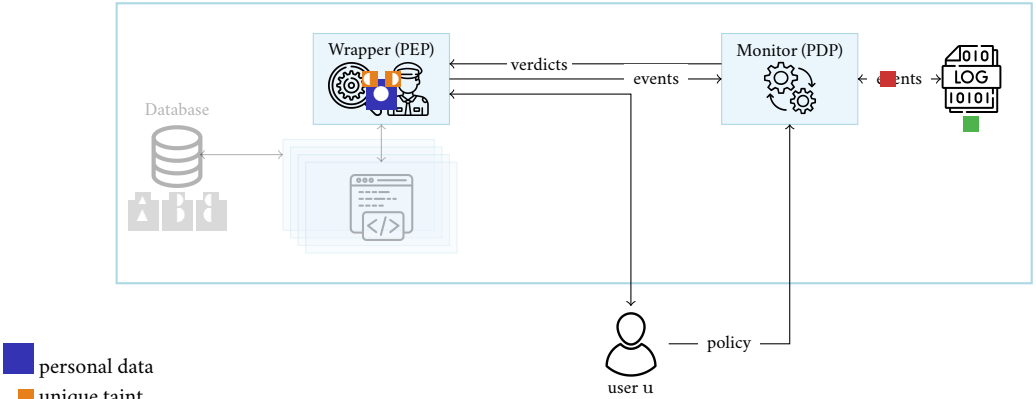
output + interference events: Out(u, p,), Itf(,), Itf(,)

Taint, Track, and Control (TTC)



- personal data
- unique taint
- input event: In( )
- output + interference events: Out(u, p,), Itf(,), Itf(,)

Taint, Track, and Control (TTC)



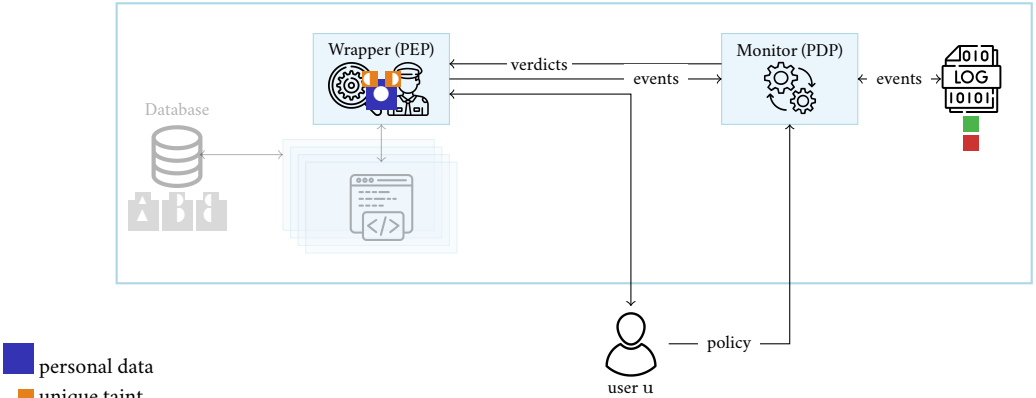
personal data

unique taint

input event: In( )

output + interference events: Out(u, p,), Itf(,), Itf(,)

Taint, Track, and Control (TTC)



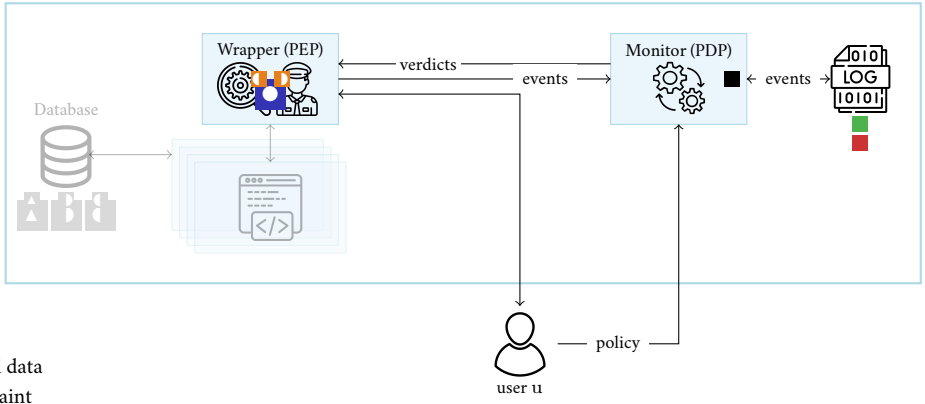
personal data

unique taint

input event: In( )

output + interference events: Out(u, p,), Itf(,), Itf(,)

Taint, Track, and Control (TTC)



personal data

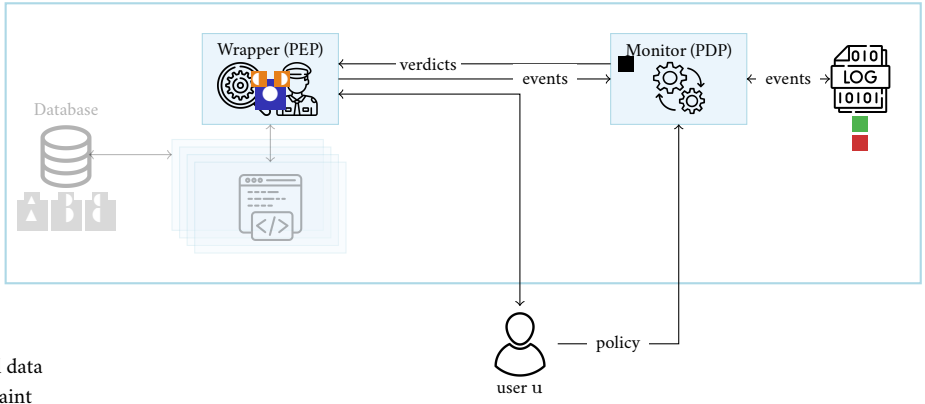
unique taint

input event: In( )

output + interference events: Out(u, p,), Itf(,), Itf(,)

