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 combine 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 its 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, 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" [74], 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 practictioners to adopt PbD include developing or maintaining a competitive advantage in privacy-critical markets [53] and avoiding the costs resulting from major privacy breaches and non-compliance [44, 66]. Moreover, Article 25 GDPR makes it obligatory for data controllers to use 'state-of-the-art' techniques to implement data protection principles [16]. Whenever technologies for ensuring compliance are available "at a reasonable price," controllers are legally obliged to use these (or similar) technologies [57].

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

Policy	Textual description
P_0	"Alice's personal data can be used for any purpose"
P_1	"Alice's personal data shall never be used for marketing purposes"
P_2	"Alice's posts in the Minitwit 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_3	"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: PbD shares with IFC the goal of restricting data flows, as well as the expectation that the protection enjoyed by data items extends to other data derived from them. But despite these similarities, existing IFC designs cannot be directly reused for PbD, since the assumptions of these designs do not match those required for privacy. 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 should 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. 45] during which data can be used and retained.

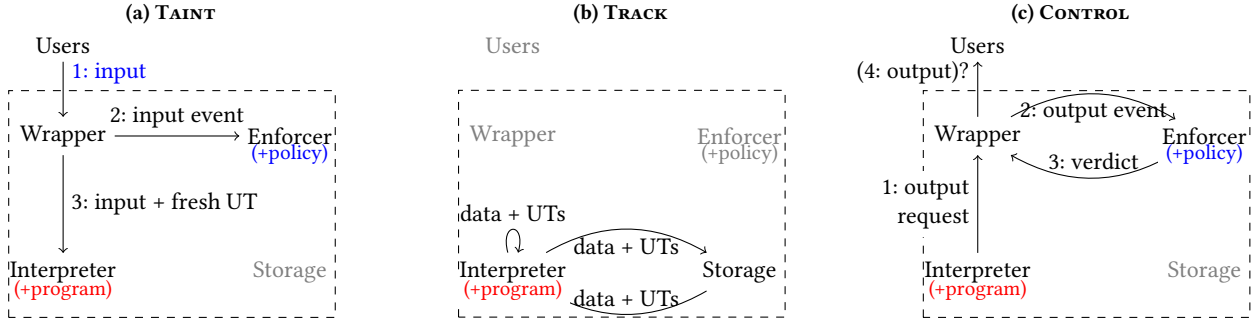
Policy P_2 in Figure 1 examplifies these requirements: user Alice requires that her posts in a microblogging platform are never used for marketing purposes, that only a single trusted third-party can be sent information about them for analytics purposes, and that after a week, the messages can only be 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 (as 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

Proceedings on Privacy Enhancing Technologies YYYY(X), 1–19

https://doi.org/XXXXXXX.XXXXXXX

Figure 2: Taint, Track, and Control



Grayed-out components are inactive in the corresponding phase. The operations in Track take place in any order. Blue: User-provided. Red: Developer-provided.

on information flow in terms of the data model [7, 20, 23, 27, 47, 64, 77, 78], rather than single inputs [R2]. Moreover, and critically, their focus is on developer-specified, rather than user-specified, policies [R1]. In some works [23, 47, 77, 78], developers can build custom user interfaces where users can select from different predefined policies, this requires additional effort and does not provide users with formal guarantees at the level of individual inputs.

To enforce user-defined policies [R1], dynamic approaches appear most promising. First-order temporal policies [R2–3] are well-supported by those runtime enforcement (RE) approaches developed within the runtime verification community [58, 69]. In such approaches, timestamped *event traces* model executions of an application at runtime (e.g., events represent specific actions, like the applications' inputs and outputs). The policy enforced is defined as a set of such traces, i.e., a *trace property*. To ensure compliance, some of the application's actions (typically, outputs) can be suppressed. Concretely, a Policy Enforcement Point (PEP) is informed before each such action is performed; the PEP reports candidate events to a Policy Decision Point (PDP) that then check the compliance of the resulting trace with the policy. Based on the PDP's verdict, the PEP either allows or suppresses the action. To cover [R1–3], a temporal first-order policy language [9, 54] can be chosen, and the overall policy to be enforced can be expressed as the conjunction of individual policies specified by end-users.

Our solution. In this paper, we develop the first language-based PbD approach that combines IFC and trace-based runtime enforcement to satisfy the requirements [R1–3]. We support a broad class of applications, which we call *online applications*, that run and continuously interact with users by collecting and processing their data (Section 2). This class includes (but is not restricted to) web applications, services, or other applications with an event-driven behavior. Our approach, called *Taint, Track, and Control* (TTC), brings together (i) an **enforcer** for *metric first-order temporal policies* [58] serving as the PDP, (ii) an **interpreter** for a programming language implementing a variant of dynamic IFC, (iii) a **wrapper** that controls inputs and outputs and serves as the PEP, and (iv) persistent **storage**, such as a database or file system. Once the enforcer is loaded with a policy and the interpreter with a program defining the application's functionality, user inputs are handled in three phases (see Figure 2):

Taint: The wrapper receives a new user input and tags it with a fresh *unique taint* (UT) (1). The UT is a unique bitstring identifying one specific input during its lifetime. The wrapper informs the enforcer about the new UT by emiting an event (2). The input and UT are sent to the interpreter (3).

Track: The semantics implemented by the interpreter ensures that every data item in memory is tagged with *the UTs of all inputs that have influenced its current value*. The interpreter interacts with persistent storage, to which UTs are propagated.

Control: Any output that the interpreter attempts to perform must go through the wrapper (1). Upon receiving an output request, the wrapper generates events capturing the attempted output and all user inputs that influenced it. To identify the latter, it uses the UTs of the candidate output. These events are sent to the enforcer (2), which can accept or reject the output (3). If its verdict is positive, the wrapper emits the output (4).

TTC guarantees the enforcement of users' privacy policies against an adversary that can impersonate other users or tamper with application code. The latter covers the realistic threat scenario of a malicious developer with no direct access to the application's production data. In particular, we make the following contributions:

(1) We introduce *information-flow traces* capturing inputs, outputs, and interference between them, and we specify privacy policies covering [R1–3] using such traces (Section 3).
(2) We introduce TTC, the first language-based PbD approach for enforcing such privacy policies in online applications. We provide formal semantics and correctness proofs checked in the Isabelle/HOL proof assistant (Section 4, Appendix A).
(3) We implement WebTTC, a proof-of-concept web development framework that incorporates TTC. WebTTC applications enforce user-defined privacy policies (Section 5).
(4) We demonstrate TTC's ease of use and efficiency by presenting and evaluating our development framework. We port applications from previous works to our framework and show their practical runtime performance (Section 6).

As consent is at the core of the GDPR's design [68] and omnipresent in web services [17], tackling consent-based PbD is an important first step towards more comprehensive enforcement of legal requirements. Still, requirements [R1-3] do not cover the full

Figure 4: Excerpt from the code of Minitwit

```
1  def generate_ad(posts):      # Returns personalized ads
2    ...
3
4  @route('/<username>')        # Displays a user's posts
5  def user_timeline(username):
6    posts = sql("""
7      SELECT post.pub_date, post.id, post.author_id,
8             post.text, user.username FROM post
9      JOIN user ON user.id = post.author_id
10     WHERE user.username = ?0""", [username])
11   ad = generate_ad(posts)
12   send("trustedanalytics.com", "Analytics", username)
13   send("evilanalytics.com", "Analytics", username)
14   return render("timeline.html",
15                 {"posts": ("Service", posts),
16                  "ad":    ("Marketing", ad)})
```

Figure 5: Excerpt from the code of Minitwit, II

```
1  def filter_check(posts):
2    posts2 = []
3    for post in posts:
4      posts2 += [post] if check("Service",posts) else []
5    return posts2
6
7  @route('/<username>')        # Displays a user's posts
8  def user_timeline(username):
9    posts = sql("""..."""", [username])
10   posts = filter_check(posts)
11   ad = generate_ad(posts) if check("Marketing", posts)
          else generate_ad("")
12   send("trustedanalytics.com", "Analytics", username)
13   send("evilanalytics.com", "Analytics", username)
14   return render("timeline.html",
15                 {"posts": ("Service", posts),
16                  "ad":    ("Marketing", ad)})
```

range of legal bases for processing defined in the GDPR: e.g., necessity for fulfilling a contract or the controller's "legitimate interest" can also be invoked [1, Art. 6]. Therefore, in the conclusion (Section 8), we discuss how our work could be extended to support a more comprehensive fragment of privacy laws.

2 ENFORCEMENT IN ONLINE APPLICATIONS

In this section, we first introduce the application model that we address (Section 2.1). We then refine it into a model of applications that run in an architecture featuring both an interpreter and an enforcer (Section 2.2). Finally, we define our adversary model (Section 2.3).

2.1 Modeling online applications

In this paper, we will consider applications that repeatedly receive queries from users, collect user data (inputs), process it, and send data back to users (outputs). This covers web applications as well as other types of applications such as microservices. We generically refer to such applications as *online applications*. Further, we assume that each output of an online application is labeled with a GDPR-style purpose, such as "Service" (if the output is necessary for offering the service for which the application was primarily developed), "Marketing" (for, e.g., ads), or "Analytics".

2.1.1 An example. The code in Figure 4, taken from a simple microblogging platform, examplifies the behavior of online applications. Users can call a function `user_timeline` that takes a `username` and displays the user's posts. The first part of the function's code (l. 6–10) extracts `username`'s posts from the database into a variable `posts`. It then generates personalized ad in the variable `ad` based on the posts' content (l. 11). Next, `username` is sent to two third-parties (l. 12-13) for analytics purposes. Finally, the page is rendered using `posts` (for service purpose: the application's primary function is to allow users to see each other's posts) and `ad` (for marketing purpose) and returned to the caller (l. 18-20).

2.1.2 Model and assumptions. More formally, we consider applications that execute the following input-output loop. First, an input is received by an application endpoint from some user. This input is processed, modifying the application's internal state and resulting in a sequence of outputs to different users, which include both users of the application and third-parties such as other controllers. After producing its outputs, the application returns to a state where it can accept further inputs. Executions of such applications can be modeled as *deterministic* sequences of transitions of one of the following three forms:

$$S \xrightarrow{\mathrm{i}(u,f,\{a_i \mapsto d_i\}_{1\le i\le k},\tau)} S', \quad S \xrightarrow{\mathrm{o}(f,u,p,d,\tau)} S', \quad \text{or} \quad S \xrightarrow{\bullet} S'.$$

These respectively correspond to reading k simultaneous input arguments, producing a single output, and performing non-IO transitions. In the above transitions, u is the user producing the input or receiving the output; the d_i are the input or output values; f is the function performing IO; and $\tau \in \mathbb{N}$ is the timestamp according to some global clock. For inputs, $(a_i)_{1\le i\le k}$ are the (distinct) arguments of f to which the inputs are passed. For outputs, p is the output's purpose. Non-IO transitions are marked with the special symbol $\bullet$. We assume that we have a fixed initial state S_0 and monotonic timestamps, i.e., if $S \xrightarrow{\dots(\dots,\tau)} S' \to \cdots \to S'' \xrightarrow{\dots(\dots,\tau')} S'''$, then $\tau' > \tau$.

Example 2.1. Assume that at the timepoint τ_0, the user Alice calls `user_timeline` (l. 6) with username Bob. Her input has label

$$\mathrm{i}(\mathtt{Alice}, \mathtt{user\_timeline}, \{\mathtt{username} \mapsto \mathtt{Bob}\}, \tau_0)\,.$$

Performing a $\bullet$ transition, the function computes the list of Bob's posts and the corresponding ad. It then produces four outputs with respective labels

$$\mathrm{o}(\mathtt{view\_timeline}, \mathtt{trustedanalytics.com}, \mathtt{Analytics}, \mathtt{Bob}, \tau_1)$$
$$\mathrm{o}(\mathtt{view\_timeline}, \mathtt{evilanalytics.com}, \mathtt{Analytics}, \mathtt{Bob}, \tau_2)$$
$$\mathrm{o}(\mathtt{view\_timeline}, \mathtt{Alice}, \mathtt{Service}, \mathit{posts}, \tau_3)$$
$$\mathrm{o}(\mathtt{view\_timeline}, \mathtt{Alice}, \mathtt{Marketing}, \mathit{ad}, \tau_4)$$

where *posts* and *ad* stand for the value of the respective variable at the time after line 16, and $\tau_0 < \tau_1 < \tau_2 < \tau_3 < \tau_4$.

Since the relation $\to$ is deterministic, the application's state only depends on the sequence of timestamped inputs that it receives. This sequence can be represented as $\left(u_i, f_i, \{a_{ij} \mapsto d_{ij}\}_{1\le j\le k_i}, \tau_i\right)_{1\le i\le p}$, where u_i are users, f_i are functions, a_{ij} are the names of input arguments, d_{ij} are the input values, and $\tau_{i+1} > \tau_i$ for all i. When, starting in the state S with input sequence I, the application can reach the new state S' with a remaining (non-processed) input sequence I', we write $S, I \Rightarrow S', I'$.

2.2 Privacy-enforcing applications

Next, we refine the previous formalism into a model of applications whose behavior results from the interplay between an interpreter and an enforcer. We first recall standard definitions of traces and enforcers, and then present our model.

2.2.1 Traces and enforcers. A *signature* is a triple $\Sigma = (\mathbb{D}, \mathbb{A}, \iota)$ where $\mathbb{D} \supseteq \mathbb{N}$ is an infinite set of constant symbols, $\mathbb{A}$ is a finite set of actions, and $\iota : \mathbb{A} \to \mathbb{N}$ is an arity function. An *event* over the signature Σ is an expression of the form $a(d_1, \ldots, d_{\iota(a)})$, where $a \in \mathbb{A}$ and $d_i \in \mathbb{D}$, that encodes an action and its parameters. A *trace* stores the full history of the actions performed by the applications; it is a finite timestamped sequence of sets of events, i.e., a sequence $\sigma = (\tau_i, D_i)_{1 \leq i \leq k}$ where the τ_i are strictly increasing timestamps, and the D_i are finite sets of events. The concatenation of two traces t and t' is denoted by $t \cdot t'$, assuming that the first timestamp of t' is larger than the last timestamp of t. The set of events (resp. traces) over a signature Σ is denoted by $\mathbb{E}_\Sigma$ (resp. $\mathbb{T}_\Sigma$). Finally, a *policy* P is a subset $P \subseteq \mathbb{T}_\Sigma$. Any trace t in P is called *compliant* with P.

For the purpose of enforcement, the system's actions are all observable, and must be classified according to whether they can *only* be observed (only-observable actions, Obs $\subseteq \mathbb{A}$), or can also be suppressed at the PEP (suppressable actions, Sup $\subseteq \mathbb{A}$).[1] An *enforcer* for a policy P is a function that takes a trace σ that complies with P and a new timestamped set of events, examines the set, and returns a subset of these events that should be suppressed in order for the trace to remain compliant. More formally, given a trace σ and (τ, D) such that $\sigma \cdot (\tau, D)$ is a trace and σ complies with P, an enforcer for P returns a subset $D' \subseteq D$ of suppressable events such that $\sigma \cdot (\tau, D \setminus D')$ complies with P. The set D' is called the enforcer's *verdict*. Any policy P such that the empty trace complies with P and an enforcer exists for P is called *enforceable*.

Example 2.2. If the action capturing the outputs of a system is suppressable, the policy "the system shall perform no output to Alice" is enforceable; a corresponding enforcer would simply suppress any tentative output to Alice that the system performs. The policy "the system shall perform an output to Alice" is not enforceable, since the system can suppress, but not cause outputs.

2.2.2 Applications. The rest of the paper considers applications whose behavior is uniquely determined by the interplay of two components: an *interpreter* component and an *enforcer component*.

The interpreter component is loaded with a *program* π, provided by developers, which encodes the application's functionality.

The enforcer component provides an enforcer $\mathcal{E}$ for an enforceable policy P over some signature Σ. To support user-defined policies, the privacy policy P is assumed to be of the form $\cap_{u \in \mathbb{U}} P_u$, where $\mathbb{U}$ is the set of users and for all $u \in \mathbb{U}$, P_u denotes the individual policy of user u. The intersection $\cap_{u \in \mathbb{U}} P_u$ is the property expressing the simultaneous compliance with all $(P_u)_{u \in \mathbb{U}}$.

A *privacy-enforcing application* is an online application whose behavior is fully determined by (i) user inputs (ii) the program executed by the interpreter and (iii) users' policies. From the point of view of runtime enforcement, the inputs to a privacy-enforcing application are provided by users and are thus only-observable, while the outputs are performed by the application and are thus suppressable. We obtain a family of transition relations for the entire application of the form $(\to_{\pi,P})$, where π ranges over the set of programs and P over the set of policies. We write $\Rightarrow_{\pi,P}$ for the relation $\Rightarrow$ that we obtain as in Section 2.1.2 by setting $\to$ to $\to_{\pi,P}$.

[1] Some actions can be observed and *caused* by the enforcer [12]. We focus on the only-observable and suppressable actions.

2.3 Adversary model

In this paper, we consider the enforcement of users' privacy policies against both malicious users (end-users or third-parties) and malicious developers that can tamper with program code.