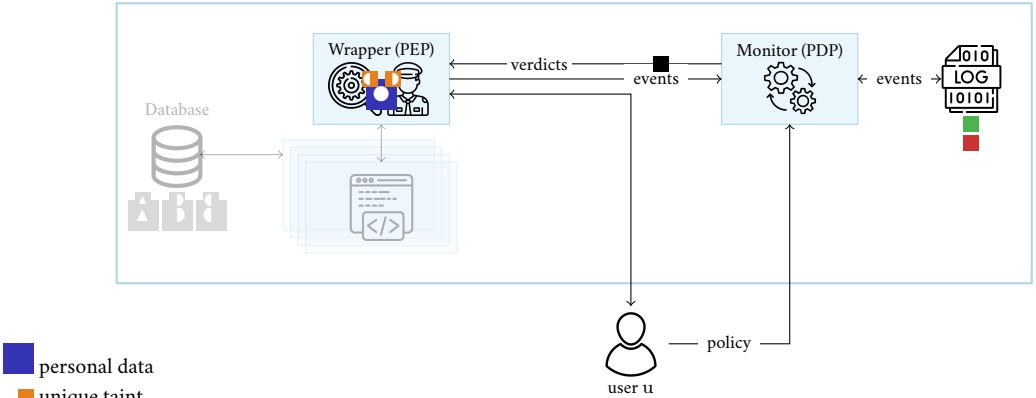
monitor verdict: ✓

Taint, Track, and Control (TTC)



- personal data
- unique taint
- input event: In( )
- output + interference events: Out(u, p,), Itf(,), Itf(,)
- monitor verdict: ✓

Taint, Track, and Control (TTC)



personal data

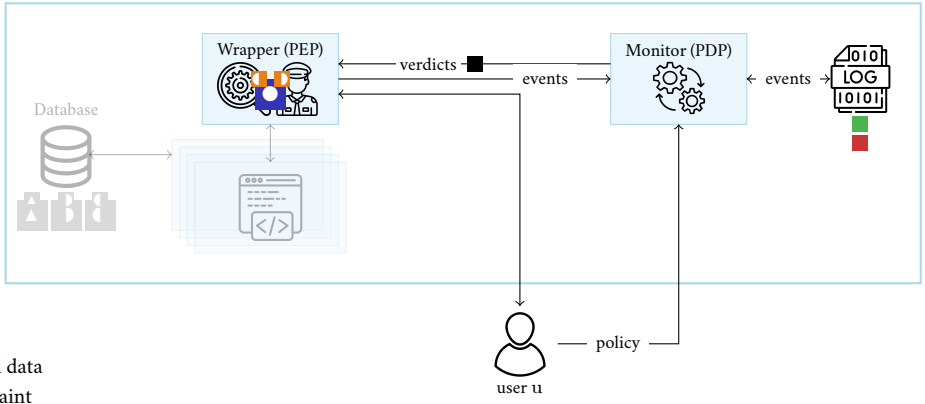
unique taint

input event: In( )

output + interference events: Out(u, p,), Itf(,), Itf(,)

monitor verdict: ✓

Taint, Track, and Control (TTC)



personal data

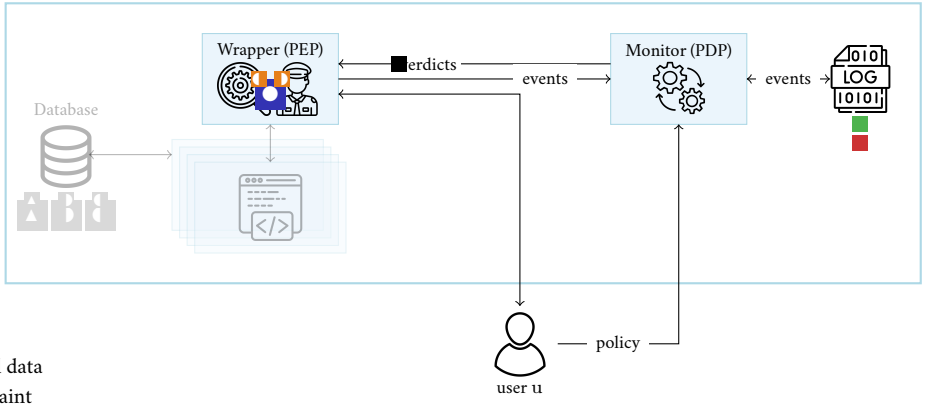
unique taint

input event: In( )

output + interference events: Out(u, p,), Itf(,), Itf(,)

monitor verdict: ✓

Taint, Track, and Control (TTC)



personal data

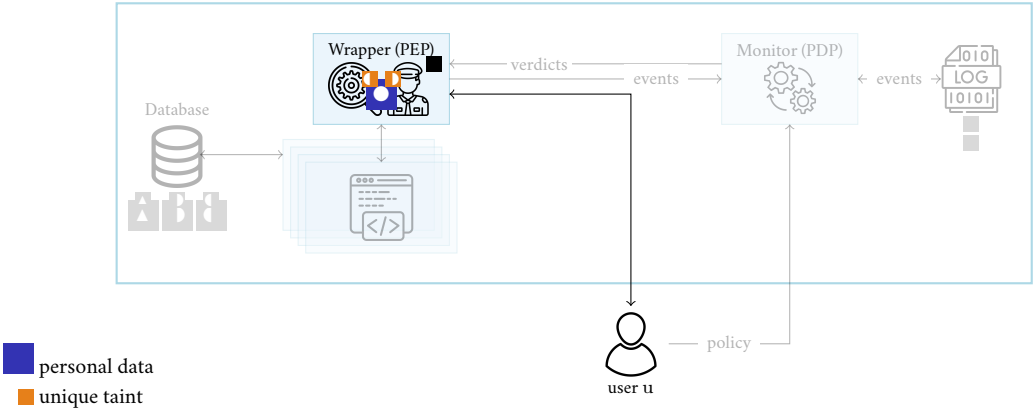
unique taint

input event: In( )