We consider an adversary that can impersonate both users and developers. As a user u, the adversary can interact with the application, observe its outputs, and set and read the policy P_u. As a developer, she can observe and set the program π. The adversary, however, can neither observe nor directly influence the content of persistent storage or the policies P_u. Neither can she observe or modify the internal state of the interpreter, wrapper, and enforcer components, or tamper with the network. In practice, the assumption that the adversary cannot directly tamper with the behavior of the TTC components is sufficient as long as (malicious) developers have no access to the production infrastructure, which is managed by a distinct group of trusted privacy-compliance specialists.

Furthermore, for purpose-based usage to be correctly enforced, we assume that a trusted party (e.g., privacy compliance specialists or external auditors) has certified that the purposes labeling the various outputs generated by π are appropriately set. This should apply irrespective of whether the adversary has impersonated the developers or not. This assumption may seem strong, but it is unavoidable once concepts such as 'purpose of processing' are introduced; currently, such checks can only be performed by privacy or legal experts, and are not readily automatable. In practice, this requirement can be fulfilled via appropriate organizational measures, such as compulsory validation of annotations by compliance teams whenever production code is deployed or modified.

Finally, we consider a termination and timing-insensitive setup in which the time intervals between inputs and outputs do not convey information about the values of inputs, and every function is assumed to terminate after a fixed duration. To this end, we require that every function performs a fixed number of outputs at fixed time intervals after receiving any set of inputs, and that this interval is smaller than the interval between consecutive inputs.

3 TRACES AND PRIVACY POLICIES

In this section, we develop TTC's policy language in three steps. First, we show how to encode the IO behavior of applications by collecting i and o labels into *input-output (IO) traces* (Section 3.1). Second, we use sets of IO traces to define *information-flow traces* (Section 3.2) that capture interference between inputs and outputs. Finally, we show how to use these information-flow traces to specify *privacy policies* covering requirements [R1-3] (Section 3.3).

3.1 Input-output traces

Applications' input-output behavior can be faithfully encoded into a trace over $\Sigma_{\text{IO}} = (\mathbb{D}, \{\mathsf{in}, \mathsf{out}\}, \{\mathsf{in} \mapsto 4, \mathsf{out} \mapsto 4\})$ (see also Figure 6). For each timestamp τ_i when a transition occurs, we generate a set of events E_i over Σ_{IO} in the following way:

Event in Σ_{IO}	**Semantics**
$\mathsf{in}(u, f, a, d)$	User u inputs d to argument a of function f
$\mathsf{out}(f, u, p, d)$	Function f outputs d to user u for purpose p
Event in Σ_{IF}	**Semantics**
$\mathsf{In}(u, f, i)$	User u inputs i to function f
$\mathsf{Out}(f, u, p, o)$	Function f outputs o to user u for purpose p
$\mathsf{Itf}(i, o)$	Input i interferes with ($\approx$ influences) output o

Figure 6: Signatures Σ_{IO} and Σ_{IF} (domain = $\mathbb{D}$)

- If the transition has label $\mathsf{i}\big(u, f, \{a_i \mapsto d_i\}_{1 \le i \le k}, \tau_i\big)$, then $E_i = \{\mathsf{in}(u, f, a_i, d_i) \mid 1 \le i \le k\}$;
- If it has label $\mathsf{o}(f, u, p, d, \tau_i)$, then $E_i = \{\mathsf{out}(f, u, p, d)\}$.

The complete input-output (IO) trace is obtained by considering the sequence of all such pairs (τ_i, E_i).

Example 3.1. The scenario described in Example 2.1 produces the following IO trace:

$((\tau_0, \{\mathsf{in}(\texttt{Alice}, \texttt{user\_timeline}, \texttt{username}, \texttt{Bob})\}),$
$(\tau_1, \{\mathsf{out}(\texttt{user\_timeline}, \texttt{trustedanalytics.com}, \texttt{Analytics}, \texttt{Bob})\}),$
$(\tau_2, \{\mathsf{out}(\texttt{user\_timeline}, \texttt{evilanalytics.com}, \texttt{Analytics}, \texttt{Bob})\}),$
$(\tau_3, \{\mathsf{out}(\texttt{user\_timeline}, \texttt{Alice}, \texttt{Service}, \textit{posts})\}),$
$(\tau_4, \{\mathsf{out}(\texttt{user\_timeline}, \texttt{Alice}, \texttt{Marketing}, \textit{ad})\})).$

In the following, we label the relation $\Rightarrow_{\pi,P}$ introduced in Section 2.2.2 with the IO trace produced by the execution. We write $S, I \overset{\sigma}{\underset{\pi,P}{\Longrightarrow}} S', I'$ if $S, I \Rightarrow S', I'$ and σ is the IO trace produced by the sequence of transitions from S to S'.

3.2 Information-flow traces

We now define *information-flow traces*, which will provide our ground truth on which to specify privacy policies. Our definition aims at (i) minimizing the amount of information to be stored in the trace to improve performance while preserving privacy and (ii) giving a precise meaning to what "data derived from an input" means. To address (i), we replace the values stored in the IO traces by unique input and output identifiers. To address (ii), we introduce a new Itf event based on a notion of (non)interference [48] capturing the influence of an input on an output. The signature is $\Sigma_{\text{IF}} = \big(\mathbb{D} \cup \mathbb{D}^2, \{\mathsf{In}, \mathsf{Out}, \mathsf{Itf}\}, \{\mathsf{In} \mapsto 3, \mathsf{Out} \mapsto 4, \mathsf{Itf} \mapsto 2\}\big)$ (cf. Figure 6).

3.2.1 Inputs and outputs. In information-flow traces, the In event encodes an input and the Out event an output. Since timestamps increase with every IO transition, each input can be uniquely identified by its timestamp τ and argument name a. We can use the pair $i = (\tau, a)$, which we call a *unique taint* (UT), as a unique input identifier. UTs allow us to refer to individual inputs specifically and are independent of the input's value. Similarly, each output can be uniquely identified by its timestamp. We obtain the following rules:

- Each $\mathsf{in}(u, f, a_i, d_i)$ in the IO trace at timestamp τ becomes $\mathsf{In}\,(u, f, (\tau, a_i))$ in the information-flow trace;
- Each $\mathsf{out}(f, u, p, d)$ in the IO trace at timestamp τ becomes $\mathsf{Out}\,(f, u, p, \tau)$ in the information-flow trace.

Example 3.2. In the scenario from Example 2.1, the information-flow trace contains the following In and Out events:

$((\tau_0, \{\mathsf{In}(\texttt{Alice}, \texttt{user\_timeline}, \texttt{username}, (\tau_0, \texttt{username}))\}),$
$(\tau_1, \{\mathsf{Out}(\texttt{user\_timeline}, \texttt{trustedanalytics.com}, \texttt{Analytics}, \tau_1), \dots\}),$
$(\tau_2, \{\mathsf{Out}(\texttt{user\_timeline}, \texttt{evilanalytics.com}, \texttt{Analytics}, \tau_2), \dots\}),$
$(\tau_3, \{\mathsf{Out}(\texttt{user\_timeline}, \texttt{Alice}, \texttt{Service}, \tau_3), \dots\}),$
$(\tau_4, \{\mathsf{Out}(\texttt{user\_timeline}, \texttt{Alice}, \texttt{Marketing}, \tau_4), \dots\})).$

where the dots are placeholders for the Itf events discussed next.

Note that a distinct a argument no longer shows up in In, since a becomes part of the identifier input $i = (\tau, a)$.

3.2.2 Interference. To reason about derived data, IO traces also contain events of the form $\mathsf{Itf}(i, o)$, where i is a UT and o an output identifier (i.e., an output timestamp). Informally, $\mathsf{Itf}(i, o)$ is present in the trace if, and only if, observing the value of the output with identifier o allows us to learn at least one bit of information about the value of the input with UT i. This definition of Itf captures (non)interference [48] in a fine-grained way, encoding the influence of a single input on a single output. In contrast, conventional IFC approaches group inputs according to users and, further, according to security levels.

Example 3.3. Assume that, in Example 2.1, the table `post` contains a single post by user Bob. The text of this post, an input from a previous function call, is tagged with a UT i_1. Consider first the application without proactive checks (Figure 4). At timestamp τ_0, `user_timeline` retrieves Bob's posts in the variable `posts`. Two inputs have influenced the content of `posts`: the one marked by i_1, and Alice's last input to `user_timeline` (with value `username` = "Bob"), which has some UT $(\tau_0, \texttt{username})$. Thus, the information-flow trace must contain $\mathsf{Itf}(i_1, \tau_3)$ and $\mathsf{Itf}((\tau_0, \texttt{username}), \tau_3)$ at timestamp τ_3. Further, `username` is sent to the two analytics third-parties, which receive the outputs with identifiers τ_1 and τ_2. Through `generate_ad`, the two inputs that influenced `posts` also influence `ad`, which is output with the identifier τ_4. We have the following information-flow trace:

$((\tau_0, \{\dots\}),$
$(\tau_1, \{\dots, \mathsf{Itf}((\tau_0, \texttt{username}), \tau_1))\},$
$(\tau_2, \{\dots, \mathsf{Itf}((\tau_0, \texttt{username}), \tau_2))\},$
$(\tau_3, \{\dots, \mathsf{Itf}(i_1, \tau_3), \mathsf{Itf}((\tau_0, \texttt{text}), \tau_3)\}),$
$(\tau_4, \{\dots, \mathsf{Itf}(i_1, \tau_4), \mathsf{Itf}((\tau_0, \texttt{text}), \tau_4)\}))$

where the dots represent the In and Out events from Example 3.2.

A formal definition of the relation "input $i = (\tau, a)$ interferes with output $o = \tau_1$ given the input sequence I," written $(\tau, a) \rightsquigarrow_{\pi,P,I} \tau_1$, is given in Appendix A.1. Using this relation, we can state the last rule defining our information-flow trace:

- Let I be an input sequence, π a program, and P a policy. Assume that we execute the application defined by $\rightarrow_{\pi,P}$ on I. Event $\mathsf{Itf}\,((\tau, a), \tau_1)$ is in the information-flow trace at timestamp τ_1 if, and only if, $(\tau, a) \rightsquigarrow_{\pi,P,I} \tau_1$.

We write $\text{trace}_{\pi,P}(I, \sigma)$ to state that the processing of all inputs in I using $\rightarrow_{\pi,P}$ generates the information-flow trace σ.

Our Itf event plays a role analogous to the *leak modality* of SecLTL [34] by capturing a *hyperproperty* [25]: the occurrence of the Itf event depends on the existence of an alternative execution

of the program in which a different input value leads to a different output value. In general, deciding whether such an alternative execution exists is impossible; hence, the information-flow trace cannot be fully computed at runtime. In Section 4, we will see how it can be soundly overapproximated using dynamic IFC techniques.

3.3 Privacy policies

We can finally define a *privacy policy* as a policy over information-flow traces, i.e., a subset of $\mathbb{T}_{\Sigma_{\text{IF}}}$. We say that the online application $A = (\mathbb{S}, S_0, \rightarrow_{\pi,P})$ enforces the privacy policy $P \subseteq \mathbb{T}_{\Sigma_{\text{IF}}}$ iff for any input sequence I and trace σ, we have $\text{trace}_{\pi,P}(I, \sigma) \in P$. In our application model, inputs are only-observable, while outputs are suppressable. Interference, which results from the program's behavior, can only be observed. Hence Obs = {In, Itf}, Sup = {Out}.

Our privacy policy definition fulfills [R1-3]:

[R1] Our architecture supports user-specified policies.
[R2] Restrictions can be defined down to the level of single inputs by specifying privacy policies that prohibit outputs involving interference from specific inputs identified by their UT.
[R3] Privacy policies can be temporal, since traces are.

Example 3.4. A way to specify a privacy policy as defined above is using a temporal logic. Metric First-Order Temporal Logic [9, 21] provides the temporal operators $\blacklozenge_I$ ('once in the past within time interval I') and $\Box$ (always) that can be used to formalize the policies in Figure 1. A full description of MFOTL and a formalization of the policies from Figure 1 in MFOTL can be found in Appendix C.

Discussion. In this section, we have shown how information-flow traces can be used to specify a general class of user-defined privacy policies. In general, the richness of this policy language and its explicit use of interference can challenge users. Hence the question naturally arises of how best to design appropriate interfaces for collecting user consent, and what concrete (high-level) policy language should be exposed to users in such interfaces. Interface and policy language design might involve standard techniques such as specification patterns [38], graphical representations [62], or natural language generation [79], and might benefit from a transdisciplinary effort. The policies collected through such interfaces could then be converted into an expressive formal language such as MFOTL. We see designing such interfaces and languages as an important, but distinct, problem. Hence, in the rest of this paper, we focus on the enforcement of policies expressed as MFOTL formulae, which provide the required expressivity and tools.

4 TAINT, TRACK, AND CONTROL

We now introduce our *Taint, Track, and Control* (TTC) approach. We first present a high-level overview of TTC (Section 4.1). Then, we proceed with a description of TTC's formal semantics: we define TTC programs (Section 4.2), introduce a new data structure called *tagged values* (Section 4.3), describe the state space on which TTC applications operate (Section 4.4), and then give the semantic rules for each of the Taint, Track, and Control steps (Sections 4.5–4.7). Finally, we establish the guarantees that TTC provides (Section 4.8).

4.1 TTC: a high-level overview

At its core, TTC consists of (a) an architecture for privacy enforcement and (b) an enforcement strategy carried out in three phases, called Taint, Track, and Control respectively.

The TTC architecture has the components shown in Figure 2: an enforcer serving as a PDP, an interpreter, a wrapper orchestrating the interaction between the interpreter, enforcer, and users, and serving as a PEP, and a persistent storage module.

Enforcement in this architecture is structured in three phases:

Taint: A set of user inputs is received by the wrapper. Each user input is tagged with a fresh UT and the corresponding In events are forwarded to the enforcer. Then, the interpreter is loaded with the code of the function called by the user, and passed the (tagged) user inputs.

Track: This code is executed by the interpreter, updating the content of permanent storage and computing a tentative output value. For every data item in both working memory and permanent storage, the interpreter additionally maintains a *UT history* (see next section) that conservatively approximates the set of user inputs that have influenced its current value. At the end of the Track phase, the interpreter returns an output value to the wrapper.

Control: The wrapper computes (an overapproximation of) the Out and Itf events corresponding to the new tentative output. The UT history of the output is used to identify a superset of all inputs that interfered with the output, and hence generate the right Itf events. These events are forwarded to the enforcer, which responds with a verdict. Finally, the output is emitted if and only if the enforcer's verdict is empty, i.e., if and only if the new output complies with the policy. After a Control phase, the system can either start a new Track phase or return to the Taint phase to process the next set of arguments. The UT histories of data in permanent storage are persisted, thereby tracking the influence of inputs over their entire lifetime.

Example 4.1. Let us revisit Example 2.1 in the TTC context. In the Taint phase, the wrapper receives the input `username` $\mapsto$ `Bob` from user `Alice` querying function `user_timeline`. It tags it with the UT $i = (\tau_0, \texttt{username})$ and sends the corresponding In event to the enforcer. It then forwards the tagged input to the interpreter. In the Track phase, the interpreter computes the first output (to be sent to `trustedanalytics.com`), which it sends back to the wrapper. Then, in the Control phase, the wrapper generates the corresponding Out and Itf events, which it sends to the enforcer. Depending on the enforcer's verdict, the corresponding output is (or is not) forwarded to `trustedanalytics.com`. After the end of the Control phase, a new Track phase starts, this time producing the second output (to `evilanalytics.com`). This Track phase is followed by a new Control phase handling the output to `evilanalytics.com`. Two other iterations follow to generate and handle the two outputs to `Alice`. After the last output has been produced, a new Taint phase can take place to process the next user input.

In practice, developers should properly handle data that they cannot output, rather than just have outputs be suppressed. This requires *proactive checks* that allow programs to obtain enforcer verdicts ahead of time and react to them appropriately. With proactive checks, developers can, e.g., remove data items from a page being built if these items cannot be displayed according to the current policy. To support such proactive checks, the interpreter must be able to perform direct queries to the enforcer in the Track phase.

Example 4.2. A version of the code from Figure 4 with proactive checks is shown in Figure 5. Proactive enforcer calls are performed via the primitive `check`. If the content of `posts` cannot be used for marketing, non-personalized ads are generated (l. 11). The function `filter_check` (l. 1-5) filters out messages that cannot be shown to the caller user for service purposes; it is used (l. 10) to remove all posts that the caller is not allowed to see from the page. These operations occur during the Track phase.

4.2 TTC programs

A TTC program is a mapping $f \to \pi(f)$ from function names to non-empty, finite sequences of tuples of the form

$$\pi(f) = ((\mathtt{code}_i, \mathtt{out\_y}_i, \mathtt{out\_u}_i, \mathtt{out\_p}_i, \mathtt{time}_i))_{1 \le i \le k}.$$

Given a function name f, the tuple sequence $\pi(f)$ specifies the behavior of f. More precisely, each tuple in the sequence encodes a block of non-IO source code $\mathtt{code}_i$ followed by a single output statement that attempts to output the content of variable $\mathtt{out\_y}_i$ to user $\mathtt{out\_u}_i$ for purpose $\mathtt{out\_p}_i$. The complete source code of the function is obtained by concatenating the $(\mathtt{code}_i)_{1 \le i \le k}$ and their respective output statements. Each $\mathtt{out\_u}_i$ can take any value in $\mathbb{U}$ or the special value `me` denoting the user that performed the last input. The integer $\mathtt{time}_i \in \mathbb{N}$ is the fixed duration of executing $\mathtt{code}_i$. In the following, the various components of $\pi(f)_i$ will be denoted by $\pi(f)_i.\mathtt{code}, \ldots, \pi(f)_i.\mathtt{time}$ respectively. Note that this definition fits into the time-insensitive framework described in Section 2.3 above.

Example 4.3. The function `user_timeline` in Figure 4 can be decomposed as follows, where ε denotes an empty source code:

$$\begin{aligned}\pi(f) = ((&\langle\text{code l. 9–15}\rangle, \mathtt{username}, \mathtt{trustedanalytics.com}, \mathtt{Analytics}),\\ &(\varepsilon, \mathtt{username}, \mathtt{evilanalytics.com}, \mathtt{Analytics}, \mathtt{Continue}\ f_3),\\ &(\varepsilon, \mathtt{posts}, \mathtt{me}, \mathtt{Service}),\\ &(\varepsilon, \mathtt{ad}, \mathtt{me}, \mathtt{Marketing})).\end{aligned}$$

To simplify the presentation, we consider non-IO source code written in the following Turing-complete language TTCWhile, that manipulates integer values. We fix a set $\mathbb{O}$ of binary operator symbols and a mapping $\Omega : \mathbb{O} \to (\mathbb{D} \times \mathbb{D} \to \mathbb{D})$ interpreting these symbols. For illustrative purposes, we assume that $\mathbb{O}$ contains at least two symbols == and + such that $\Omega(==) = (x, y) \mapsto$ if $x = y$ then 1 else 0 and $\Omega(+) = (x, y) \mapsto$ if $\{x, y\} \subseteq \mathbb{N}$ then $x + y$ else 0.

The following is the grammar of TTCWhile source code:

$$\begin{aligned} e &:= c \mid x \mid e \oplus e\\ s &:= x = e \mid \mathtt{while}\ x\ \{b\} \mid x = \mathtt{check}\ y\ [u|\mathtt{me}]\ p\\ b &:= s \mid b; b \mid \varepsilon \end{aligned}$$

where $c \in \mathbb{D}$ is a constant, $\oplus \in \mathbb{O}$ a binary operator symbol, x a variable name, u a user name, and p a purpose.

The function source code, represented by the non-terminal b ("block") is a (possibly empty) sequence of statements. Statements include assignments of expressions ($x = e$), *while* loops ($\mathtt{while}\ x\ \{b\}$), and the special `check` instruction ($x = \mathtt{check}\ y\ u\ p$). Expressions can be constructed from constants (c), variables (x), and binary operators ($e \oplus e$). The instruction $x = \mathtt{check}\ y\ u\ p$ puts 1 into x if the content of a variable y can be output to some user u for some purpose p, and 0 otherwise; the user that performed the last input can be referred to as `me`. From assignments and *while* loops, we define *if* statements as syntactic sugar, writing $\mathtt{if}\ x\ \{b\}$ as shorthand for $\mathtt{t} = x; \mathtt{while}\ \mathtt{t}\ \{b; \mathtt{t} = 0\}$.

For the rest of this section, we fix a TTCWhile program π, an enforceable policy $P \subseteq \mathbb{T}_{\Sigma_{\mathrm{IF}}}$, and choose an enforcer $\mathcal{E}$ for P.

4.3 Tagged values

Tagged values are a new data structure designed to support fine-grained information-flow tracking and proactive checks.

A tagged value is a pair $w = \langle v, \alpha\rangle$ of a value $v \in \mathbb{D}$ and a finite list $\alpha = [\{\ell_{11}, \ldots, \ell_{1k_1}\}, \ldots, \{\ell_{L1}, \ldots, \ell_{Lk_L}\}]$ of finite sets of UTs. The list α is called a *UT history*. The UT history α represents all inputs that have influenced the value v, as well as dependencies between these influences. More specifically, whenever a variable `x` contains w as above, we require the following properties to hold:

(1) The value v may have been influenced only by those inputs whose UT is one of the ℓ_{ij};
(2) The UTs in the first set, i.e., $\{\ell_{11}, \ldots, \ell_{1k_1}\}$, are part of the history of `x` independently of any input;
(3) For any $i > 1$, the content of the ith set of the history, $\{\ell_{i1}, \ldots, \ell_{ik_i}\}$, has been influenced only by those inputs whose UTs are in the $i - 1$ previous sets.

Example 4.4. Consider a variable `x` containing the tagged value $\langle 1, [\{\ell_1\}, \{\ell_2, \ell_3\}]\rangle$. The value of `x` is 1. The history indicates that:

(1) Inputs with UTs ℓ_1, ℓ_2, and ℓ_3 may have influenced `x`'s value;
(2) No input has influenced that ℓ_1 tags `x`;
(3) The input with UT ℓ_1 has influenced that ℓ_2 and ℓ_3 tag `x`.

In practice, such a history can be obtained, e.g., by executing the code `x = 10; if y {x = u + v}` when `y`, `u`, and `v` contain inputs $\langle 1, [\{\ell_1\}]\rangle$, $\langle 0, [\{\ell_2\}]\rangle$, and $\langle 1, [\{\ell_3\}]\rangle$ respectively. In this case:

(1) The values of the inputs `y`, `u`, `v` with UTs ℓ_1, ℓ_2, ℓ_3 respectively have all influenced the final value of `x`;
(2) The value of the input `y` with UT ℓ_1 has influenced the final value of `x` irrespective of the value of any other input;
(3) The value of the input `y` with UT ℓ_1 has influenced that `u` and `v` (with UTs ℓ_2 and ℓ_3) influence `x`: if the `if` block is executed, there is influence of `u` and `v` on `x`, whereas there is no such influence if the `if` block is not executed.

When property (1) above holds, the Itf events resulting from outputting v must all be of the form $\mathrm{Itf}(\ell_{ij}, o)$ for some ℓ_{ij} in α. This allows for a sound overapproximation of Itf events at runtime.

Unlike conventional dynamic IFC techniques based on tagging data with labels from a fixed security lattice, our notion of tagged values allows for arbitrarily fine-grained, per-input information-flow tracking. The use of UT histories fulfilling properties (1)-(3), rather than unordered UT sets that can fulfill only (1), is motivated by the need to perform proactive checks while still keeping track of

information flows. As we will demonstrate in Section 4.6.2 below, UT histories allow for such checks to be performed on outputs that combine several inputs by providing an *order* in which individual UTs can be checked.

4.4 State space

TTC applications operate on a state space $\mathbb{S}_{\text{TTC}} = \mathbb{S}_W \times \mathbb{S}_I \times \mathbb{S}_E \times \mathbb{S}_M$, where $\mathbb{S}_W$, $\mathbb{S}_I$, $\mathbb{S}_E$, and $\mathbb{S}_M$ are the state spaces of the wrapper, the interpreter, the enforcer, and persistent storage (also called *memory*), respectively. The states of the wrapper and interpreter are represented as record types; their structure will be introduced in the next sections. The enforcer state stores a *trace approximation*. A trace approximation is a trace $\sigma \in \mathbb{T}_{\Sigma_{\text{IF}}} = \mathbb{S}_E$ that contains *at least as many* ltf *events as the actual information-flow trace of the system*. Finally, the memory state is a mapping from variable names to tagged values.

4.5 Taint

The semantics of the Taint phase is given by the rule

$$\frac{\tau' = \tau + \pi(f)_1.\texttt{time} \qquad s'_E = s_E \cdot (\tau, \{\text{In}(u, f, (\tau, a_i))\}_{1 \le i \le k})}{\begin{array}{l}\{\texttt{step} = \textsc{Taint}\}, s_I, s_E, s_M \xrightarrow{\text{i}(u,f,\{a_i \mapsto d_i\}_{1\le i\le k},\tau)}_{\pi,P} \\ \{\texttt{step} = \textsc{Track}, \texttt{u} = u, \texttt{f} = f, \texttt{t} = \tau', \texttt{i} = 1\}, \\ \text{init}_I(f, u, s'_E, \tau', 1), s'_E, s_M(a_i := \langle d_i, [\{(\tau, a_i)\}]\rangle)\end{array}}\ \textsc{taint}$$

This uses the three auxiliary functions

$$\text{init}_I(f, u, \sigma, \tau', i) = \{\texttt{code} = \pi(f)_i.\texttt{code}, \texttt{pc} = [], \texttt{u\_me} = u, \texttt{c} = c^{\mathcal{E}}_{f,u,\sigma,\tau'}\},$$
$$c^{\mathcal{E}}_{f,u,\sigma,\tau'}(\ell, u', p) = \left(\mathcal{E}\left(\sigma, \left(\tau', \{\text{Out}\left(f, \text{usr}(u, u'), p, \tau'\right), \text{ltf}\left(\ell, \tau'\right)\}\right)\right) = \emptyset\right)$$
$$\text{usr}(u, u') = \texttt{if } u' = \texttt{me } \texttt{then } u \texttt{ else } u'.$$

This Taint rule can be performed only when the wrapper state has its `step` parameter set to Taint. First (Step 1 in Figure 2, highlighted in yellow), the next set of inputs is retrieved. The caller user u, the called function f, and the input timestamp τ are read in the wrapper's state. Moreover, the timestamp τ is incremented by $\pi(f)_1.\texttt{time}$ to obtain the timestamp $\tau' = \tau + \pi(f)_1.\texttt{time}$ of the next tentative output, and the field `i`, which encodes the index of the tuple of $\pi(f)$ currently being executed, is set to 1. Second (Step 2 in Figure 2), the trace approximation is updated with the In events corresponding to the last input. Third (Step 3 in Figure 2), the interpreter's state is initialized using the auxiliary function init_I: its field `code` receives the source code $\pi(f)_1.\texttt{code}$ of f; its *program counter history*, discussed in the next section, is initialized to []; the constant `u_me` is set to u; finally, the single-UT enforcement oracle `c`, which will be used to execute `check` instructions in the Track phase, is set to $c^{\mathcal{E}}_{f,u,\sigma,\tau}$. The function `c` takes as arguments a UT ℓ, a user u, and a purpose p; it returns 1 if one can perform an output to u for purpose p at timestamp τ' with a value influenced by ℓ without violating P. To detect violations, `c` calls the enforcer $\mathcal{E}$. The auxiliary function usr makes it possible to refer to the caller user u in `c` using the special constant `me`. Finally, the arguments a_i are set in memory, each tagged with a fresh UT (τ, a_i).

$$\frac{c \in \mathbb{D}}{[\![c]\!](m) = \langle c, \emptyset\rangle}\ \textsc{econst} \qquad \frac{}{[\![y]\!](m) = m(y)}\ \textsc{evar}$$

$$\frac{[\![e_1]\!] = \langle v_1, \alpha_1\rangle \quad [\![e_2]\!] = \langle v_2, \alpha_2\rangle}{[\![e_1 \oplus e_2]\!](m) = \langle \Omega(\oplus)(v_1, v_2), \alpha_1 \uplus \alpha_2\rangle}\ \textsc{ebinop}$$

(a) Evaluation of values

$$\frac{[\![e]\!](m) = \langle v, \alpha\rangle}{x = e; \pi, pc, m \dashrightarrow \pi, pc, m(x := \langle v, \text{all}(pc) \cdot \alpha\rangle)}\ \textsc{assign}$$

$$\frac{m(x) = \langle v, \alpha\rangle \quad v \neq 0}{\begin{array}{l}\texttt{while } x \{b\}; \pi, pc, m \dashrightarrow \\ b; \texttt{while } x \{b\}; \texttt{pop}; \pi, [\alpha] \cdot pc, \text{sanitize}(m, b, \alpha)\end{array}}\ \textsc{while\_true}$$

$$\frac{m(x) = \langle 0, \alpha\rangle}{\texttt{while } x \{b\}, pc, m \dashrightarrow \texttt{pop}; \pi, [\alpha] \cdot pc, \text{sanitize}(m, b, \alpha)}\ \textsc{while\_false}$$

$$\frac{}{\texttt{pop}; \pi, [s] \cdot pc, m \dashrightarrow \pi, pc, m}\ \textsc{pop}$$

$$\frac{m(y) = \langle v, \alpha\rangle \quad \kappa(u, p, \alpha) = \langle v', \alpha'\rangle}{x = \texttt{check } y\ u\ p; \pi, pc, m \dashrightarrow \pi, pc, m(x := \langle v', \text{all}(pc) \cdot \alpha'\rangle)}\ \textsc{check}$$

(b) Semantic rules

$$\text{lhs}(b) = \begin{cases} \emptyset & \text{if } b = \varepsilon \\ \{x\} \cup \text{lhs}(b') & \text{if } b = x = e; b' \\ \{x\} \cup \text{lhs}(b') & \text{if } b = x = \texttt{check } y\ u\ p; b' \\ \text{lhs}(b') \cup \text{lhs}(b'') & \text{if } b = \texttt{while } x \{b'\}; b'' \end{cases}$$
$$\text{sanitize}(m, b, \alpha) = m(\forall v \in \text{lhs}(b).\ v := \langle d, \alpha \cdot \beta\rangle \text{ where } \langle d, \beta\rangle = m(v))$$
$$\text{all}([\alpha_1, \ldots, \alpha_k]) = \alpha_k \cdot \ldots \cdot \alpha_1$$
$$\kappa(u, p, [s_1, \ldots, s_k]) = \begin{cases} \langle 1, [\,]\rangle \text{ if } k = 0 \\ \langle 0, [\,]\rangle \text{ if } \exists \ell \in s_1.\ \text{c}(u, p, \ell) \neq 1 \\ \langle v, s_1 \cdot \beta\rangle \text{ where } \langle v, \beta\rangle = \kappa(u, p, [s_2, \ldots, s_k]) \\ \quad \text{if } \forall \ell \in s_1.\ \text{c}(u, p, \ell) = 1. \end{cases}$$

(c) Auxiliary functions

Figure 7: Small-step semantics of TTCWhile

4.6 Track

In the Track step, the interpreter executes the code stored in its code field. We first present the semantics of assignments and loops (Section 4.6.1), then the semantics of checks (Sections 4.6.2), and finally state the track rule itself (Section 4.6.3).

4.6.1 TTCWhile semantics. The small-step semantics of TTCWhile is shown in Figure 7. The five rules assign, while_true, while_false, pop, and check (Figure 7b) are of the form $\pi, pc, m \dashrightarrow \pi', pc', m'$, where π is a source code, pc is a program counter history, and m is a memory state. The program counter history is a list of UT histories. It keeps track of all inputs that have influenced the control flow: when entering a *while* loop, the UT history of the variable in the loop branches is added to pc; when leaving the loop, this history is removed from pc. Three rules econst, evar, and ebinop of the form $[\![v]\!](m) = \langle w, \alpha\rangle$ (Figure 7a) evaluate expressions to tagged values over a given memory state m.

The evaluation rules econst and evar are straightforward. Rule ebinop evaluates expressions of the form $e_1 \oplus e_2$. For this, the two subexpressions are recursively evaluated to $[\![e_1]\!](m) = \langle v_1, \alpha_1\rangle$ and $[\![e_2]\!](m) = \langle v_2, \alpha_2\rangle$; then, a tagged value with value $\Omega(\oplus)(v_1, v_2)$

and UT history $\alpha_1 \Cup \alpha_2$ is computed, where $\Cup$ is defined by

$$[s_1, ..., s_L] \Cup [t_1, ..., t_M] = \begin{cases} [s_1 \cup t_1, ..., s_L \cup t_L, t_{L+1}, ..., t_M] \text{ if } M \geq L \\ [s_1 \cup t_1, ..., s_M \cup t_M, s_{M+1}, ..., s_L] \text{ if } L \geq M. \end{cases}$$

By this definition, the UT history of $[\![e_1 \oplus e_2]\!](m)$ contains all UTs in the histories of $[\![e_1]\!](m)$ and $[\![e_2]\!](m)$. This reflects that the inputs that have influenced the evaluation $e_1 \oplus e_2$ are those that have influenced the evaluation of e_1 or e_2. Moreover, the relative order of the UTs within α_1 and α_2 respectively is preserved in $\alpha_1 \Cup \alpha_2$.

The ASSIGN rule defines the semantics of assignments $x = e$. First, $[\![e]\!]$ is evaluated to some $\langle v, \alpha \rangle$. Then, $m(x)$ receives a new tagged value with value v and UT history $\mathsf{all}(pc) \cdot \alpha$, where $\mathsf{all}(pc)$ concatenates of all UT histories in pc (see Figure 7c). Thus the new UT history of $m(x)$ contains all UTs in the history of $[\![e]\!](m)$, as well as all UTs in the program counter history. This reflects that the inputs that influence the value of $m(x)$ are those that influenced the result of evaluating v or the control flow. Prepending of $\mathsf{all}(pc)$ to α records that the presence of the UTs from α in $m(x)$ depends on the inputs with UTs in $\mathsf{all}(pc)$, which have influenced the control flow.