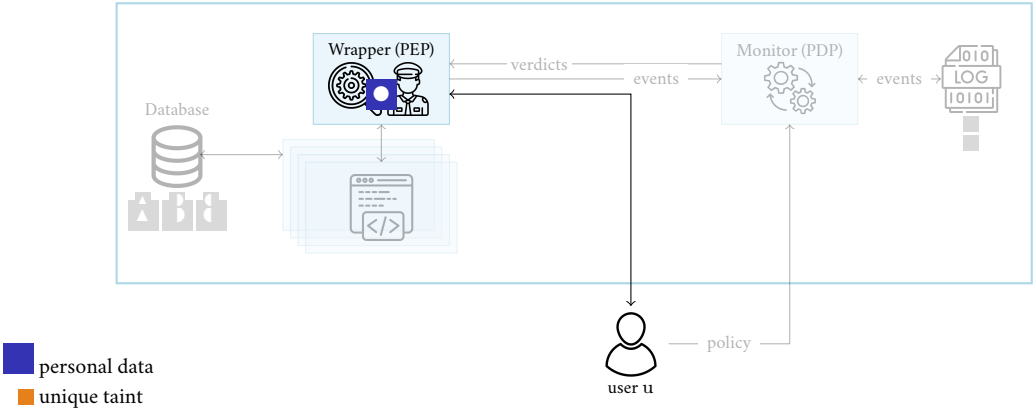
output + interference events: Out(u, p,), Itf(,), Itf(,)

monitor verdict: ✓

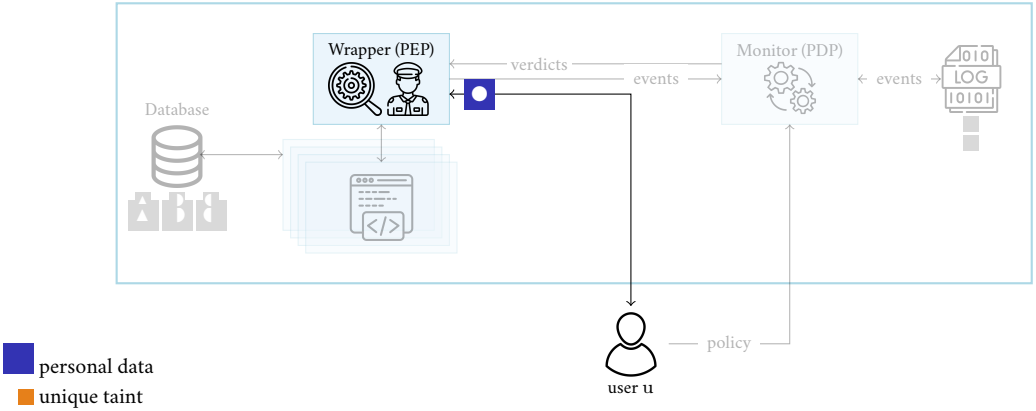
Taint, Track, and Control (TTC)



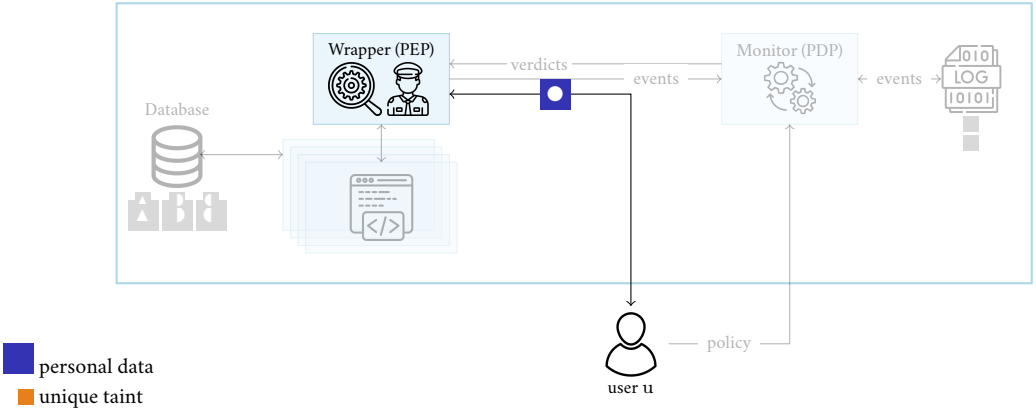
Taint, Track, and Control (TTC)



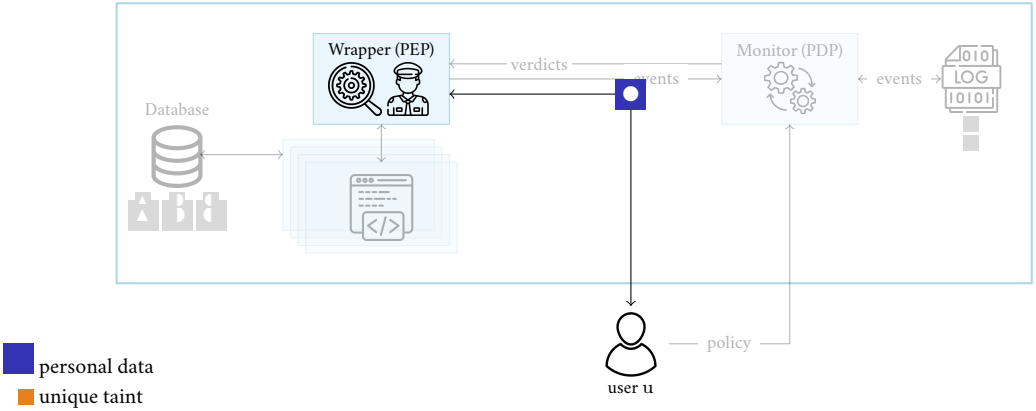
Taint, Track, and Control (TTC)



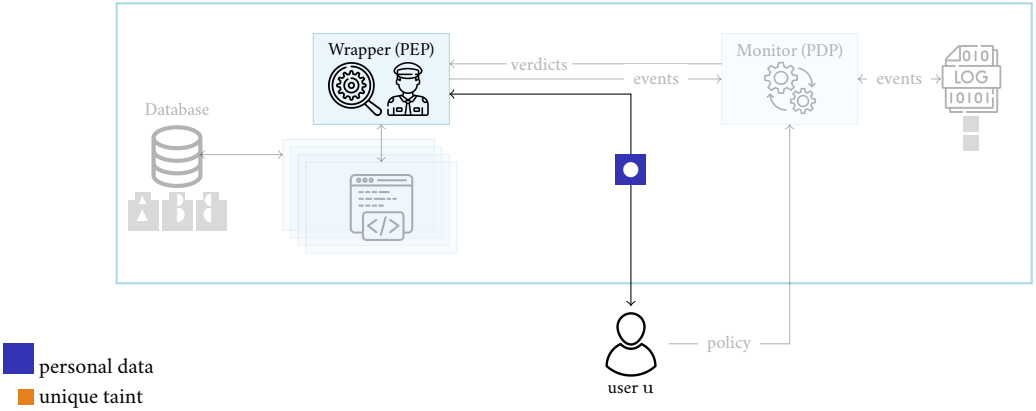
Taint, Track, and Control (TTC)