The WHILE_TRUE and WHILE_FALSE rules define the semantics of loops `while` x $\{b\}$. If $m(x) = \langle v, \alpha \rangle$ and $v \neq 0$, then WHILE_TRUE is applicable. This rule adds b at the front of the code to be executed and prepends α to pc, recording that the inputs corresponding to the UTs in α now influence the control flow. To cover any *implicit flows* that the loop's execution can cause, WHILE_TRUE additionally adds the UTs α to $m(v)$ for all variables v that can be modified by b. The set of such variables is computed using the auxiliary function lhs (see Figure 7c). Finally, a special pop instruction is inserted after `while`. Its semantics is defined by the POP rule and it removes the top element of the program counter history after the *while* loop completes. If $v = 0$, the rule WHILE_FALSE is applied instead. In contrast to WHILE_TRUE, it does not add b to the code to be executed.

Example 4.5. Let k_0 = `while x {z = a + b; x = 0}`. Consider the initial memory state $m_0 = \{\mathsf{a} \mapsto \langle 3, [\{\ell_1\}] \rangle, \mathsf{b} \mapsto \langle 2, [\{\ell_2\}] \rangle, \mathsf{x} \mapsto \langle 1, [\{\ell_3\}] \rangle, \mathsf{z} \mapsto \langle 0, [] \rangle \dots\}$. Executing k_0 on m_0 yields:

$k_0, [], m_0 \dashrightarrow$ `z = a + b;` $\dots, [\{\ell_3\}], m_0(\mathsf{z} := \langle 0, [\{\ell_3\}] \rangle)$ [WHILE_TRUE]
$\dashrightarrow$ `x = 0;` $\dots, [\{\ell_3\}], m_0(\mathsf{z} := \langle 5, [\{\ell_3\}, \{\ell_1, \ell_2\}] \rangle) =: m_1$ [ASSIGN]
$\dashrightarrow$ `while x ...; pop; pop`, $[\{\ell_3\}], m_1(\mathsf{x} := \langle 0, [\{\ell_3\}] \rangle)$ [ASSIGN]
$\dashrightarrow$ `pop; pop`, $[\{\ell_3\}, \{\ell_3\}], m_1(\mathsf{x} := \langle 0, [\{\ell_3\}] \rangle)$ [WHILE_FALSE]
$\dashrightarrow$ `pop`, $[\{\ell_3\}], m_1(\mathsf{x} := \langle 0, [\{\ell_3\}] \rangle)$ [POP]
$\dashrightarrow [], [], m_1(\mathsf{x} := \langle 0, [\{\ell_3\}] \rangle)$ [POP].

The formal noninterference property satisfied by this semantics is given in Appendix A (Lemma A.25).

4.6.2 Checks. The CHECK rule in Figure 7b defines the semantics of x = check y u p. It uses the function κ in Figure 7c, calling it with the arguments (u, p, α) where α is the UT history of $m(x)$. The function κ is defined so that this call returns 1 if, and only if, $\mathsf{c}(u, p, \ell) = 1$ for every UT ℓ contained in α, where c is the single-UT enforcement oracle defined in the previous section. In other words, $\kappa(u, p, \alpha) = 1$ if all inputs that have influenced y can be output to u for purpose p. This matches the intuitive semantics of check.

While defining the *value* of $\kappa(u, p, \alpha)$ using $\mathsf{c}(u, p, \ell)$ is easy, choosing the right definition for the *UT history* of $\kappa(u, p, \alpha)$ is more difficult. This is because this definition must both (a) account for all information flows between user inputs and the $\mathsf{c}(u, p, \ell)$ and (b) ensure that the result of the check can still be branched on to proactively react to (especially negative) enforcer verdicts.

Regarding (a): In general, the output value of κ can be influenced by user inputs in such a way that concrete information about inputs *can* be learnt. An example of this is given in Appendix A.2.

Regarding (b): We expect check to be used to proactively check if an output is allowed: typically, after executing x = check y u p, an *if* branch will be used to generate different output values depending on whether y can be output or not. But if x is tagged by any UT for which $c(u, p, \cdot) = 0$, then no output value computed within the *if* branch may be output, removing the practical benefit expected from the check primitive. Therefore, we do not want $\kappa(u, p, \alpha)$ to be tagged by any UTs for which $\mathsf{c}(u, p, \cdot)$ returns 0. Formally:

$$\forall p, \alpha, v, \beta, \ell.\ \kappa(u, p, \alpha) = \langle v, \beta \rangle \wedge \ell \in \mathsf{uts}(\beta) \Rightarrow \mathsf{c}(u, p, \ell) = 1$$

where $\mathsf{uts}([s_1, \dots, s_k]) = s_1 \cup \dots \cup s_k$.

In Appendix A.2, we show that the following algorithm, together with our definition of UT histories, defines a $\kappa(u, p, \alpha)$ that fulfills the constraints expressed in both (a) and (b):

- Call $\mathsf{c}(u, p, \ell)$ for every UT ℓ in α in the order of the history;
- If any call returns 0, then return 0, otherwise return 1;
- Tag the return value with all UTs for which c returned 1 and that are not in the last set of the history, in the same order as in the original history.

The definition of κ in Figure 7c formalizes this.

Example 4.6. Consider again the tagged value $\langle 1, [\{\ell_1\}, \{\ell_2, \ell_3\}] \rangle$. Fix some u_0 and p_0, and let $\alpha = [\{\ell_1\}, \{\ell_2, \ell_3\}]$.

If $\mathsf{c}(u_0, p_0, \ell_1) = 0$, then we have $\kappa(u_0, p_0, \alpha) = \langle 0, [] \rangle$: the first check performed is on ℓ_1, which immediately fails. By invariant (2) of UT histories (see Section 4.3), we know that the presence of ℓ_1 in the first set of α has not been influenced by the value of any input. Hence, the output of $\kappa(u_0, p_0, \alpha)$ does not depend on the value of any input and the empty UT history in $\langle 0, [] \rangle$ correctly accounts for all information flows (a). Moreover, since the history is empty, condition (b) is trivially fulfilled.

If $\mathsf{c}(u_0, p_0, \ell_1) = 1$, then $\kappa(u_0, p_0, \alpha) = \langle (\mathsf{c}(u_0, p_0, \ell_2) = 1) \wedge (\mathsf{c}(u_0, p_0, \ell_3) = 1), [\{\ell_1\}] \rangle$: the first call of $\mathsf{c}(u_0, p_0, \cdot)$ on ℓ_1 is successful, hence the value of $\kappa(u_0, p_0, \alpha)$ depends on the output of $\mathsf{c}(u_0, p_0, \cdot)$ on ℓ_2 and ℓ_3. The content of $\kappa(u_0, p_0, \alpha)$ now depends on the value of the input with UT ℓ_1 (since this value influences the presence of ℓ_2 and ℓ_3 in the history), which is indicated by having ℓ_1 in the history $[\{\ell_1\}]$ (a). But since $\mathsf{c}(u_0, p_0, \ell_1) = 1$, (b) is also fulfilled.

Note that the algorithm we described requires an order on the set of UTs tagging a given value in memory that reflects the dependencies between the influences of the various inputs. This motivates the introduction of UT histories.

4.6.3 TRACK rule. The rule for the TRACK phase is

$$\frac{s_I.\mathsf{code}, [], s_M \dashrightarrow^* \varepsilon, pc', s'_M}{\{\mathsf{step} = \textsc{Track}, \mathsf{u} = u, \mathsf{f} = f, \mathsf{t} = \tau, \mathsf{i} = i\}, s_I, s_E, s_M \xrightarrow{\bullet}_{\pi,P} \{\mathsf{step} = \textsc{Control}, \mathsf{u} = u, \mathsf{f} = f, \mathsf{t} = \tau, \mathsf{i} = i\}, s_I, s_E, s'_M}\ \textsc{Track},$$

where $\dashrightarrow^*$ denotes the transitive closure of $\dashrightarrow$. After executing the code in $s_I.\mathsf{code}$ (1), the wrapper moves the system into a CONTROL state using a $\bullet$-transition, updating the memory state (2).

4.7 Control

The rule defining the semantics of the Control phase is

$$\frac{u' = \text{if } \pi(f)_i.\text{out\_u} = \text{me then } u \text{ else } \pi(f)_i.\text{out\_u} \quad p = \pi(f)_i.\text{out\_p} \quad \langle v, \alpha\rangle = m(\pi(f)_i.\text{out\_y}) \quad b = \mathcal{E}(s_E, (\tau, N(f,u',p,\tau,\alpha))) \quad s'_E = s_E \cdot (\tau, N(f,u',p,\tau,\alpha))}{\{\text{step} = \textsc{Control}, \text{u} = u, \text{f} = f, \text{t} = \tau, \text{i} = i\}, \; s_I, s_E, s_M \xrightarrow{\text{if } b=\emptyset \text{ then o}(f,u',p,v,\tau) \text{ else } \bullet}_{\pi,P} \; \text{next}_W(f,u,\tau,i), \; \text{next}_I(f,u,s'_E,\tau,i), \; (\text{if } b = \emptyset \text{ then } s'_E \text{ else } s_E \cdot (\tau,\emptyset)), \; s_M.} \; \textsc{ctrl}$$

where $N(f,u,p,\tau,\alpha) = \{\text{Out}(f,u,p,\tau)\} \cup \{\text{Itf}(\ell,\tau) \mid \ell \in \text{uts}(\alpha)\}$

$$\text{next}_W(f,u,\tau,i) = \begin{cases} \{\text{step} = \textsc{Taint}\} & \text{if } i = |\pi(f)| \\ \{\text{step} = \textsc{Track}, \text{u} = u, \text{f} = f, \\ \quad \text{t} = \tau + \pi(f)_{i+1}.\text{time}, \text{i} = i+1\} & \text{otherwise} \end{cases}$$

$$\text{next}_I(f,u,\sigma,\tau,i) = \begin{cases} \{\} & \text{if } i = |\pi(f)| \\ \text{init}_I(f,u,\sigma,\tau+\pi(f)_{i+1}.\text{time}, i+1) & \text{otherwise.} \end{cases}$$

First, the recipient user u', the purpose p, and the output value w of the tentative output defined by the program are retrieved (Step 1 in Figure 2). Second, an overapproximation $N(f,u',p,\tau,\alpha)$ of the set of events caused by the output is computed, and the enforcer's verdict is obtained (Steps 2-3 in Figure 2). Third, depending on whether $b = \emptyset$ (output complies with P) or $b \neq \emptyset$ (output violates P), the output is either performed or the suppressed, with the trace approximation updated accordingly (Step 4 in Figure 2). Finally, the system returns to a Track state incrementing i if $i < |\pi(f)|$, or to a Taint state if $i = |\pi(f)|$; the corresponding updates in the state of the wrapper and interpreter use the auxiliary functions next_W and next_I (Step 5, not shown in Figure 2).

4.8 Guarantees

We now state and discuss sufficient conditions on P that guarantee the correct enforcement of the policy P by any application implementing our TTC semantics. These conditions are (1) Itf-monotonicity and (2) independence of past outputs.

(1) We say that a policy P is a-monotonic for some action $a \in \mathbb{A}$ iff, for any trace σ that violates P, adding additional a events in σ cannot restore compliance with P. Formally, P is a-monotonic if for all $\sigma = ((\sigma_i, D_i))_i$ and $\sigma' = ((\sigma'_i, D'_i))_i$, we have

$$\left(\forall i.\ D_i \subseteq D'_i \wedge D'_i - D_i \subseteq \left\{a(\overline{d}) \mid \overline{d} \in \mathbb{D}^{\iota(a)}\right\}\right) \wedge \sigma \notin P \Rightarrow \sigma' \notin P.$$

Example 4.7. The policies in Figure 1 are Itf-monotonic, as they are of the form $\Phi_\psi = \Box\,[\forall f,u,p,o,i.\ \text{Out}(f,u,p,o) \wedge \text{Itf}(i,o) \Rightarrow \psi]$, where ψ contains no Itf event.

In general, TTC cannot enforce policies that are non-Itf-monotonic, since UTs *overapproximate* information flows.

(2) We say that a policy P is *independent of past outputs* if compliance with P does only dependent on past In events, but not on past Out and Itf events. If we allow for malicious developers and users, policies that depend on past outputs cannot, in general, be enforced using TTC. This is shown in Appendix B. To support such policies, one would need to adopt a coarser propagation of UTs in the presence of implicit flows, or to store the UTs of all inputs that affected the content of the trace approximation. Both approaches risk an explosion in the number of UTs to be stored ('label creep'). Hence, in this paper, we focus on policies that are independent of past outputs.

Example 4.8. All policies in Figure 1 are independent of past outputs, since only In events appear below temporal operators.

Theorem 4.9 (Enforcement). *Let*

$$S_{0,\text{TTC}} = \{\text{step} = \textsc{Taint}\}, \{\}, (), (x \mapsto \langle 0, []\rangle).$$

If P is enforceable, Itf-monotonic, and independent of past outputs, and the online application $A_{\text{TTC}} = (\mathbb{S}_{\text{TTC}}, S_{0,\text{TTC}}, \rightarrow_{\pi,P})$ terminates, then A_{TTC} enforces the privacy policy $P \subseteq \mathbb{T}_{\Sigma_{\text{IF}}}$.

Finally, we discuss the usefulness of the check instruction for different classes of policies. The check instruction enables the program to obtain the enforcer's verdict in advance for individual inputs or data items. For instance, in the code from Figure 4, the HTML representations of a sequence of events are incrementally generated and added to the page being only when check succeeds. The programmer's intent is that by doing so, she can ensure that the page sent to the user will be accepted by the enforcer. However, this is the case if the enforcer's verdict on a UT history $\alpha \Cup \beta$ is no stricter than the conjunction of its verdicts on α and β respectively. Otherwise, the output can fail despite the checks succeeding; this would not affect the application's security but may reduce its usability. A policy of the form Φ_ψ, however, always fulfills the above condition, and can therefore be chosen as a general form of policies amenable to both enforcement and checking.

Formal proofs. All definitions and theorems from Sections 2–4 have been machine-checked using the Isabelle/HL proof assistant. The formalization, which has approximately 3,500 lines of Isar code, is openly available [3]. More details are provided in Appendix A.

```
prog ::= fun; ... ; fun                                          (1)
fun ::= @route(str)
        def id(id, ..., id) :                                    (2)
          block
block ::= stmt
          ...                                                    (3)
          stmt
stmt ::= lhs = exp                                               (4)
       | del lhs[exp]                                            (5)
       | return exp
       | if exp:
           block
         [[else:
           block]]                                               (6)
       | while exp:
           block
       | for id in exp:                                          (7)
           block
lhs ::= id[[[exp]]]*                                             (8)
exp ::= const | id | exp[exp]                                    (9)
      | Ω1 exp | exp Ω2 exp | id(exp, ..., exp)                  (10)
      | {exp:exp, ..., exp:exp}                                  (11)
      | [exp, ..., exp] | {exp, ..., exp}                        (12)
      | len(exp) | keys(exp)                                     (13)
      | sql(query,exp) | render(exp,exp)                        (14)
      | redirect(exp) | get_session(const)                      (15)
      | set_session(const, exp)                                  (16)
      | get(const) | post(const)                                (17)
      | send(const, const, exp)                                  (18)
      | check(exp,const[[,exp]]) | me()                         (19)
```

Figure 8: Syntax of PythonTTC programs

5 IMPLEMENTATION

To show how our approach can be used, we have implemented WebTTC, a web programming framework with TTC semantics.

WebTTC's main component is a Python-like language called PythonTTC, which extends TTCWhile with support for basic web programming primitives. PythonTTC's syntax is presented in Figure 8. Figures 4 and 5 show examples of PythonTTC code. Beyond standard Python features (functions, dictionaries, lists, sets, l. 1–8

Table 1: Size of code base for each application

	Flask/Python			Jacqueline				WebTTC		
Application	View*	Model	Template	View*	Model	Security	Template	View*	Security	Template
Conf	284	-	633	335	75	94	643	298	16	675
HIPAA	136	146	846	134	189	134	948	140	12	979
Minitwit	102	-	114					115	9	91
Minitwit$^+$	121	-	119					141	9	100

* only functionality code, without registration functions

and 9–13), PythonTTC supports standard web-programming primitives (l. 2, 14–19) like the binding of URLs to functions (`@route`), SQL database queries (`sql`, which accepts `SELECT`, `DELETE`, and `INSERT` statements), templates (`render`), redirections (`redirect`), getting and setting session variables (`get_session`, `set_session`), reading GET and POST arguments (`get`, `post`), and sending data to third-parties over the network (`send`). The `render` function supports assigning different purposes to different arguments passed to the template renderer (l. 18-19 in Figure 4), generating one event per argument. Suppression of one of these events causes the corresponding argument to be set to a default error value. The `check` primitive from TTCWhile is also available (l. 19), and the constant `me` returns the identifier of the current user (l. 19). Note that despite using a Python-like syntax, PythonTTC cannot make use of external Python libraries, since these libraries may, in general, rely on Python features not supported by PythonTTC. Instead, WebTTC supports web-programming directly as part of PythonTTC.

WebTTC compiles PythonTTC programs deployed by programmers into Python3 code. It provides an out-of-the-box interface in which users can log in and specify their privacy policies, and implements the wrapper through which they can interact with individual applications. We use SQLite 3.31.1 for persistent storage and the state-of-the-art MFOTL enforcer EnfPoly [58] for enforcement. WebTTC consists of approximately 3,500 lines of Python code. We provide a ready-to-use image [4] containing all artifacts and tests.

The framework discharges our model's assumptions as follows:

(1) The assumptions on privacy policies (independence of past outputs, monotonicity, enforceability) are verified by restricting policies to a 'safe' syntactic fragment of MFOTL on which these requirements are fulfilled [9, 58].
(2) To keep the number of outputs in each function fixed, sending data to third-parties is disallowed in `if` and `then` blocks.
(3) The intervals between inputs are kept larger than the functions' running time by processing the next input only after the previous function terminates.
(4) Termination is ensured by introducing a timeout on function execution. A more powerful alternative would be requiring (automated) termination proofs from developers. Proving the termination of programs is a well-researched task (see, e.g., [46]) that is largely orthogonal to this work.

6 EVALUATION

Beyond showcasing the feasibility of TTC, our framework provides a basis for demonstrating how TTC can be used to enforce nontrivial user-provided privacy policies in real-world applications. We evaluate WebTTC by answering the following research questions:

RQ1. Is WebTTC sufficiently expressive to develop realistic web applications? How is the size of the code base impacted?
RQ2. How much runtime overhead does WebTTC incur? How does it scale with the database size and policy choice?
RQ3. How does the runtime performance of WebTTC compare to the performance of state-of-the-art IFC languages?

RQ1: Generality. We have implemented the following benchmark applications from previous work within our framework:

- A conference management system (**Conf**) [77];
- A HIPAA-style health record manager (**HIPAA**) [77];
- The microblogging app **Minitwit** [75]; and
- **Minitwit$^+$**, a **Minitwit** extension with personalized ads.

For each of these applications, we have implemented a Flask/Python baseline application without security, both using pure SQL (for **Conf** and **Minitwit**) and the SQLAlchemy ORM (for **HIPAA**). For the two applications implemented in Jacqueline [77], we also consider the original implementation. To our best knowledge, the Riverbed [75] implemention of Minitwit is not publicly available.

For each application, every argument of `call` or `render` was annotated with a purpose in {`Service`, `Marketing`, `Analytics`} as in Figure 4. In the considered applications, this process was straightforward, as all webpage components and calls to third-parties had a clearly identifiable purpose (e.g., a block containing ads has purpose `Marketing`, a list of other users' posts has purpose `Service`). However, we recall that correct purpose annotation is critical and generally requires certification by trusted legal experts (see Section 2.3).

We have been able to implement the original application functionality of all four applications in WebTTC. Table 1 shows the size of the code base for each of these implementations. Each PythonTTC implementation requires under 20 lines of security code, which consists mostly of applications of `check` in views showing lists of objects. The number of lines of code needed to implementation the application's functionality grows by about 10% with respect to the unsecured baseline, and is comparable to Jacqueline's.

In the experience of this paper's authors, porting applications from the Flask/Python baseline to WebTTC was a relatively smooth process. The similarity between the two interfaces allowed us to convert most code straightforwardly without altering its structure. The main difference between the Flask and WebTTC source codes consists of the introduction of checks in functions displaying lists of objects that may be subject to different permissions, and in the addition of purpose annotations. Some additional testing with various user policies was also required to ensure that the `check` statements had been adequately added. However, since programmers need not design the policies themselves, we expect the overall process of porting applications to WebTTC to be less time-consuming and more mechanical than with conventional IFC frameworks.

RQ2: Enforcement overhead. We assess the total runtime overhead of TTC by comparing the latency of selected function calls from our four test applications to the latency of the Flask/Python implementation without security. We thus obtain a *conservative* estimate of the overhead of TTC. For each application, we selected (i) a function displaying one relevant entity, if available (a *paper* in **Conf**, an *individual* in **HIPAA**), (ii) a function displaying a list of entities (all papers for **Conf**, all individuals for **HIPAA**, the latest 30 *posts* in **Minitwit**), and (iii) a function adding one entity, if available (a

Table 2: Average latency of Flask/Python, Jacqueline and WebTTC

(a) $n = 256$, $P_u = P_2^u$, $N = 100$ **(data in ms)**

Function	Impl.	u = 1	4	16	64	256	1024
Conf: `view_paper`	Flask	2	2	2	2	3	2
	Jacqueline	9	9	9	9	9	9
	WebTTC	4	4	4	4	4	5
	O_{Flask}	2.0	2.0	2.0	2.0	1.3	2.5
Conf: `view_papers`	Flask	61	62	62	62	62	62
	Jacqueline	392	390	389	390	389	391
	WebTTC	274	282	287	301	355	424
	O_{Flask}	4.5	4.5	4.6	4.9	5.7	6.8
Conf: `submit_paper`	Flask	5	5	5	5	6	6
	Jacqueline	10	10	10	10	9	10
	WebTTC	9	8	8	8	8	9
	O_{Flask}	1.8	1.6	1.6	1.6	1.3	1.5
HIPAA: `view_patient`	Flask	3	3	3	3	3	3
	Jacqueline	65	65	67	66	47	57
	WebTTC	2	3	2	2	3	3
	O_{Flask}	0.7	1.0	0.7	0.7	1.0	1.0
HIPAA: `view_index`	Flask	6	6	6	6	6	6
	Jacqueline	780	786	784	770	747	816
	WebTTC	103	103	103	103	103	106
	O_{Flask}	17	17	17	17	17	18
Minitwit: `timeline`	Flask	2	2	2	3	3	2
	WebTTC	9	9	10	10	12	12
	O_{Flask}	4.5	4.5	5	3.3	4.0	6.0
Minitwit: `add_message`	Flask	6	6	6	5	5	5
	WebTTC	15	15	10	6	6	6
	O_{Flask}	2.5	2.5	1.7	1.2	1.2	1.2
Minitwit$^+$: `timeline`	Flask	4	4	5	6	6	7
	WebTTC	10	10	10	11	13	13
	O_{Flask}	2.5	2.5	2.0	1.8	2.2	1.9
Minitwit$^+$: `add_message`	Flask	7	6	7	7	9	9
	WebTTC	17	17	13	6	5	6
	O_{Flask}	2.4	2.8	1.9	0.9	0.6	0.7

(b) $u = 256$, $P_u = P_2^u$, $N = 100$ **(data in ms)**

Function	Impl.	n = 1	4	16	64	256	1024
Conf: `view_paper`	Flask	3	3	2	3	3	3
	Jacqueline	9	10	10	10	9	10
	WebTTC	4	4	4	4	4	4
	O_{Flask}	1.3	1.3	2	1.3	1.3	1.3
Conf: `view_papers`	Flask	2	3	6	17	62	239
	Jacqueline	6	11	29	100	389	1637
	WebTTC	4	7	22	82	355	1607
	O_{Flask}	2.0	2.3	3.7	4.8	5.7	6.7
Conf: `submit_paper`	Flask	6	6	5	6	6	6
	Jacqueline	9	10	9	10	9	10
	WebTTC	8	9	9	9	8	9
	O_{Flask}	1.3	1.5	1.8	1.5	1.3	1.5
HIPAA: `view_patient`	Flask	3	3	3	3	3	3
	Jacqueline	30	30	31	33	47	39
	WebTTC	3	2	3	3	3	3
	O_{Flask}	0.7	1.0	1.0	1.0	1.0	1.0
HIPAA: `view_index`	Flask	3	3	3	4	6	16
	Jacqueline	34	40	65	164	747	4560
	WebTTC	2	3	5	15	103	1887
	O_{Flask}	0.7	1.0	1.7	3.8	17	118
Minitwit: `timeline`	Flask	1	2	2	2	3	3
	WebTTC	3	4	8	12	12	11
	O_{Flask}	3.0	2.0	4.0	6.0	4.0	3.7
Minitwit: `add_message`	Flask	4	5	5	5	5	6
	WebTTC	6	6	6	7	6	7
	O_{Flask}	1.5	1.2	1.2	1.4	1.2	1.2
Minitwit$^+$: `timeline`	Flask	6	6	6	6	6	7
	WebTTC	3	4	8	13	13	11
	O_{Flask}	0.5	0.7	1.3	2.2	2.2	1.6
Minitwit$^+$: `add_message`	Flask	7	7	7	9	9	9
	WebTTC	5	6	5	5	5	7
	O_{Flask}	0.7	0.9	0.7	0.6	0.6	0.8

(c) $n = 256$, $u = 256$, $N = 100$ **(data in ms)**

Function	Impl.	P_u P_0	P_1	P_2	P_3
Conf: `view_paper`	WebTTC	4	4	4	4
Conf: `view_papers`	WebTTC	421	369	355	368
Conf: `submit_paper`	WebTTC	8	8	8	8
HIPAA: `view_patient`	WebTTC	2	3	3	3
HIPAA: `view_index`	WebTTC	102	102	103	102
Minitwit: `timeline`	WebTTC	12	8	12	8
Minitwit: `add_message`	WebTTC	6	6	6	7
Minitwit$^+$: `timeline`	WebTTC	14	8	13	8
Minitwit$^+$: `add_message`	WebTTC	5	6	5	6

(d) $n = 256$, $u = 256$, $N = 100$ **(data in ms)**

Function	Policy	WebTTC	Jacqueline
Conf: `view_paper`	$P^u_{\textbf{Conf}}$	4	9
Conf: `view_papers`	$P^u_{\textbf{Conf}}$	363	389
Conf: `submit_paper`	$P^u_{\textbf{Conf}}$	9	9
HIPAA: `view_patient`	$P^u_{\textbf{HIPAA}}$	2	47
HIPAA: `view_index`	$P^u_{\textbf{HIPAA}}$	103	747

(e) Requirements covered by the policies

Policy	[R1]	[R2]	[R3]
P_0			
P_1	✓		
P_2	✓	✓	✓
P_3	✓		
$P^u_{\textbf{Conf}}$	✓	✓	
$P^u_{\textbf{HIPAA}}$	✓	✓	

paper in **Conf**, a post in **Minitwit**). For each function, we measure the page latency using the Python `requests` library on a high-end laptop (Intel Core i5-1135G7, 32 GB RAM) over $N = 100$ repetitions, while varying the number n of entities, the number u of users, and the policies P_u of users. Large values of n cover the scenario where the application was used for an extended period of time. The range of values for n and u is taken from Yang et al. [77]. We set P_u to either $P_0^u, P_1^u, P_2^u, P_3^u$ where, for $i \in \{0, \ldots, 3\}$, P_i^u is identical to policy P_i in Figure 1, except that Alice is replaced by u and posts are replaced by papers in **Conf** and individuals in **HIPAA**. We denote by O_{Flask} the ratio between the latencies of WebTTC and Flask.

The results of our experiments is presented in Tables 2a–b. The latency is constant with respect to n and u for functions of type (i) and (iii), as well as for `timeline`. In these functions, a finite number of elements is displayed and WebTTC only adds a constant number of checks, leading to a constant, low overhead between 1 and 2.5. The latency of these functions remains consistently below 15 ms, irrespective of the choice of n, u, and P_u. For `view_papers` and `view_index`, we observe both linear latency in n and an increase in latency when u and n become large, with the overhead versus Flask being 1–10 for `view_papers`, or even larger for `view_index`. The larger overhead is caused by the large number of checks and more complex UT propagation induced by the large number of inputs being combined to compute the output. These functions show all entities stored in the database, a 'stress test' [77] that is unlikely to materialize in applications featuring appropriate pagination.

Provided that pagination is introduced, our experiments thus show that the overhead of the TTC application with respect to a baseline without security is moderate, and that its performance is at least similar to that of conventional IFC systems. With a slowdown of 1 to 2.5× and latency below 15 ms for all values of the parameters tested, the usability of the case study applications is not significantly impacted by the additional runtime costs.

RQ3: Comparison with the state of the art. In the previous section, we have given a conservative estimate of the overhead of TTC with respect to a baseline *without security*. Next, we compare TTC's runtime performance on **Conf** and **HIPAA** to that of Jacqueline [77], a state-of-the-art Python-based IFC web framework [78]. For all values of n and u tested, WebTTC exhibits a latency equal to or lower than Jacqueline's (see Table 2a–b). Better performance is also observed in those functions for which WebTTC exhibits a significant overhead with respect to the baseline without security.

Since Jacqueline is based on a different, developer-specified policy model, we additionally converted **Conf**'s and **HIPAA**'s original IFC policies from [77] into MFOTL privacy policies $P_{\textbf{Conf}}$ and $P_{\textbf{HIPAA}}$ that lead to equivalent information-flow restrictions (see Appendix D for details). The results of these additional experiments, presented in Table 2d, confirm that for all values of n and u, WebTTC exhibits a latency equal or lower than Jacqueline's.

7 RELATED WORK

IFC languages. Denning's seminal works [30, 31] gave rise to a rich IFC literature. Many IFC languages have been developed, including JIF [61], FlowCaml [72], Jeeves [78], JSFlow [56], LWeb [64], and others [7, 20, 22, 23, 27, 47, 49, 59, 60, 65, 71, 75]. Our work builds on two ideas from this field: dynamic information-flow [6, 39, 67, 77]

	JIF [61]	FlowCaml [72]	SIF [23]	SELinks [27]	UrFlow [20]	Jeeves [78]	JSFlow [56]	Jacqueline [77]	JSLINQ [7]	Hails [47]	DAISY [49]	LWeb [64]	Riverbed [75]	Estrela [13]	Lifty [65]	Storm [59]	**WebTTC**
[R1] User-defined	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓
[R2] Fine-grained	✓	✗	✓	_[a]	_[a]	_[a]	✗	_[a]	✓	_[a]	_[a]	✗	✗	_[a]	_[a]	_[a]	✓
[R3] Time-dependent	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Proofs	✓	✓	✗	✗	✗	✓	✓	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓
Noninterference	✓	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓
Code available	✓	✓	✓	✓	✓	?	✓	✓	✓	?	✓	✓	?	✓	?	✓	✓

[a] Fine-grained restrictions defined at the data-model level only.

Table 3: Comparison of related work

and monitored execution [50, 51] with a monitor integrated into the program's semantics ensuring compliance with a specification.

Table 3 summarizes previous work with respect to support for [R1-3], existence of formal correctness proofs for a core of the language ('Proofs'), and the enforcement of noninterference, as opposed to just access control ('Noninterference'). Riverbed [75] is, to our knowledge, the only approach tackling user-defined data usage policies [R1]. However, Riverbed does not consider noninterference, and comes with no formal proofs. More critically, its policy language has very limited expressivity: a user can only (dis)allow that *all* their data is aggregated with data of other users or written to permanent storage. As a result, users can neither associate custom policies to different inputs they provide nor exchange information with users that have different policies.

Jif [61], SIF [23], FlowCaml [72], JSLINQ [7], Lifty [65], and Storm [59] use type systems, while UrFlow[20] relies on symbolic execution and theorem proving to enforce privacy policies. These inherently static approaches are not applicable to the case of user-defined policies [R1], since programs would then need to be checked against arbitrary conjunctions of user policies.

The dynamic IFC systems of Jeeves [78], Jacqueline [77], Hails [47], LWeb [64], and Estrela [13] rely on policies defined at the data-model level. Hails [47] and LWeb [64] are based on Haskell monads, Estrela [13] on modifying database queries, and Jeeves [78] and Jacqueline [77] on a modified λ-calculus. In Jeeves and Jacqueline, sensitive values are evaluated either to their actual value or a default value according to the content of some *level variable*, which is set according to the application's policy. In all these systems, the fields or rows of databases can be assigned confidentiality labels, whose semantics may depend on the values of the data and on the context of execution. Such approaches support fine-grained policies but not arbitrary *per-input* policies as in requirement [R2], as they model interference from the database to outputs rather than from inputs to outputs. In particular, labels are not persisted between user queries.

Jif [61], SIF [23], and JSLINQ [7] support to statically reasoning about interference between inputs and outputs in a fine-grained way, thus fulfilling [R2]. Additionally, in SIF [23] and JSLINQ [7], programs can dynamically create new security labels encoding data usage permissions. This can be used to let users select between different policy options. However, even then, the policy space available to users is still strictly limited by the developers' choices, and users obtain no formal guarantees that the preferences they express are correctly taken into account. Hence, these systems fall short of fulfilling [R1]. In contrast, SELinks [27] uses a type system to guarantee that *access* to sensitive data is always guarded by appropriate policy checks. In SELinks, arbitrary functions can be assigned to security labels to define custom policies, a mechanism that could possibly be extended to support user-defined policies; however, SELinks does not provide non-interference guarantees. None of the above approaches supports time-dependent policies [R3].

Other approaches. While we focus on the enforcement of IFC in server-side code, IFC techniques are also used in the browser [11, 14, 15, 19, 29, 32, 56, 73] to avoid leaks caused by client-side code.