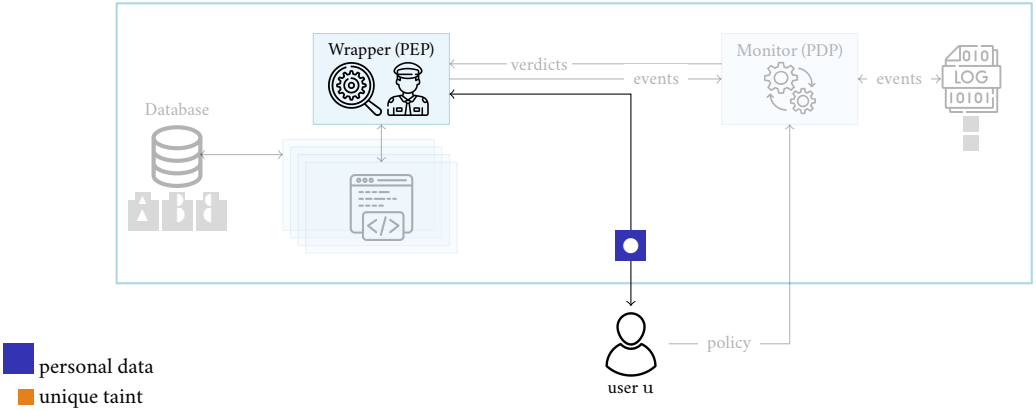
Taint, Track, and Control (TTC)



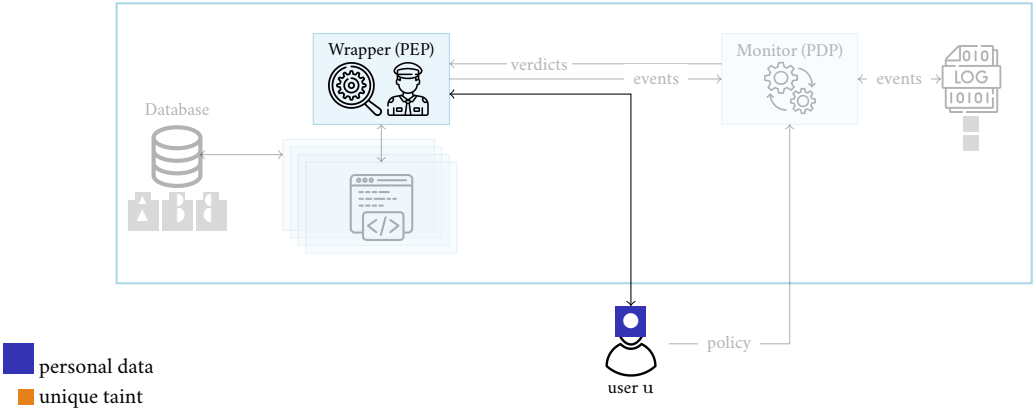
Taint, Track, and Control (TTC)



Taint, Track, and Control (TTC)



Taint, Track, and Control (TTC)



Taint & Control: Runtime Enforcement

In both the Taint and Control phases, events are generated, timestamped, and logged:

Phase	Event	Description
Taint	In(i)	The user inputs i

Taint & Control: Runtime Enforcement

In both the Taint and Control phases, events are generated, timestamped, and logged:

Phase	Event	Description
Taint	$\mathsf{In}(i)$	The user inputs i
Control	$\mathsf{Out}(u, p, o)$	The application outputs o to agent u for purpose p

Taint & Control: Runtime Enforcement

In both the Taint and Control phases, events are generated, timestamped, and logged:

Phase	Event	Description
Taint	$\mathsf{In}(\boxed{i})$	The user inputs $\boxed{i}$
Control	$\mathsf{Out}(u, p, o)$	The application outputs o to agent u for purpose p
Control	$\mathsf{Itf}(\boxed{i}, o)$	Input $\boxed{i}$ interferes with ($\approx$ influences) output o

$\boxed{i}$ = unique taint for input i

Taint & Control: Runtime Enforcement

In both the Taint and Control phases, events are generated, timestamped, and logged:

Phase	Event	Description
Taint	$\mathsf{In}(i)$	The user inputs i
Control	$\mathsf{Out}(u, p, o)$	The application outputs o to agent u for purpose p
Control	$\mathsf{Itf}(i, o)$	Input i interferes with ($\approx$ influences) output o

i = unique taint for input i

Monitor: Enfpoly (Hublet et al., 2022). Policy language: Metric First-Order Temporal Logic (MFOTL).

Taint & Control: Runtime Enforcement

In both the Taint and Control phases, events are generated, timestamped, and logged:

Phase	Event	Description
Taint	$\mathsf{In}(\boxed{i})$	The user inputs $\boxed{i}$
Control	$\mathsf{Out}(u, p, o)$	The application outputs o to agent u for purpose p
Control	$\mathsf{Itf}(\boxed{i}, o)$	Input $\boxed{i}$ interferes with ($\approx$ influences) output o

$\boxed{i}$ = unique taint for input i

Monitor: Enfpoly (Hublet et al., 2022). Policy language: Metric First-Order Temporal Logic (MFOTL).

Example: Supported policy

$$\Box\big[\forall u, p, o, i.\ \mathsf{Out}(u, p, o) \wedge \mathsf{Itf}(i, o) \Rightarrow \blacklozenge_{[0,\infty)}\ \mathsf{In}(i)$$
$$\Rightarrow\ (p \neq \mathit{"Marketing"}) \wedge (p = \mathit{"Analytics"} \Rightarrow u = \mathit{"trustedanalytics.com"}) \wedge (p = \mathit{"Service"} \Rightarrow \blacklozenge_{[0,1\ \mathtt{week}]}\ \mathsf{In}(i))\big]$$

Track: Dynamic Information Flow Control

Track: Dynamic Information Flow Control

- New programming language PythonTTC with dynamic IFC that propagates unique taints (UTs)

Track: Dynamic Information Flow Control

- New programming language PythonTTC with dynamic IFC that propagates unique taints (UTs)
- Inputs are marked with a single UT, data in memory/database with sets of UTs

Track: Dynamic Information Flow Control

- New programming language PythonTTC with dynamic IFC that propagates unique taints (UTs)
- Inputs are marked with a single UT, data in memory/database with sets of UTs
- Supports introspection, i.e., checking whether output is allowed ahead of time → see paper