Another line of research has considered temporal IFC from the point of view of runtime verification and (trace-based) runtime enforcement. Hyperproperty extensions of temporal logic such as HyperLTL [24] and SecLTL [33] can be used to monitor noninterference. However, hyperproperty monitors usually take the set of all system traces as an input [2, 18, 26, 40–42, 52], which cannot be computed in reasonable time for general programs.

For classical (non-hyper-)properties, efficient monitors [9, 10, 54, 70] and enforcers [58] exist and have been used to verify data usage [8] and GDPR [5] restrictions in real-world settings.

Differential privacy [37] is a promising approach to enforcing privacy regulations [28], providing strong *statistical* privacy guarantees. However, being statistical, these guarantees may be practically insufficient or of limited usability depending on the data type, the size of datasets, and the queries considered [35, 36, 76].

The combination of fully homomorphic encryption [45] with trusted computing allows outsourcing complex computations with reduced information leakage [43, 63]. However, these approaches are still prohibitively slow for general computation, while more efficient alternatives [43] sacrifice the coverage of implicit flows.

8 CONCLUSION

In this paper, we have presented Taint, Track, and Control (TTC), the first language-based PbD approach that enforces user-defined, fine-grained, temporal privacy policies in online applications against both malicious peers and developers. We have introduced a notion of information-flow traces that can be used to define expressive privacy policies; defined the formal semantics of TTC and proven its correctness; and implemented and evaluated WebTTC, a framework that allows for the development of private-by-design applications.

The work described in this paper can be extended to address further aspects of privacy regulations. In addition to requiring consent-based usage, the GDPR recognizes users' "right to be forgotten." Enforcing such a right requires timely *erasure* of data at rest, i.e., *causation* of systems actions [58]. Hence, adding support for causation to our TTC framework would allow it to cover a larger set of the GDPR requirements. GDPR also allows programmers to disregard user consent when data is anonymized, or when other legal grounds (e.g., *legitimate interest*) are met. One may therefore consider extending TTC with controlled *declassification* of user inputs.

Extending TTC (and WebTTC) to support more modern software systems features, like communicating components, is another possible future work. With more features and a broad GDPR coverage, studies involving external developers and users would become possible, paving the way to the development of new generation of user-centric, private-by-design software systems.

REFERENCES

[1] 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *Official Journal of the European Union* (2016).

[2] Shreya Agrawal and Borzoo Bonakdarpour. 2016. Runtime verification of k-safety hyperproperties in HyperLTL. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. IEEE, 239–252.

[3] Anonymous authors. 2023. User-Controlled Privacy: Taint, Track, and Control. Isabelle/HOL formalization. https://drive.google.com/file/d/1Y2YMu3p92ke1fChbQG_UtUDek2NRkiXo

[4] Anonymous authors. 2023. User-Controlled Privacy: Taint, Track, and Control. VirtualBox Image. https://drive.google.com/file/d/1l1sByhZAtfFd9yjLGlvh1-LIPJaOGzm4

[5] Emma Arfelt, David Basin, and Søren Debois. 2019. Monitoring the GDPR. In *24th European Symposium on Research in Computer Security (ESORICS, Vol. 11735)*, Karuze Sako, Steve Schneider, and Peter Y. A. Ryan (Eds.). Springer, 681–699. https://doi.org/10.1007/978-3-030-29959-0_33

[6] Thomas H. Austin and Cormac Flanagan. 2009. Efficient purely-dynamic information flow analysis. In *4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, Stephen Chong and David A. Naumann (Eds.). ACM, 113–124. https://doi.org/10.1145/1554339.1554353

[7] Musard Balliu, Benjamin Liebe, Daniel Schoepe, and Andrei Sabelfeld. 2016. Jslinq: Building secure applications across tiers. In *6th ACM Conference on Data and Application Security and Privacy (CODASPY)*. 307–318. https://doi.org/10.1145/2857705.2857717

[8] David Basin, Matús Harvan, Felix Klaedtke, and Eugen Zalinescu. 2013. Monitoring Data Usage in Distributed Systems. *IEEE Trans. Software Eng.* 39, 10 (2013), 1403–1426. https://doi.org/10.1109/TSE.2013.18

[9] David Basin, Felix Klaedtke, Samuel Müller, and Eugen Zălinescu. 2015. Monitoring Metric First-Order Temporal Properties. *J. ACM* 62, 2, Article 15 (May 2015), 45 pages. https://doi.org/10.1145/2699444

[10] David Basin, Felix Klaedtke, and Eugen Zalinescu. 2017. The MonPoly Monitoring Tool. In *An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*, Giles Reger and Klaus Havelund (Eds.), Vol. 3. EasyChair, 19–28. https://doi.org/10.29007/89hs

[11] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. 2015. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *NDSS*.

[12] Lujo Bauer, Jarred Ligatti, and David Walker. 2002. More enforceable security policies. In *Workshop on Foundations of Computer Security (FCS)*. Citeseer.

[13] Abhishek Bichhawat, Matt Fredrikson, Jean Yang, and Akash Trehan. 2020. Contextual and granular policy enforcement in database-backed applications. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 432–444.

[14] Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. 2014. Information flow control in WebKit's JavaScript bytecode. In *Principles of Security and Trust*, Vol. 3. Springer, 159–178.

[15] Abhishek Bichhawat, Vineet Rajani, Jinank Jain, Deepak Garg, and Christian Hammer. 2017. Webpol: Fine-grained information flow policies for web browsers. In *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I 22*. Springer, 242–259.

[16] European Data Protection Board. 2019. Guidelines 4/2019 on Article 25. Data Protection by Design and by Default.

[17] Dino Bollinger, Karel Kubicek, Carlos Cotrini, and David Basin. 2022. Automating cookie consent and GDPR violation detection. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association.

[18] Borzoo Bonakdarpour and Bernd Finkbeiner. 2016. Runtime verification for HyperLTL. In *International Conference on Runtime Verification*. Springer, 41–45.

[19] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering information leakage from browser extensions. In *ACM SIGSAC Conference on Computer and Communications Security*. 1687–1700.

[20] Adam Chlipala. 2010. Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications. In *9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, Remzi H. Arpaci-Dusseau and Brad Chen (Eds.). USENIX, 105–118.

[21] Jan Chomicki. 1995. Efficient Checking of Temporal Integrity Constraints Using Bounded History Encoding. *ACM Trans. Database Syst.* 20, 2 (June 1995), 149–186. https://doi.org/10.1145/210197.210200

[22] Stephen Chong, Andrew C. Myers, K. Vikram, and Lantian Zheng. 2009. Jif Reference Manual. https://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html

[23] Stephen Chong, K. Vikram, and Andrew C. Myers. 2007. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*, Niels Provos (Ed.). USENIX.

[24] Michael R Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K Micinski, Markus N Rabe, and César Sánchez. 2014. Temporal logics for hyperproperties. In *International Conference on Principles of Security and Trust*. Springer, 265–284.

[25] Michael R Clarkson and Fred B Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (2010), 1157–1210.

[26] Norine Coenen, Bernd Finkbeiner, Christopher Hahn, Jana Hofmann, and Yannick Schillo. 2021. Runtime enforcement of hyperproperties. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 283–299.

[27] Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. 2009. Cross-Tier, Label-Based Security Enforcement for Web Applications. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM, 269–282. https://doi.org/10.1145/1559845.1559875

[28] Rachel Cummings and Deven Desai. 2018. The role of differential privacy in gdpr compliance. In *Workshop on Responsible Recommendation (FATREC)*.

[29] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 748–759.

[30] Dorothy E. Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (may 1976), 236–243. https://doi.org/10.1145/360051.360056

[31] Dorothy E. Denning and Peter J. Denning. 1977. Certification of programs for secure information flow. *Commun. ACM* 20, 7 (jul 1977), 504–513. https://doi.org/10.1145/359636.359712

[32] Dominique Devriese and Frank Piessens. 2010. Noninterference through secure multi-execution. In *IEEE Symposium on Security and Privacy*. IEEE, 109–124.

[33] Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N Rabe, and Helmut Seidl. 2012. Model checking information flow in reactive systems. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 169–185.

[34] Rayna Dimitrova, Bernd Finkbeiner, and Markus N Rabe. 2012. Monitoring temporal information flow. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer, 342–357.

[35] Zeyu Ding, Yuxin Wang, Guanhong Wang, Danfeng Zhang, and Daniel Kifer. 2018. Detecting violations of differential privacy. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 475–489. https://doi.org/10.1145/3243734.3243818

[36] Linkang Du, Zhikun Zhang, Shaojie Bai, Changchang Liu, Shouling Ji, Peng Cheng, and Jiming Chen. 2021. AHEAD: Adaptive Hierarchical Decomposition for Range Query under Local Differential Privacy. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 1266–1288. https://doi.org/10.1145/3460120.3485668

[37] Cynthia Dwork. 2006. Differential privacy. In *International Colloquium on Automata, Languages, and Programming*. Springer, 1–12.

[38] Matthew B Dwyer, George S Avrunin, and James C Corbett. 1998. Property specification patterns for finite-state verification. In *Proceedings of the second workshop on Formal methods in software practice*. 7–15.

[39] J. S. Fenton. 1974. Memoryless subsystems. *Comput. J.* 17, 2 (1974), 143–147. https://doi.org/10.1093/comjnl/17.2.143

[40] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. 2018. RVHyper: A Runtime Verification Tool for Temporal Hyperproperties. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 194–200.

[41] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. 2019. Monitoring hyperproperties. *Formal Methods in System Design* 54, 3 (2019), 336–363.

[42] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. 2020. Efficient monitoring of hyperproperties using prefix trees. *International Journal on Software Tools for Technology Transfer* 22, 6 (2020), 729–740.

[43] Andreas Fischer, Benny Fuhry, Florian Kerschbaum, and Eric Bodden. 2020. Computation on Encrypted Data using Dataflow Authentication. *Proc. Priv. Enhancing Technol.* 2020, 1 (2020), 5–25. https://doi.org/10.2478/popets-2020-0002

[44] Michal S Gal and Oshrit Aviv. 2020. The competitive effects of the GDPR. *Journal of Competition Law & Economics* 16, 3 (2020), 349–391.

[45] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *41st Annual ACM Symposium on Theory of Computing (STOC)*, Michael Mitzenmacher (Ed.). ACM, 169–178. https://doi.org/10.1145/1536414.1536440

[46] Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. 2006. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *Automated Reasoning: Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 3*. Springer, 281–286.

[47] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. 2017. Hails: Protecting data privacy in untrusted web applications. *J. Comput. Secur.* 25, 4-5 (2017), 427–461. https://doi.org/10.3233/JCS-15801

[48] Joseph A Goguen and José Meseguer. 1982. Security policies and security models. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 11–20. https://doi.org/10.1109/SP.1982.10014

[49] Marco Guarnieri, Musard Balliu, Daniel Schoepe, David Basin, and Andrei Sabelfeld. 2019. Information-flow control for database-backed applications. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, IEEE, 79–94. https://doi.org/10.1109/EuroSP.2019.00016

[50] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A Schmidt. 2006. Automata-based confidentiality monitoring. In *Annual Asian Computing Science Conference*. Springer, 75–89.

[51] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A. Schmidt. 2007. Automata-Based Confidentiality Monitoring. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*. Springer, 75–89. https://doi.org/10.1007/978-3-540-77505-8_7

[52] Christopher Hahn. 2019. Algorithms for monitoring hyperproperties. In *International Conference on Runtime Verification*. Springer, 70–90.

[53] David Harborth, Maren Braun, Akos Grosz, Sebastian Pape, and Kai Rannenberg. 2018. Anreize und Hemmnisse für die Implementierung von Privacy-Enhancing Technologies im Unternehmenskontext. *Sicherheit* (2018).

[54] Klaus Havelund, Doron Peled, and Dogan Ulus. 2018. DejaVu: A Monitoring Tool for First-Order Temporal Logic. In *2018 IEEE Workshop on Monitoring and Testing of Cyber-Physical Systems (MT-CPS)*. IEEE. https://doi.org/10.1109/mt-cps.2018.00013

[55] Katia Hayati and Martín Abadi. 2004. Language-based enforcement of privacy policies. In *International Workshop on Privacy Enhancing Technologies*. Springer, 302–313.

[56] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking Information Flow in JavaScript and Its APIs. In *29th ACM Symposium on Applied Computing (SAC)*. ACM, 1663–1671. https://doi.org/10.1145/2554850.2554909

[57] Mireille Hildebrandt and Laura Tielemans. 2013. Data protection by design and technology neutral law. *Computer Law & Security Review* 29, 5 (2013), 509–521.

[58] François Hublet, David Basin, and Srđan Krstić. 2022. Real-time Policy Enforcement with Metric First-Order Temporal Logic. In *Proceedings of ESORICS'22*.

[59] Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. 2021. {STORM}: Refinement Types for Secure Web Applications. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 441–459.

[60] Jed Liu, Michael D George, Krishnaprasad Vikram, Xin Qi, Lucas Waye, and Andrew C Myers. 2009. Fabric: A platform for secure distributed computation and storage. In *ACM SIGOPS 22nd symposium on Operating systems principles*. 321–334.

[61] Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 228–241. https://doi.org/10.1145/292540.292561

[62] Henrik Nergaard, Nils Ulltveit-Moe, Terje Gj, et al. 2015. A scratch-based graphical policy editor for XACML. In *2015 International Conference on Information Systems Security and Privacy (ICISSP)*. IEEE, 1–9.

[63] Elena Pagnin, Carlo Brunetta, and Pablo Picazo-Sanchez. 2018. HIKE : Walking the Privacy Trail. In *17th International Conference on Cryptology and Network Security (CANS, Vol. 11124)*, Jan Camenisch and Panos Papadimitratos (Eds.). Springer, 43–66. https://doi.org/10.1007/978-3-030-00434-7_3

[64] James Parker, Niki Vazou, and Michael Hicks. 2019. LWeb: Information Flow Security for Multi-Tier Web Applications. *Proc. ACM Program. Lang.* 3, POPL, Article 75 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290388

[65] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. 2020. Liquid information flow control. *ACM on Programming Languages* 4, ICFP (2020), 1–30.

[66] Ira S Rubinstein. 2011. Regulating privacy by design. *Berkeley Tech. LJ* 26 (2011), 1409.

[67] Andrei Sabelfeld and Alejandro Russo. 2010. From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research. In *Perspectives of Systems Informatics*. Springer, 352–365. https://doi.org/10.1007/978-3-642-11486-1_30

[68] Beata A Safari. 2016. Intangible privacy rights: How europe's gdpr will set a new global standard for personal data protection. *Seton Hall L. Rev.* 47 (2016), 809.

[69] Fred B. Schneider. 2000. Enforceable Security Policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (Feb. 2000), 30–50. https://doi.org/10.1145/353323.353382

[70] Joshua Schneider, David Basin, Srđan Krstić, and Dmitriy Traytel. 2019. A Formally Verified Monitor for Metric First-Order Temporal Logic. In *Runtime Verification*, Bernd Finkbeiner and Leonardo Mariani (Eds.). LNCS, Vol. 11757. Springer, 310–328. https://doi.org/10.1007/978-3-030-32079-9_18

[71] Daniel Schoepe, Daniel Hedin, and Andrei Sabelfeld. 2014. SeLINQ: Tracking Information across Application-Database Boundaries. In *19th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 25–38. https://doi.org/10.1145/2628136.2628151

[72] Vincent Simonet. 2003. The Flow Caml System (version 1.00): Documentation and user's manual. http://www.normalesup.org/~simonet/soft/flowcaml/manual/

[73] Deian Stefan, Edward Z Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazieres. 2014. Protecting Users by Confining JavaScript with {COWL}. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 131–146.

[74] Shukun Tokas, Olaf Owe, and Toktam Ramezanifarkhani. 2019. Language-based mechanisms for privacy-by-design. In *IFIP International Summer School on Privacy and Identity Management*. Springer, 142–158.

[75] Frank Wang, Ronny Ko, and James Mickens. 2019. Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 615–630. https://www.usenix.org/conference/nsdi19/presentation/wang-frank

[76] Benjamin Weggenmann and Florian Kerschbaum. 2021. Differential Privacy for Directional Data. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 1205–1222. https://doi.org/10.1145/3460120.3484734

[77] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, Dynamic Information Flow for Database-Backed Applications. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chandra Krintz and Emery Berger (Eds.). ACM, 631–647. https://doi.org/10.1145/2908080.2908098

[78] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A Language for Automatically Enforcing Privacy Policies. In *39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 85–96. https://doi.org/10.1145/2103656.2103669

[79] Le Yu, Tao Zhang, Xiapu Luo, and Lei Xue. 2015. Autoppg: Towards automatic generation of privacy policy for android applications. In *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. 39–50.

A FORMAL DEFINITIONS AND PROOFS

Our Isabelle/HOL formalization [3] is divided into four files:

File	Section	LOC	Description
`Traces.thy`	2–3	633	Applications, traces
`TaggedValues.thy`	4.4	518	Tagged values
`TTCWhile.thy`	4.6.1–4.6.2	1145	TTCWhile
`TTC.thy`	4.2–4.5, 4.6.3, 4.7–4.8	1995	TTC semantics

This section summarizes the main definitions and theorems.

A.1 Applications and traces (`Traces.thy`)

We denote with :: the standard cons operation and with · concatenation on tuples and lists, i.e., $a_1 :: [a_2, \ldots, a_k] = [a_1, \ldots, a_k]$ and $[a_1, \ldots, a_k] \cdot [b_1, \ldots, b_k] = [a_1, \ldots, a_k, b_1, \ldots, b_k]$. We assume given a set of states $\mathbb{S}$, an initial state S_0, a transition relation $\rightarrow$, and a boolean predicate `wf_in_seq2` on input sequences.

Definition A.1 (Well-formed input sequences, `wf_in_seq'`). An input sequence I is called well-formed, written `wf_in_seq'` I, if `wf_in_seq2` I holds and the timestamps in I are strictly increasing.

Definition A.2 (IO trace, `ttrans`). We define $\overset{\cdot}{\Longrightarrow}$ inductively as:

$$\frac{}{S, I \overset{()}{\Longrightarrow} S, I}\ \text{IO\_NOP}$$

$$\frac{S, I \overset{\sigma}{\Longrightarrow} S', \left(u, f, A, \tau''\right) :: T, \quad S' \xrightarrow{\mathrm{i}(u,f,A,\tau'')} S''}{S, I \xrightarrow{\sigma \cdot (\tau'', \{\mathrm{in}(u,f,a,d) \mid A(a) = d\})} S'', T}\ \text{IO\_IN}$$

$$\frac{S, I \overset{\sigma}{\Longrightarrow} S', I' \quad S' \xrightarrow{\bullet} S''}{S, I \overset{\sigma}{\Longrightarrow} S'', I'}\ \text{IO\_NONE}$$

$$\frac{S, I \overset{\sigma}{\Longrightarrow} S', I', \quad S' \xrightarrow{\mathrm{o}(f,u,p,d,\tau'')} S''}{S, I \xrightarrow{\sigma \cdot (\tau'', \{\mathrm{out}(f,u,p,d)\})} S'', I'}\ \text{IO\_OUT.}$$

The formal definition of non-interference is as follows. Assume we can reach state S_1 from S_0 by processing the inputs I, i.e., we have $S_0, I \overset{\sigma}{\underset{\pi,P}{\Longrightarrow}} S_1, I_1$ for some σ, I_1. Moreover, assume that there exists $1 \leq j \leq |I|$ and a an argument name such that $I_j = (u, f, A, \tau)$, $a \in \operatorname{dom} A$, and $A[a] = d$. We focus on this input, which has value d and UT (τ, a). For any $d' \in \mathbb{D}$, denote by $I(d'/d)$ the input sequence I' that is identical to I, except that $I'_j = (u, f, A[a := d'], \tau)$. Now, assume that in the transition following state S_1, the application produces an output with value v_1 and identifier τ_1. We say that input (τ, a) interferes with output τ_1, written $(\tau, a) \rightsquigarrow_{\pi,P,I} \tau_1$, if there exists at least one value of the input identified by (τ, a) that is incompatible with the output we observe. In other words, $(\tau, a) \rightsquigarrow_{\pi,P,I} \tau_1$ holds if we can find $d' \in \mathbb{D}$ such that if $I(d'/d), 0 \overset{\sigma'}{\underset{\pi,P}{\Longrightarrow}} S'_1, I_1$, then the application defined by $\dashrightarrow_{\pi,P}$ never outputs v_1 to u at time τ_1 in S'_1. Thus, observing v_1 tells us that the value of i cannot have been d'.

Definition A.3 (Input-output interference, `interf`*).*
Assume that we have

$$a \in \operatorname{dom} A \quad I = H \cdot (u, f, A, \tau) :: T \quad S_0, I \overset{\sigma}{\Longrightarrow} S_1, I_1$$

$$S_1 \xrightarrow{\mathsf{o}(f_1, u_1, p_1, v_1, \tau_1)} S'.$$

We say that input (τ, a) interferes with output τ_1, written $(\tau, a) \rightsquigarrow_I \tau_1$, iff there exists I', d', σ'_1, τ_2 such that

$$I' = H \cdot \left(u, f, A[a := d'], \tau\right) :: T$$

$$S_0, I' \overset{\sigma'_1 \cdot (\tau_2, D)}{\Longrightarrow} S'_1, I'_1 \quad \tau_2 \geq \tau_1$$

$$\forall i \leq |\sigma'_1|.\ \forall D'_1.\ (\sigma'_1 \cdot (\tau_2, D))_i = (\tau_1, D'_1) \Rightarrow \mathsf{out}(f_1, u_1, p_1, v_1) \notin D'_1$$

Definition A.4 (Information-flow traces, enforcement). We say that an application *enforces* a property P, written `enforces` P, iff, for any input sequence I such that `wf_in_seq'` I and $S_0, I \overset{\sigma}{\Longrightarrow} S', I'$, we have $\Delta_I(\sigma) \in P$, where

$$\rho(\mathsf{in}(u, f, a, d), \tau) = \{\mathsf{In}(u, f, (\tau, a))\}$$

$$\rho(\mathsf{out}(f, u, p, d), \tau) = \{\mathsf{Out}(f, u, p, \tau)\} \cup \{\mathsf{Itf}((\tau', a), \tau) \mid (\tau', a) \rightsquigarrow_I \tau\}$$

$$\delta_I(\tau, D) = \left(\tau, \bigcup_{e \in D} \rho_I(e, \tau)\right)$$

$$\Delta_I = \mathsf{map}\ \delta_I.$$

We will use the following lemma to prove correct enforcement through (information-flow-trace) approximation:

Lemma A.5 (`enforces_approx`). *Let $\sqsubseteq \in \mathcal{P}(\mathbb{T} \times \mathbb{T})$, $\Pi \subseteq \mathbb{T}$, and $t : \mathbb{S} \to \mathbb{T}_{\Sigma_{\mathrm{IF}}}$. Assume that the following conditions hold:*

(1) $\forall \sigma, \sigma'.\ P \in \Pi \wedge \sigma' \in P \wedge \sigma \sqsubseteq \sigma' \Rightarrow \sigma \in P$;

(2) $\forall I, \sigma, S, I'.\ P \in \Pi \wedge$ `wf_in_seq'` $I \wedge \left[S_0, I \overset{\sigma}{\Longrightarrow} S, I'\right] \Rightarrow \Delta_I(\sigma) \sqsubseteq t\ S$;

(3) $\forall I, \sigma, S, I'.\ P \in \Pi \wedge$ `wf_in_seq'` $I \wedge \left[S_0, I \overset{\sigma}{\Longrightarrow} S, I'\right] \Rightarrow t\ S \in P$.

Then $P \in \Pi \Rightarrow$ `enforces` P.

Here, we will instantiate $\sqsubseteq$ with the following:

Definition A.6. We define $\sqsubseteq_{\mathrm{TTC}} \in \mathcal{P}(\mathbb{T} \times \mathbb{T})$ as follows:

$$((\tau_1, D_1), \ldots, (\tau_k, D_k)) \sqsubseteq_{\mathrm{TTC}} ((\tau'_1, D'_1), \ldots, (\tau'_{k'}, D'_{k'}))$$

$$\Leftrightarrow \exists (\hat{\tau}_i, \hat{D}_i)_{1 \leq i \leq \hat{k}}.\ \mathsf{remove\_empty}((\tau'_i, D'_i)_{1 \leq i \leq k'}) = (\hat{\tau}_i, \hat{D}_i)_{1 \leq i \leq \hat{k}}$$

$$\wedge\ k = \hat{k}$$

$$\wedge\ (\forall 1 \leq i \leq k.\ \tau_i = \hat{\tau}_i \wedge D_i \subseteq \hat{D}_i \wedge \hat{D}_i \setminus D_i \subseteq \{\mathsf{Itf}(i, o) \mid i, o\})$$

where

$$\mathsf{remove\_empty}((\tau_i, D_i)_{1 \leq i \leq k}) = [(\tau_i, D_i) \mid 1 \leq i \leq k,\ D_i \neq \emptyset].$$

A.2 Tagged values (`TaggedValues.thy`)

Let $\mathcal{H}$ denote the set of UT histories, $\mathcal{T}$ the set of tagged values, and Var the set of variable names. We consider the following operations on UT histories:

- **Concatenation**: $\alpha \cdot \beta$ (see above);
- **Union**: $\alpha \Cup \beta$ (see above);
- **Normalization**: $\nu([L_1, \ldots, L_k]) = [L_1, L_2 \backslash L_1, \ldots, L_k \backslash \cup_{i=1}^{k-1} L_i]$;
- **Normalized concatenation**: $\alpha \mathbin{\hat{\cdot}} \beta := \nu(\alpha \cdot \beta)$;
- **Normalized union**: $\alpha \mathbin{\hat{\Cup}} \beta := \nu(\alpha \Cup \beta)$.

The normalization operation preserves the set of UTs contained in the history while removing all duplicates, thus improving both the memory and runtime performance of the interpreter. The normalized operators $\hat{\cdot}$ and $\hat{\Cup}$ are obtained from the standard operators $\cdot$ and $\Cup$ by applying the standard operator followed by a normalization step. As we will show next, the normalized operators can be used in place of the standard operators in the semantics of TTCWhile without prejudice to security. For this reason, while we sticked to $\cdot$ and $\Cup$ in the body of this paper to simplify the presentation, the normalized variant of the operators was used throughout in our formal proofs and prototype.

We define the following indistinguishability relations on UT histories, tagged values, and memory states:

Definition A.7. Let ℓ be a UT. The relations $=_\ell \subseteq \mathcal{H}^2$ and $\approx_\ell \subseteq \mathcal{T}^2$ are defined as follows:

$$\frac{}{[] =_\ell []} \qquad \frac{\ell \in h}{h :: t =_\ell h :: t'} \qquad \frac{t =_\ell t'}{h :: t =_\ell h :: t'}$$

$$\frac{\alpha =_\ell \alpha'}{\langle v, \alpha \rangle \approx_\ell \langle v, \alpha' \rangle} \qquad \frac{\alpha =_\ell \alpha' \quad \ell \in \mathsf{uts}(\alpha) \cap \mathsf{uts}(\alpha')}{\langle v, \alpha \rangle \approx_\ell \langle v', \alpha' \rangle.}$$

We lift $\approx_\ell$ to $(\mathrm{Var} \to \mathcal{T})^2$ by defining

$$m \approx_\ell m' \equiv \forall v \in \mathrm{Var}.\ m(v) \approx_\ell m'(v).$$

We prove a series of technical lemmata, of which the most important are:

Lemma A.8 (`equiv_ind_history`, `equiv_ind_tv`). *For any UT ℓ, the relations $=_\ell$ and $\approx_\ell$ are equivalence relations.*

Lemma A.9. *If $\langle v, \alpha \rangle \approx_\ell \langle v', \alpha' \rangle$ but $\langle v, \alpha \rangle \neq \langle v', \alpha' \rangle$, then $\ell \in \mathsf{uts}(\alpha) \cap \mathsf{uts}(\alpha')$.*

Lemma A.10. *If $\langle v, \alpha \rangle \approx_\ell \langle v', \alpha' \rangle$, then $\alpha =_\ell \alpha'$.*

Lemma A.11. *If $\langle v, \alpha \rangle \approx_\ell \langle v', \alpha' \rangle$ and $\beta =_\ell \beta'$, then $\langle v, \beta \mathbin{\hat{\cdot}} \alpha \rangle \approx_\ell \langle v', \beta' \mathbin{\hat{\cdot}} \alpha' \rangle$.*

Lemma A.12. *If $\alpha =_\ell \alpha'$ and $\ell \in \mathsf{uts}(\alpha) \cap \mathsf{uts}(\alpha')$, then for all v, v', β, β', we have $\langle v, \alpha \mathbin{\hat{\cdot}} \beta \rangle \approx_\ell \langle v', \alpha' \mathbin{\hat{\cdot}} \beta' \rangle$.*

Lemma A.13. *If $\alpha_1 =_\ell \alpha_1'$ and $\alpha_2 =_\ell \alpha_2'$, then $\alpha_1 \mathbin{\hat{\Cup}} \alpha_2 =_\ell \alpha_1' \mathbin{\hat{\Cup}} \alpha_2'$.*

Lemma A.14. $\mathsf{uts}(\alpha \mathbin{\hat{\Cup}} \beta) = \mathsf{uts}(\alpha \mathbin{\hat{;}} \beta) = \mathsf{uts}(\alpha) \cup \mathsf{uts}(\beta)$

Lemma A.15. *If $m \approx_\ell m'$ and $w \approx_\ell w'$, then $m(x := w) \approx_\ell m'(x := w')$.*

Lemma A.16. *If $m \approx_\ell m'$ and $\alpha =_\ell \alpha'$, then $\mathsf{sanitize}(m, b, \alpha) \approx_\ell \mathsf{sanitize}(m', b, \alpha')$.*

Lemma A.17. *If $\alpha =_\ell \alpha'$, then $\kappa(u, p, \alpha) \approx_\ell \kappa(u, p, \alpha')$.*

In general, the output value of κ can be influenced by user inputs. Consider the following code k_1, where u_1 and p_1 are arbitrary:

k_1 = `u = check x u1 p1; if x {z = a}; v = check z u1 p1; w = u == v.`

This program first checks `x`, assigns `a` to `z` if `x` is non-zero, and checks `z`; then, it stores the result of the comparison `u == v` in `w`. We claim that if `w` is 0 after executing k_1 on m_0, the initial value of `x` must be nonzero. The proof is by contraposition: if `x` is initially 0, then the *if* block is never executed; instead, the UTs of `x` are added to the UTs of `z`, which can be modified within the *if* block. Hence, when `z` is checked, it contains exactly the same UTs as `x` contained initially, and the values of `u` and `v` must be equal. Hence if `w` is 0, then `x` cannot have been 0 initially, i.e., `x` influences `w` through the use of `check`. Therefore, our definition of κ must ensure that, after executing k_1, `w` is tagged with the initial UT of `x`.

However, we can prove the additional property of κ from Section 4.6.2:

Theorem A.18 (`κ_accept`). *If $\kappa_c(u, p, \alpha) = \langle v', \alpha' \rangle$ (where κ_c is κ defined as in Section 4.6.2 using the single-UT enforcement oracle `c`) and $\ell \in \mathsf{uts}(\alpha')$, then $\mathsf{c}(u, p, \ell) = 1$.*

Example A.19. With the code k_1 introduced above, our definition gives $\kappa(\mathsf{u}_1, \mathsf{p}_1, [\{\ell_3\}]) = \langle 0, [] \rangle$ if $c(\mathsf{u}_1, \mathsf{p}_1, \ell_3) = 0$, and $\kappa(\mathsf{u}_1, \mathsf{p}_1, [\{\ell_3\}]) = \langle 1, [\{\ell_3\}] \rangle$ if $c(\mathsf{u}_1, \mathsf{p}_1, \ell_3) = 1$. In the former case, ℓ_3 is not allowed to be output, and `w` must contain 0 after executing k_1 irrespective of the value of `x`, as both calls to `check` return 0. Hence, `w` is not influenced by the value of `x`. In the latter case, ℓ_3 is allowed to be output, but the initial content of `y` may not be. In this case, the first `check` may yield $v_1 = 1$ and the second $v_2 = 0$, and we may learn that `x` was initially nonzero by observing that `w` contains 0. But since after executing k_1, the variable `w` now contains $\langle v_1 = v_2, [\{\ell_3\}] \rangle$, this information flow is correctly tagged by the UTs. In both cases, $\kappa(\mathsf{u}_1, \mathsf{p}_1, [\{\ell_3\}])$ is only tagged with ℓ_3 if $c(\mathsf{u}_1, \mathsf{p}_1, \ell_3) = 1$.

A.3 TTCWhile (`TTCWhile.thy`)

We first need to define a few properties of programs:

Definition A.20 (Termination, `terminates'`). We say that a code π *terminates* on a program counter history pc and a memory state m for a given choice of `c`, written `terminates'` $\pi\ pc\ m$ `c`, iff there exists pc' and m' such that $\pi, pc, m \dashrightarrow_c^* \varepsilon, pc', m'$.

Definition A.21 (`pops`). For any code π, we denote by $\mathsf{pops}(\pi)$ the number of `pop` statements that appear in π (excluding potential `pop` statements inside `while` blocks).

Definition A.22 (Well-formedness, `wf_code`). We say that a code π is *well-formed*, written `wf_code` π, iff it contains no `pop` statements inside its `while` blocks.

We prove that evaluating the same TTCWhile expression against two indistinguishable memory states results in indistinguishable tagged values:

Lemma A.23 (`eval_ind`). *If $m \approx_\ell m'$, $[\![e]\!](m) = w$ and $[\![e]\!](m') = w'$, then $w \approx_\ell w'$.*

We then show the following properties for TTCWhile programs:

Lemma A.24 (Determinism, `steps_deterministic`). *If*

$$\pi, pc_0, m \dashrightarrow_c^* \varepsilon, pc_1, m_1 \wedge \pi, pc_0, m \dashrightarrow_c^* \varepsilon, pc_2, m_2,$$

then $pc_1 = pc_2 \wedge m_1 = m_2$.