Example

```
x, y, z = input()
if x == "Hi":
  r = y + z
else:
  r = ""
...
```

Variable	Value	UTs
x =	⟨	, ⟩
y =	⟨	, ⟩
z =	⟨	, ⟩
r =	⟨	, ⟩
pc =	⟨	, ⟩

Track: Dynamic Information Flow Control

- New programming language PythonTTC with dynamic IFC that propagates unique taints (UTs)
- Inputs are marked with a single UT, data in memory/database with sets of UTs
- Supports introspection, i.e., checking whether output is allowed ahead of time → see paper

Example

```
x, y, z = input()
if x == "Hi":
  r = y + z
else:
  r = ""
...
```

Variable	Value	UTs
x =	⟨	, ⟩
y =	⟨	, ⟩
z =	⟨	, ⟩
r =	⟨	, ⟩
pc =	⟨	, ⟩

Track: Dynamic Information Flow Control

- New programming language PythonTTC with dynamic IFC that propagates unique taints (UTs)
- Inputs are marked with a single UT, data in memory/database with sets of UTs
- Supports introspection, i.e., checking whether output is allowed ahead of time → see paper

Example

```
1. ▶ x, y, z = input()
2.   if x == "Hi":
3.     r = y + z
4.   else:
5.     r = ""
6.   ...
```

Variable	Value	UTs
x =	⟨	, ⟩
y =	⟨	, ⟩
z =	⟨	, ⟩
r =	⟨	, ⟩
pc =	⟨ 1	, ⟩

Track: Dynamic Information Flow Control

- New programming language PythonTTC with dynamic IFC that propagates unique taints (UTs)
- Inputs are marked with a single UT, data in memory/database with sets of UTs
- Supports introspection, i.e., checking whether output is allowed ahead of time → see paper

Example

```
1. ▶ x, y, z = input()
2.   if x == "Hi":
3.     r = y + z
4.   else:
5.     r = ""
6.   ...
```

Variable	Value	UTs
x =	⟨ "Hi" ,	X ⟩
y =	⟨ "PE" ,	Y ⟩
z =	⟨ "TS" ,	Z ⟩
r =	⟨ ,	⟩
pc =	⟨ 1 ,	⟩

Track: Dynamic Information Flow Control

- New programming language PythonTTC with dynamic IFC that propagates unique taints (UTs)
- Inputs are marked with a single UT, data in memory/database with sets of UTs
- Supports introspection, i.e., checking whether output is allowed ahead of time → see paper

Example

```
1.   x, y, z = input()
2. ▶ if x == "Hi":
3.     r = y + z
4.   else:
5.     r = ""
6.   ...
```

Variable	Value	UTs
x =	"Hi"	X
y =	"PE"	Y
z =	"TS"	Z
r =		
pc =	2	

Track: Dynamic Information Flow Control

- New programming language PythonTTC with dynamic IFC that propagates unique taints (UTs)
- Inputs are marked with a single UT, data in memory/database with sets of UTs
- Supports introspection, i.e., checking whether output is allowed ahead of time → see paper

Example

```
1.   x, y, z = input()
2. ▶ if x == "Hi":
3.     r = y + z
4.   else:
5.     r = ""
6.   ...
```

Variable	Value	UTs
x =	⟨ "Hi" ,	X ⟩
y =	⟨ "PE" ,	Y ⟩
z =	⟨ "TS" ,	Z ⟩
r =	⟨ ,	⟩
pc =	⟨ 2 ,	X ⟩

Track: Dynamic Information Flow Control

- New programming language PythonTTC with dynamic IFC that propagates unique taints (UTs)
- Inputs are marked with a single UT, data in memory/database with sets of UTs
- Supports introspection, i.e., checking whether output is allowed ahead of time → see paper

Example

```
1.    x, y, z = input()
2.    if x == "Hi":
3. ▶    r = y + z
4.    else:
5.      r = ""
6.    ...
```

Variable	Value	UTs
x =	"Hi"	X
y =	"PE"	Y
z =	"TS"	Z
r =		
pc =	3	X

Track: Dynamic Information Flow Control

- New programming language PythonTTC with dynamic IFC that propagates unique taints (UTs)
- Inputs are marked with a single UT, data in memory/database with sets of UTs
- Supports introspection, i.e., checking whether output is allowed ahead of time → see paper

Example

```
1.    x, y, z = input()
2.    if x == "Hi":
3. ▶    r = y + z
4.    else:
5.      r = ""
6.    ...
```

Variable		Value		UTs			
x =	⟨	"Hi"	,	X			⟩
y =	⟨	"PE"	,	Y			⟩
z =	⟨	"TS"	,	Z			⟩
r =	⟨	"PETS"	,	X	Y	Z	⟩
pc =	⟨	3	,	X			⟩

Track: Dynamic Information Flow Control

- New programming language PythonTTC with dynamic IFC that propagates unique taints (UTs)
- Inputs are marked with a single UT, data in memory/database with sets of UTs
- Supports introspection, i.e., checking whether output is allowed ahead of time → see paper

Example

```
1.    x, y, z = input()
2.    if x == "Hi":
3. ▶    r = y + z
4.    else:
5.      r = ""
6.    ...
```

Variable		Value		UTs	
x =	⟨	"Hi"	,	X	⟩
y =	⟨	"PE"	,	Y	⟩
z =	⟨	"TS"	,	Z	⟩
r =	⟨	"PETS"	,	X Y Z	⟩
pc =	⟨	3	,		⟩

Track: Dynamic Information Flow Control

- New programming language PythonTTC with dynamic IFC that propagates unique taints (UTs)
- Inputs are marked with a single UT, data in memory/database with sets of UTs
- Supports introspection, i.e., checking whether output is allowed ahead of time → see paper

Example

```
1.   x, y, z = input()
2.   if x == "Hi":
3.     r = y + z
4.   else:
5.     r = ""
6. ▶ ...
```

Variable		Value		UTs	
x =	⟨	"Hi"	,	X	⟩
y =	⟨	"PE"	,	Y	⟩
z =	⟨	"TS"	,	Z	⟩
r =	⟨	"PETS"	,	X Y Z	⟩
pc =	⟨	6	,		⟩

Formal guarantees and proofs

Formal guarantees and proofs

In our paper, we:

- Model online applications as labeled transition systems

Formal guarantees and proofs

In our paper, we:

- Model online applications as labeled transition systems
- Formalize interference between inputs and outputs

Formal guarantees and proofs

In our paper, we:

- Model online applications as labeled transition systems
- Formalize interference between inputs and outputs
- Provide formal semantics for a core of PythonTTC

Formal guarantees and proofs

In our paper, we:

- Model online applications as labeled transition systems
- Formalize interference between inputs and outputs
- Provide formal semantics for a core of PythonTTC
- Provide the semantics of the full TTC system (wrapper, monitor, applications)

Formal guarantees and proofs

In our paper, we:

- Model online applications as labeled transition systems
- Formalize interference between inputs and outputs
- Provide formal semantics for a core of PythonTTC
- Provide the semantics of the full TTC system (wrapper, monitor, applications)

Using Isabelle/HOL, we prove:

Theorem: 4.9

If the monitor is loaded with a policy P *that is enforceable,* ltf*-monotonic, and independent of past outputs*, then any code running in TTC complies with* P.

* see definitions in paper

Artifact and empirical evaluation

Artifact and empirical evaluation

Artifact: Reproduced

- Prototype web programming framework: WebTTC (~ 3500 loc) based on Python
- Three proof-of-concept applications:
  - CMS
  - Microblogging
  - Health record manager
- Full mechanized formalization and proofs in Isabelle/HOL (~ 5000 loc)

Artifact and empirical evaluation

Artifact: Reproduced

- Prototype web programming framework: WebTTC (~ 3500 loc) based on Python
- Three proof-of-concept applications:
  - CMS
  - Microblogging
  - Health record manager
- Full mechanized formalization and proofs in Isabelle/HOL (~ 5000 loc)

Evaluation

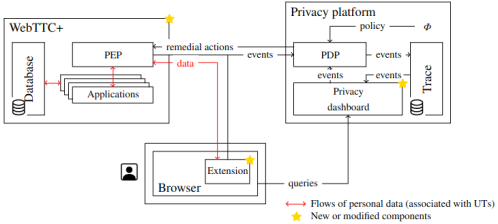
- Successfully ported POC applications from previous work
- Enforcement overhead of 1.5–2× over pure Flask provided that pagination is introduced
- Performance comparable or better than Jacqueline (Yang et al., 2016)

Applications and future work

Applications and future work

ESORICS'23

Applied WebTTC to enforce 10 GDPR requirements

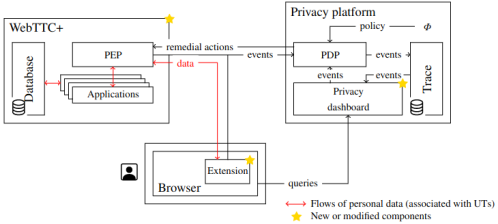


Source: Hublet et al. (2023). Enforcing the GDPR. *ESORICS*'23.

Applications and future work

ESORICS'23

Applied WebTTC to enforce 10 GDPR requirements



Source: Hublet et al. (2023). Enforcing the GDPR. *ESORICS'23*.

- Optimizing tracking using static analysis
- TTC in distributed systems
- Anonymization and declassification

Thank you for your attention!

If you are interested in this work, feel free to drop us an e-mail:

François Hublet `francois.hublet@inf.ethz.ch`
David Basin `basin@inf.ethz.ch`
Srđan Krstić `srdan.krstic@inf.ethz.ch`

User-Controlled Privacy: Taint, Track, and Control

François Hublet, ETH Zürich, francois.hublet@inf.ethz.ch
David Basin, ETH Zürich, basin@inf.ethz.ch
Srđan Krstić, ETH Zürich, srdan.krstic@inf.ethz.ch

ABSTRACT

We develop the first language-based, Privacy by Design approach that provides support for a rich class of privacy policies. The policies are user-defined, rather than programmer-defined, and support fine-grained information flow restrictions (considering individual application inputs and outputs) with temporal constraints. Our approach, called Taint, Track, and Control (TTC), combines dynamic information flow control and runtime verification to enforce these policies in the presence of malicious users and developers.

We provide TTC's semantics and proofs of correct enforcement, formalized in the Isabelle/HOL proof assistant. We also implement our approach in a web development framework and port three baseline applications from previous work into this framework for evaluation. Overall, we find that our approach enforces expressive user-defined privacy policies with practical runtime performance.

1 INTRODUCTION

Motivation and Problem Statement. Over the last decade, new legislation, such as the European Union's General Data Protection Regulation (GDPR), has paved the way for a more extensive recognition of individuals' right to privacy. While, from a legal viewpoint, the GDPR has set a de-facto global standard [68], the level of effective privacy enjoyed by users of online applications remains unsatisfactory. Amid reports of widespread non-compliance, the gap between expectations and real-world practice has only become more evident.

Privacy by Design (PbD), the principle of "design[ing] and develop[ing] products with a built-in ability to demonstrate compliance" [76], offers a promising path to improving the status quo. With Privacy by Design, *privacy policies* are specified that define who can use which data, when, and for which purposes. Incentives for practitioners to adopt PbD include developing or maintaining a competitive advantage in privacy-critical markets [51] and avoiding the costs resulting from major privacy breaches and non-compliance [42, 46]. Moreover, Article 25 GDPR makes it obligatory for data controllers to use 'state-of-the-art' techniques to implement data protection principles [14]. Whenever technologies for ensuring compliance are available "at a reasonable price," controllers are legally obliged to use these (or similar) technologies [35].

Privacy policies can be specified by developers (e.g., at the code level) or end-users (e.g., through a policy management interface). An example of a user-specified policy that addresses the key GDPR concern of *purpose* limitation is policy P_1 in Figure 1, which states that *Alice's personal data* shall never be used for marketing. Assuming that *Alice's personal data* is only input by herself, a conservative

Proceedings on Privacy Enhancing Technologies 2024(1), 297–410
© 2023 Copyright held by the author(s).
https://doi.org/10.56553/popets-2024-0014

Policy Textual description

- P_1: "Alice's personal data can be used for any purpose"
- P_2: "Alice's personal data shall never be used for marketing purposes"
- P_3: "Alice's posts in the Minitwitter app shall never be used for marketing purposes; after one week, it shall only be used for service purposes; for analytics purposes, it can only be sent to trustedanalytics.com, but not to any other party"
- P_4: "Alice's personal data shall only be shown to herself"

Figure 1: Example privacy policies for Alice

interpretation of this policy is that *Alice's inputs* and any *data derived from those inputs* shall never be used for marketing purposes.