Lemma A.25 (Noninterference, `ni`). *If*

$$\begin{aligned} &\texttt{terminates'}\ \pi\ pc\ m_1\ \mathsf{c} \wedge \texttt{terminates'}\ \pi\ pc\ m_2\ \mathsf{c} \\ &\quad \wedge \texttt{wf\_code}\ \pi \wedge \mathsf{pops}(\pi) = |pc| \wedge m_1 \approx_\ell m_2 \\ &\quad \wedge \pi, pc, m_1 \dashrightarrow_c^* \varepsilon, pc_1, m_1' \wedge \pi, pc, m_2 \dashrightarrow_c^* \varepsilon, pc_2, m_2', \end{aligned}$$

then $m_1' \approx_\ell m_2'$.

A.4 TTC semantics (`TTC.thy`)

We prove our enforcement property in two steps. First, we fix a set of objects and assumptions about them (a *locale* in Isabelle parlance) and prove that, if these assumptions hold, the semantics of TTC guarantees correct enforcement. Second, we show that the assumptions in the locale hold for TTCWhile.

A.4.1 The locale `PL`. We fix a program transition relation $\dashrightarrow^*$ as above, a program π, an enforcer $\mathcal{E}$, a property P, an initial program counter history pc_0, and a null program ϵ.

Further, we define $S_{0,\mathrm{TTC}} = \{\texttt{step} = \textsc{Taint}\}, \{\}, (), (x \mapsto \langle 0, [] \rangle)$ and restrict acceptable input sequences to those where inputs occur at intervals larger than the maximal execution time of functions:

Definition A.26 (`wf_in_seq2`).

$$\begin{aligned} \texttt{wf\_in\_seq2}\ I \equiv \forall i \in \{1..|\pi(f)|\}, u, f, A, \tau, u', f', A', \tau'. \\ I_i = (u, f, A, \tau) \wedge I_{i+1} = (u', f', A', \tau') \\ \Rightarrow \tau' > \tau + \sum_{i=1}^{|\pi(f)|} \pi(f)_i.\texttt{time}. \end{aligned}$$

Finally, we assume:

(1) P is independent of past outputs and ltf-monotonic,
(2) $\mathcal{E}$ is an enforcer for P,
(3) For all f, $|\pi(f)| > 0$,
(4) For all f, $i \in \{0..|\pi(f)| - 1\}$, $\pi(f)_i.\texttt{time} > 0$,
(5) **Noninterference**: for all f, $i \in \{0..|\pi(f)| - 1\}$, m_1, m_2, pc_1, pc_2, m_1', and m_2',

$$\begin{aligned} &m_1 \approx_\ell m_2 \\ &\wedge\ \pi(f)_i.\mathsf{code}, pc_0, m_1 \dashrightarrow_c^* \epsilon, pc_1, m_1' \\ &\wedge\ \pi(f)_i.\mathsf{code}, pc_0, m_2 \dashrightarrow_c^* \epsilon, pc_2, m_2' \\ &\Rightarrow m_1' \approx_\ell m_2', \end{aligned}$$

(6) **Determinism**: for all $f, i \in \{0..|\pi(f)|-1\}$, m, pc_1, pc_2, m_1, and m_2,

$$\pi(f)_i.\mathsf{code}, pc_0, m \dashrightarrow_c^* \epsilon, pc_1, m_1 \\ \wedge\ \pi(f)_i.\mathsf{code}, pc_0, m \dashrightarrow_c^* \epsilon, pc_2, m_2 \\ \Rightarrow m_1 = m_2.$$

Under these assumptions, we show:

Lemma A.27 (approx). *If*

$$\mathtt{wf\_in\_seq'}\ I \quad S_0, I \overset{\sigma}{\underset{\pi,P}{\Longrightarrow}} S', I' \quad S' = (s_W, s_I, s_M, s_E),$$

then $\Delta_I(\sigma) \sqsubseteq_{\mathrm{TTC}} s_E$.

Lemma A.28 (happy). *If*

$$\mathtt{wf\_in\_seq'}\ I \quad S_0, I \overset{\sigma}{\underset{\pi,P}{\Longrightarrow}} (s_W, s_I, s_E, s_M), I',$$

then $s_E \in P$.

We finally prove our main theorem.

Theorem A.29 (PL.main). enforces P *holds.*

Proof. Using Lemma A.5 with $\sqsubseteq = \sqsubseteq_{\mathrm{TTC}}$, $t = (s_W, s_I, s_M, s_E) \mapsto s_E$, and Π the set of all policies that are enforceable, Itf-monotonic, and independent of past outputs.

(1) By definition of $\sqsubseteq_{\mathrm{TTC}}$ and Itf-monotonicity of policies in Π.
(2) By Lemma A.27.
(3) By Lemma A.28.

□

A.4.2 Refining the locale: TTCWhile_PL. In this section, $\dashrightarrow^*$ denotes the transition relation defined in Section A.3, programs are TTCWhile programs, pc_0 is [], and ϵ is ε. We now assume:

(1)–(5) As above,
(8) For all $f, i \in \{0..|\pi(f)|-1\}$,

$$\mathtt{wf\_code}(\pi(f)_i.\mathsf{code}) \wedge \mathtt{pops}(\pi(f)_i.\mathsf{code}) = 0.$$

We easily obtain:

Lemma A.30 (TTCWhile_PL ⊆ PL). *The assumptions of* TTCWhile_PL *entail those of* PL.

Proof. Assumptions (1)–(5) are identical.
Assumption (6) in PL follows from Lemma A.25 using (4) and (8).
Assumption (7) in PL follows from Lemma A.24. □

In particular, Theorem 4.9 holds when the programming language in the TTC semantics is instantiated with TTCWhile. Given that pop is not part of the language accessible to developers, this provides the desired correctness property:

Theorem 4.9. *Assume that for all f, $|\pi(f)| > 0$, and that every $\pi(f)_i$ has a strictly positive running time.*
If P is enforceable, Itf*-monotonic, and independent of past outputs, then A_{TTC} enforces the privacy policy $P \subseteq \mathbb{T}_{\Sigma_{\mathrm{IF}}}$.*

Proof. Immediate corollary of TTCWhile_PL.main. □

```
1  def foo():
2    y = 0
3    if secret == 1:
4      y = adv_input
5    return y
6
7  def bar():
8    y = 1
9    if check(adv_input,
10             me, "..."):
11     y = 0
12   return y
```

```
9  def set_adv_input(x):
10   adv_input = x
11
12 def set_secret(x):
13   secret = x
```

$P_{\mathtt{Alice}}$: "data derived from Alice's inputs shall only be output to Alice."

$P_{\mathtt{Eve}}$: "data derived from each of Eve's inputs shall be output at most once."

Steps: 1. Alice calls `set_secret` with `x = 1`; 2. Eve calls `set_adv_input` with `x = 0`; 3. Alice calls `foo`; 4. Eve calls `bar`. If `bar` returns 1, then `secret` is 1.

Figure 9: Leak caused by a policy dependent on past outputs

B POLICIES THAT DEPEND ON PAST OUTPUTS

Consider the code in Figure 9. Assume that the honest user Alice and the impersonated user Eve first call `set_secret` and `set_adv_input` to set `adv_input` and `secret` respectively. Eve has set her policy to allow only one output for each of her inputs, while Alice wants to prevent any other user from seeing her inputs. Clearly, Eve's policy depends on past outputs. Now, Alice calls `foo`, which outputs 1 to her (the output is allowed since both `adv_input` and `secret` are allowed to be output to Alice once). Finally, Eve can call `bar`. The check (l. 9-10) fails, since `adv_input` is not allowed to be output a second time, and `bar` returns 1 to Eve. But now, the adversary knows that `secret` was 1, since if the secret had been 0 the output in `foo` would not have involved any of Alice's input, thus allowing the check on l. 9-10 to succeed. The adversary was thus able to break $P_{\mathtt{Alice}}$ by choosing malicious code and $P_{\mathtt{Eve}}$. To prevent the previous leak, we must either adopt a coarser propagation of UTs when covering the implicit flow l. 4, or the UTs that affected the content of the trace must be permanently remembered. Both approaches risk an explosion in the number of UTs to be remember ('label creep').

C METRIC FIRST-ORDER TEMPORAL LOGIC

In this section, we formally describe the syntax and semantics of MFOTL. Our description is closest to [58], which studies MFOTL from the perspective of enforcement.

Definition C.1. MFOTL formulae over $\Sigma = (\mathbb{D}, \mathbb{A}, \iota)$ are defined by the grammar

$$\varphi ::= a(t_1, \ldots, t_{\iota(r)}) \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\ \varphi \\ \mid \bullet_I \varphi \mid \bigcirc_I \varphi \mid \varphi\ \mathsf{S}_I\ \varphi \mid \varphi\ \mathsf{U}_I\ \varphi,$$

where $t_1, \ldots, t_{a(r)} \in \mathrm{Var} \cup \mathbb{D}$, $a \in \mathbb{A}$, and I are intervals on $\mathbb{N}$.

We define the following as syntactic sugar: $\top := p \vee \neg p$, $\bot := \neg\top$, $\varphi \Rightarrow \psi := \neg\varphi \vee \psi$, and the operators "once" ($\blacklozenge_I\ \varphi := \top\ \mathsf{S}_I\ \varphi$), "eventually" ($\lozenge_I\ \varphi := \top\ \mathsf{U}_I\ \varphi$), "always" ($\square_I\ \varphi := \neg\lozenge_I \neg\varphi$), and "historically" ($\blacksquare_I\ \varphi := \neg\blacklozenge_I \neg\varphi$).

Satisfaction of an MFOTL formulae is defined over *valuations.*

Definition C.2. A *valuation* is a function $v : \mathrm{Var} \cup \mathbb{D} \to \mathbb{D}$ such that $v(d) = d$ for all $d \in \mathbb{D}$.

We write $v[x \mapsto d]$ for the valuation obtained from v by setting $v(x)$ to d. We can now define the semantics of MFOTL.

Definition C.3. Fix a signature Σ. Given $k \in \mathbb{N}$, a trace $\sigma = ((\tau_i, D_i))_{1 \le i \le k}$ over Σ, a timepoint $1 \le i \le |\sigma|$, a valuation v, and an MFOTL formula φ over Σ, we define the satisfaction relation $\models$ as follows:

$$
\begin{array}{ll}
v, i \models_\sigma a(t_1, ..., t_n) & \text{iff } (r, (v(t_1), ..., v(t_n))) \in D_i \\
v, i \models_\sigma \neg\varphi & \text{iff } v, i \not\models_\sigma \varphi \\
v, i \models_\sigma \exists x.\ \varphi & \text{iff } v[x \mapsto d], i \models_\sigma \varphi \text{ for } d \in \mathbb{D} \\
v, i \models_\sigma \varphi \vee \psi & \text{iff } v, i \models_\sigma \varphi \text{ or } v, i \models_\sigma \psi \\
v, i \models_\sigma \bullet_I \varphi & \text{iff } i > 1 \text{ and } v, i-1 \models_\sigma \varphi \text{ and } \tau_i - \tau_{i-1} \in I \\
v, i \models_\sigma \bigcirc_I \varphi & \text{iff } i+1 \le |\sigma| \text{ and } v, i+1 \models_\sigma \varphi, \text{ and } \tau_{i+1} - \tau_i \in I \\
v, i \models_\sigma \varphi\ \mathsf{S}_I\ \psi & \text{iff } v, j \models_\sigma \psi \text{ for some } j \le i, \tau_i - \tau_j \in I, \\
 & \text{and } v, k \models_\sigma \varphi \text{ for all } k,\ j < k \le i \\
v, i \models_\sigma \varphi\ \mathsf{U}_I\ \psi & \text{iff } v, j \models_\sigma \psi \text{ for some } |\sigma| \ge j \ge i, \tau_j - \tau_i \in I, \\
 & \text{and } v, k \models_\sigma \varphi \text{ for all } k,\ j > k \ge i
\end{array}
$$

$v, i \models_\varepsilon \varphi$.

Each MFOTL formula φ over Σ defines a policy $P_\varphi \subseteq \mathbb{T}_\Sigma$ as follows:

Definition C.4. The MFOTL formula φ over Σ defines the policy

$$P_\varphi = \{\sigma \in \mathbb{T}_\Sigma \mid \exists v.\ v, 1 \models_\sigma \varphi\}.$$

Sufficient enforceability conditions are established in [58].

Using the operators $\blacklozenge_I$ and $\Box$, the policies from Figure 1 can be formalized as follows:

$$\varphi_{P_0} = \top$$

$$\varphi_{P_1} = \Box\big[\forall f, u, p, o, i, f'.\ \mathsf{Out}(f, u, p, o) \wedge \mathsf{Itf}(i, o) \Rightarrow \blacklozenge_{[0,\infty)}\ \mathsf{In}(\text{“Alice”}, f', i) \Rightarrow p \neq \text{“Marketing”}\big]$$

$$\begin{aligned}\varphi_{P_2} = \Box\big[&\forall f, u, p, o, i, f'.\ \mathsf{Out}(f, u, p, o) \wedge \mathsf{Itf}(i, o) \\ &\Rightarrow \blacklozenge_{[0,\infty)}\ \mathsf{In}(\text{“Alice”}, f', i) \\ &\Rightarrow f' = \text{“add\_message”} \\ &\Rightarrow (p \neq \text{“Marketing”}) \\ &\quad \vee (p = \text{“Analytics”} \wedge u \neq \text{“trustedanalytics.com”}) \\ &\quad \vee (p \neq \text{“Service”} \wedge \neg(\blacklozenge_{[0,1\ \mathsf{week}]}\ \mathsf{In}(\text{“Alice”}, f', i)))\big]\end{aligned}$$

$$\varphi_{P_3} = \Box\big[\forall f, u, p, o, i.\ \mathsf{Out}(f, u, p, o) \wedge \mathsf{Itf}(i, o) \Rightarrow \blacklozenge_{[0,\infty)}\ \mathsf{In}(\text{“Alice”}, f', i) \Rightarrow u = \text{“Alice”}\big]$$

Generally, for any user v, we can express user policies of the form

$$\Box\big[\forall f, u, p, o, i, f'.\ \mathsf{Out}(f, u, p, o) \wedge \mathsf{Itf}(i, o) \Rightarrow \blacklozenge_{[0,\infty)}\ \mathsf{In}(\mathsf{v}, f', i) \Rightarrow \phi\big]$$

where ϕ expresses the condition under which v consents that the data she input to function f' can be output to user u by function f for purpose p.

D CONVERSION OF JACQUELINE IFC POLICIES

D.1 Conf

In the **Conf** application from [77], the authors enforce the following restrictions:

- Email of users is only visible to the chair or user itself;
- Author of papers is visible if the phase of the submission process is 'final', or if the current user is the author, a chair, a PC member, and there is no conflict with the paper;
- Coauthors, review assignments, paper versions, reviews, comments are visible only if the current user is the author, a chair, a PC member, and there is no conflict with the paper.

We assume that the phase is not final, that there are no conflicts, and that the user identifiers are of the form "0", "1",... where "0" is a chair, "1",..., "10" are PC members, and "11",... are regular users. The following TTC policy allows for the same policies:

$$\begin{aligned}&\varphi_{P^{\langle u\rangle}_{\mathbf{Conf}}} \\ &= \forall f', u, p, o, l, f, a.\ \mathsf{Out}(f', u, p, o) \wedge \mathsf{Itf}(l, o) \wedge \blacklozenge\,\mathsf{In}(f, a, l) \\ &\quad \Rightarrow [((f = \text{“submit\_view”} \vee f = \text{“assign\_reviewer”} \\ &\qquad \vee f = \text{“paper\_view”} \vee f = \text{“submit\_review\_view”} \\ &\qquad \vee f = \text{“submit\_comment\_view”}))) \\ &\quad \wedge (u = \langle u\rangle \vee u = \text{“0”} \vee (u = \text{“1”} \vee \cdots \vee u = \text{“10”})) \\ &\quad \vee f = \text{“register”} \wedge a = \text{“email”} \wedge (u = \langle u\rangle \vee u = \text{“0”})]\,.\end{aligned}$$

D.2 HIPAA

In the **HIPAA** application from [77], the authors enforce the following restrictions:

- Data from model `Individual` is visible to the individual;
- Location from model `HospitalVisit` is visible to the patient, hospital, and users with profile type 6;
- Patient ID from model `Treatment` is visible to the patient and prescribing entity;
- Patient ID from model `Diagnosis` is visible to the patient and recognizing entity;
- Shared information from model `BusinessAssociateAgreement` is visible to the covered entity and business associate;
- Standard, first party, second party, and purpose from model `Transaction` is visible to the first party and second party;
- E-mail from model `UserProfile` is visible to the user itself.

We perform tests with `Individual` only, hence we consider:

$$\begin{aligned}&\varphi_{P^{\langle u\rangle}_{\mathbf{HIPAA}}} \\ &= \forall f', u, p, o, l, f, a.\ \mathsf{Out}(f', u, p, o) \wedge \mathsf{Itf}(l, o) \wedge \blacklozenge\,\mathsf{In}(f, a, l) \\ &\quad \Rightarrow f = \text{“Individual”} \wedge u = \langle u\rangle\,.\end{aligned}$$