Privacy policies can be enforced using a language-based approach [74]: developers write applications in a specially designed programming language that guarantees the enforcement of policies either statically or dynamically. Language-based PbD can in turn be implemented as an extension of traditional information-flow control (IFC) [55, 74], leveraging the similarity between the two approaches: both PbD and IFC must restrict data flows and guarantee that, by *default*, the protection enjoyed by data *items* extends to any other data derived from them. But despite these similarities, existing IFC designs cannot be directly reused for PbD, since the assumptions made by these designs do not match the requirements of privacy laws. In particular, the following requirements stand out:

[R1] *User-specified policies.* Privacy policies must be specified by individual users, rather than developers. In the GDPR, this reflects that the allowed usage of data depend on end-user consent, which must be freely given [1, Art. 7].

[R2] *Per-input policies.* The policy language must distinguish between different user inputs to support fine-grained restrictions on data usage, e.g., to provide the additional protection enjoyed by special data categories [1, Art. 9]. Users must be able to define different restrictions for each of their inputs.

[R3] *Time-dependent policies.* The policy language must allow users to define restrictions that apply only for a specific time period. For example, the GDPR has the notion of a *storage period* [1, Rec. 85] during which data can be used and retained.

Policy P_3 in Figure 1 exemplifies these requirements: user Alice demands that her posts in a microblogging platform *are never used* for marketing purposes, that only a single trusted third-party can be sent information about the posts for analytics purposes, and that after a week, the messages can only be used for service purposes. These limitations extend to all data derived from her posts. The policy is user-specific (only concerning Alice's inputs), per-input (only applying to Alice's posts), and time-dependent (no additional constraints apply after a week).

State of the Art. Requirements. [R1–3] are out of the scope of most previous work on IFC. Existing approaches usually provide little or no support for time-dependent policies [R3], and define restrictions

Track: Supporting introspection

Track: Supporting introspection

- Policies are user-controlled — developers cannot simply assume a specific policy

Track: Supporting introspection

- Policies are user-controlled — developers cannot simply assume a specific policy
- If programs cannot check what data can be output for what purpose, many outputs will be blocked

Track: Supporting introspection

- Policies are user-controlled — developers cannot simply assume a specific policy
- If programs cannot check what data can be output for what purpose, many outputs will be blocked
- Need to support introspection: `check`(x, u, p) with data x, user u, purpose p

Track: Supporting introspection

- Policies are user-controlled — developers cannot simply assume a specific policy
- If programs cannot check what data can be output for what purpose, many outputs will be blocked
- Need to support introspection: `check`(x, u, p) with data x, user u, purpose p
- Requires ordering of taints: **UT histories** (see paper)

Track: Supporting introspection

- Policies are user-controlled — developers cannot simply assume a specific policy
- If programs cannot check what data can be output for what purpose, many outputs will be blocked
- Need to support introspection: `check`(x, u, p) with data x, user u, purpose p
- Requires ordering of taints: **UT histories** (see paper)

Example: Showing timeline in a microblogging app

```
def filter_check(posts):
  posts2 = []
  for post in posts:
    posts2 += [post] if check("Service", me(), post) else []
  return posts2

@route('/<username>')
def user_timeline(username):
  posts = sql("SELECT * FROM posts WHERE username = ?0", [username])
  posts = filter_check(posts)
  ad    = generate_ad(posts) if check("Marketing", me(), posts) else generate_ad([])
  return render("timeline.html",
                {"posts": ("Service", posts), "ad": ("Marketing", ad)})
```

Track: Support introspection (cont'd)

Track: Support introspection (cont'd)

- The `check`$(\mathtt{x}, \mathtt{u}, \mathtt{p})$ function calls the enforcer on the UTs of $\mathtt{x}$

Track: Support introspection (cont'd)

- The `check`(x, u, p) function calls the enforcer on the UTs of x
- Since the UTs of a variable may depend on personal data, this induces new information flows!

Track: Support introspection (cont'd)

- The `check`(x, u, p) function calls the enforcer on the UTs of x
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Track: Support introspection (cont'd)

- The `check`(`x`, `u`, `p`) function calls the enforcer on the UTs of `x`
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
x, y, z = input()
if x == "Hi":
  r = y + z
else:
  r = ""
...
```

Variable	Value	UTs
x =	⟨	, ⟩
y =	⟨	, ⟩
z =	⟨	, ⟩
r =	⟨	, ⟩
pc =	⟨	, ⟩

Track: Support introspection (cont'd)

- The `check`(`x`, `u`, `p`) function calls the enforcer on the UTs of `x`
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
1. ▶ x, y, z = input()
2.   if x == "Hi":
3.     r = y + z
4.   else:
5.     r = ""
6.   ...
```

Variable	Value	UTs
x =	⟨	, ⟩
y =	⟨	, ⟩
z =	⟨	, ⟩
r =	⟨	, ⟩
pc =	⟨ 1	, ⟩

Track: Support introspection (cont'd)

- The `check`(`x`, `u`, `p`) function calls the enforcer on the UTs of `x`
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
1. ▶ x, y, z = input()
2.   if x == "Hi":
3.     r = y + z
4.   else:
5.     r = ""
6.   ...
```

Variable	Value	UTs
x =	"Hi"	[X]
y =	"PE"	[Y]
z =	"TS"	[Z]
r =		
pc =	1	

Track: Support introspection (cont'd)

- The `check`(x, u, p) function calls the enforcer on the UTs of x
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
1.   x, y, z = input()
2. ▶ if x == "Hi":
3.     r = y + z
4.   else:
5.     r = ""
6.   ...
```

Variable	Value	UTs
x =	"Hi"	[X]
y =	"PE"	[Y]
z =	"TS"	[Z]
r =		
pc =	2	

Track: Support introspection (cont'd)

- The `check`(x, u, p) function calls the enforcer on the UTs of x
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
1.   x, y, z = input()
2. ▶ if x == "Hi":
3.     r = y + z
4.   else:
5.     r = ""
6.   ...
```

Variable	Value	UTs
x =	"Hi"	[X]
y =	"PE"	[Y]
z =	"TS"	[Z]
r =		
pc =	2	[X]

Track: Support introspection (cont'd)

- The `check`(`x`, `u`, `p`) function calls the enforcer on the UTs of `x`
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
1.    x, y, z = input()
2.    if x == "Hi":
3. ▶    r = y + z
4.    else:
5.      r = ""
6.    ...
```

Variable	Value	UTs
x =	"Hi"	[X]
y =	"PE"	[Y]
z =	"TS"	[Z]
r =		
pc =	3	[X]

Track: Support introspection (cont'd)

- The `check`(x, u, p) function calls the enforcer on the UTs of x
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
1.    x, y, z = input()
2.    if x == "Hi":
3. ▶    r = y + z
4.    else:
5.      r = ""
6.    ...
```

Variable	Value	UTs
x =	"Hi"	[X]
y =	"PE"	[Y]
z =	"TS"	[Z]
r =	"PETS"	[X, Y Z]
pc =	3	[X]

Track: Support introspection (cont'd)

- The `check`(`x`, `u`, `p`) function calls the enforcer on the UTs of `x`
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
1.    x, y, z = input()
2.    if x == "Hi":
3. ▶    r = y + z
4.    else:
5.      r = ""
6.    ...
```

Variable	Value	UTs
x =	"Hi"	[X]
y =	"PE"	[Y]
z =	"TS"	[Z]
r =	"PETS"	[X, Y Z]
pc =	3	

Track: Support introspection (cont'd)

- The `check`(`x`, `u`, `p`) function calls the enforcer on the UTs of `x`
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
1.    x, y, z = input()
2.    if x == "Hi":
3.      r = y + z
4.    else:
5.      r = ""
6. ▶  ...
```

Variable	Value	UTs
x =	"Hi"	[X]
y =	"PE"	[Y]
z =	"TS"	[Z]
r =	"PETS"	[X, Y Z]
pc =	6	

Track: Support introspection (cont'd)

- The `check`(`x`, `u`, `p`) function calls the enforcer on the UTs of `x`
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
6.  ▶ if check(r, "Alice", "purp"):
7.      s = r
8.    else:
9.      s = ""
10.   ...
```

Variable	Value	UTs
r =	"PETS"	[X, Y, Z]
s =		
pc =	6	

Track: Support introspection (cont'd)

- The `check(x, u, p)` function calls the enforcer on the UTs of `x`
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
6. ▶ if check(r, "Alice", "purp"):
7.     s = r
8.   else:
9.
10.
```

Variable	Value	UTs
`r =`	⟨ `"PETS"` ,	[X, Y Z] ⟩
`s =`	⟨ ,	⟩
		⟩

Can X be output to `"Alice"` for purpose `"purp"`?

Track: Support introspection (cont'd)

- The `check`(`x`, `u`, `p`) function calls the enforcer on the UTs of `x`
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
6. ▶ if check(r, "Alice", "purp"):
7.     s = r
8.   else:
9.
10.
```

Variable	Value	UTs
`r =`	⟨ `"PETS"` ,	[X , Y Z] ⟩
`s =`	⟨ ,	⟩
		⟩

Can X be output to `"Alice"` for purpose `"purp"`? Monitor: **Yes.**

Track: Support introspection (cont'd)

- The `check(x, u, p)` function calls the enforcer on the UTs of `x`
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
6. ▶ if check(r, "Alice", "purp"):
7.     s = r
8.   else:
9.
10.
```

Variable	Value	UTs
`r =`	⟨ `"PETS"` ,	[X , Y Z] ⟩
`s =`	⟨ ,	⟩
		⟩

Can Y and Z be output to `"Alice"` for purpose `"purp"`?
Already checked UTs = [X]

Track: Support introspection (cont'd)

- The `check`(`x`, `u`, `p`) function calls the enforcer on the UTs of `x`
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
6. ▶ if check(r, "Alice", "purp"):
7.     s = r
8.   else:
9.
10.
```

Variable	Value	UTs
`r` =	⟨ `"PETS"` ,	[X , Y Z] ⟩
`s` =	⟨ ,	⟩
		⟩

Can Y and Z be output to `"Alice"` for purpose `"purp"`? Monitor: **No.**
Already checked UTs = [X]

Track: Support introspection (cont'd)

- The `check`(`x`, `u`, `p`) function calls the enforcer on the UTs of `x`
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
6.  ▶ if check(r, "Alice", "purp"):
7.      s = r
8.    else:
9.
10.
```

Variable	Value	UTs
`r` =	⟨ `"PETS"` ,	[X , Y Z] ⟩
`s` =	⟨ ,	⟩
		⟩

Can Y and Z be output to `"Alice"` for purpose `"purp"`? Monitor: **No.**

Already checked UTs = [X] ⟹ `check(r, "Alice", "purp")` = ⟨`False`, [X]⟩

Track: Support introspection (cont'd)

- The `check`(`x`, `u`, `p`) function calls the enforcer on the UTs of `x`
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
6.    if check(r, "Alice", "purp"):
7.      s = r
8.    else:
9. ▶    s = ""
10.   ...
```

Variable	Value	UTs
r =	⟨ "PETS" ,	[X, Y Z] ⟩
s =	⟨ ,	⟩
pc =	⟨ 9 ,	[X] ⟩

Track: Support introspection (cont'd)

- The `check`(`x`, `u`, `p`) function calls the enforcer on the UTs of `x`
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
6.    if check(r, "Alice", "purp"):
7.      s = r
8.    else:
9. ▶    s = ""
10.   ...
```

Variable	Value	UTs
r =	"PETS"	[X, Y Z]
s =	""	[X]
pc =	9	[X]

Track: Support introspection (cont'd)

- The `check`(x, u, p) function calls the enforcer on the UTs of x
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
6.    if check(r, "Alice", "purp"):
7.      s = r
8.    else:
9. ▶    s = ""
10.   ...
```

Variable	Value	UTs
r =	⟨ "PETS" ,	[X, Y Z] ⟩
s =	⟨ "" ,	[X] ⟩
pc =	⟨ 9 ,	⟩

Track: Support introspection (cont'd)

- The `check`(`x`, `u`, `p`) function calls the enforcer on the UTs of `x`
- Since the UTs of a variable may depend on personal data, this induces new information flows!
- Solution: Introduce **UT histories** = lists of sets of UTs

Example

```
6.    if check(r, "Alice", "purp"):
7.      s = r
8.    else:
9.      s = ""
10.▶  ...
```

Variable	Value	UTs
r =	"PETS"	[X, Y Z]
s =	""	[X]
pc =	10	