User-Controlled Privacy: Taint, Track, and Control

François Hublet  
ETH Zürich  
francois.hublet@inf.ethz.ch

David Basin  
ETH Zürich  
basin@inf.ethz.ch

Srđan Krstić  
ETH Zürich  
srdan.krstic@inf.ethz.ch

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

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

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

Privacy by Design (PbD), the principle of “design[ing] and develop[ing] products with a built-in ability to demonstrate compliance” [74], offers a promising path to improving the status quo. With Privacy by Design, privacy policies are specified that define what an application shall do with data, when, and for which purposes. Incentives for practitioners to adopt PbD include developing or maintaining a competitive advantage in privacy-critical markets [51] and avoiding the costs resulting from major privacy breaches and non-compliance [42, 66]. Moreover, Article 25 GDPR makes it obligatory for data controllers to use “state-of-the-art” techniques to implement data protection principles [14]. Whenever technologies for ensuring compliance are available “at a reasonable price,” controllers are legally obliged to use these (or similar) technologies [55].

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 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) [53, 74], leveraging the similarity between the two approaches: both PbD and IFC must restrict data flows and guarantee that, by default, the protection enjoyed by data items extends to any other data derived from them. But despite these similarities, existing IFC designs cannot be directly reused for PbD, since the assumptions made by these designs do not match the requirements of privacy laws. In particular, the following requirements stand out:

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

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

[R3] Time-dependent policies. The policy language must allow users to define restrictions that apply only for a specific time period. For example, the GDPR has the notion of a storage period [1, Rec. 45] during which data can be used and retained.

Policy $P_2$ in Figure 1 exemplifies these requirements: user Alice demands that her posts in a microblogging platform are never used for marketing purposes, that only a single trusted third-party can be sent information about the posts for analytics purposes, and that after a week, the messages can only be used for service purposes. These limitations extend to all data derived from her posts. The policy is user-specific (only concerning Alice’s inputs), per-input (only applying to Alice’s posts), and time-dependent (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...
on information flow in terms of the data model [5, 18, 21, 25, 45, 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 [21, 45, 77, 78], developers can build custom user interfaces where users can select from different predefined policies, but 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. Indeed, first-order temporal policies [R2–3] are well-supported by those runtime enforcement (RE) techniques developed within the runtime verification community [56, 69]. These techniques consider timestamps event traces modeling 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 [7, 52] 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 [56] 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):

1. **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 emitting an event (2). The input and UT are sent to the interpreter (3).
2. **Track:** The semantics implemented by the interpreter ensures that at all times, 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.
3. **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).

This work focuses on consent-based PbD. As consent is at the core of the GDPR’s design [68] and omnipresent in web services [15], tackling consent-based PbD is an important first step towards more comprehensive enforcement of legal requirements.

**Figure 2: Taint, Track, and Control**

---

François Hublet, David Basin, and Srđan Krstić

Proceedings on Privacy Enhancing Technologies 2024(1)
def generate_ad(posts): # Returns personalized ads
    ad = generate_ad(posts)
    return ad

def user_timeline(username):
    posts = generate_ad(posts)
    return posts

@route('<Username>')</def)
    # Displays a user's posts
    return [post for post in posts if check('Service', post)]

filter_check(posts):
    posts2 = []
    for post in posts:
        posts2 == [post if check('Service', posts) else []
    return posts2

filter_check(posts):
    posts2 = []
    for post in posts:
        posts2 == [post if check('Marketing', posts) else []
    return posts2

Example 2.1. Assume that at the timepoint $t_0$, the user Alice calls user_timeline (l. 5) with username Bob. Her input has label $i(Alice, user_timeline, [username := Bob], t_0)$.

Performing a $\bullet$ transition, the function computes the list of Bob’s posts and the corresponding ad (l. 6–11). It then produces four outputs (l. 12–16) with respective labels

$\bullet o(view_timeline, trustedanalytics.com, Analytics, Bob, t_1)$

$\bullet o(view_timeline, evilanalytics.com, Analytics, Bob, t_2)$

$\bullet o(view_timeline, Alice, Service, posts, t_3)$

$\bullet o(view_timeline, Alice, Marketing, ad, t_4)$

where $posts$ and $ad$ stand for the value of the respective variable after line 16, and $t_0 < t_1 < t_2 < t_3 < t_4$.

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

Figure 3: Excerpt from the code of Minitwit

Figure 4: Excerpt from the code of Minitwit, II
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 $\Sigma = (\mathcal{D}, \mathcal{A}, i)$ where $\mathcal{D} \supseteq \mathbb{N}$ is an infinite set of constant symbols, $\mathcal{A}$ is a finite set of actions, and $i: \mathcal{A} \rightarrow \mathbb{N}$ is an arity function. An event over the signature $\Sigma$ is an expression of the form $a(d_1, \ldots, d_{i(a)})$, where $a \in \mathcal{A}$ and $d_i \in \mathcal{D}$, 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 $\sigma = (t_i, D_i)_{1 \leq i \leq k}$ where the $t_i$ 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 $\Sigma$ is denoted by $\mathcal{E}_\Sigma$ (resp. $\mathcal{T}_\Sigma$). Finally, a policy $P$ is a subset $P \subseteq \mathcal{T}_\Sigma$. 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, $\text{Obs} \subseteq \mathcal{A}$), or can also be suppressed at the PEP (suppressible actions, $\text{Sup} \subseteq \mathcal{A}$).1 An enforcer for a policy $P$ is a function that takes a trace $\sigma$ 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 $\sigma$ and $(t, D)$ such that $\sigma \cdot (t, D)$ is a trace and $\sigma$ complies with $P$, an enforcer for $P$ returns a subset $D' \subseteq D$ of suppressible events such that $\sigma \cdot (t, D \setminus D')$ 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 suppressible, the policy “the system shall perform no output to Alice” is enforceable; a corresponding enforcer would simply suppress any tentative output to Alice. On the other hand, the policy “the system shall perform an output to Alice” is not enforceable, 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 $\pi$, provided by developers, which encodes the application’s functionality. The enforcer component provides an enforcer $E$ for an enforceable policy $P$ over some signature $\Sigma$. To support user-defined policies, the privacy policy $P$ is assumed to be of the form $\bigcap_{u \in \mathcal{U}} P_u$, where $\mathcal{U}$ is the set of users and for all $u \in \mathcal{U}$, $P_u$ denotes the individual policy of user $u$. The intersection $\bigcap_{u \in \mathcal{U}} P_u$ is the property expressing the simultaneous compliance with all $(P_u)_{u \in \mathcal{U}}$.

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’ privacy policies. 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 suppressible. We obtain a family of transition relations for the entire application of the form $(\rightarrow_{\pi, P})$, where $\pi$ ranges over the set of programs and $P$ over the set of policies. We write $\Rightarrow_{\pi, P}$ for the relation that we obtain as in Section 2.1.2 by setting $\rightarrow$ to $\rightarrow_{\pi, P}$.

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 $\pi$. 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 $\pi$ 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 assume 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 $\Sigma_{\text{IO}} = (\mathcal{D}, \{\text{in, out}\}, \{(\text{in} \leftrightarrow 4, \text{out} \leftrightarrow 4)\})$ (see also Figure 5). For each timestamp $t_i$ when a transition $S \rightarrow S'$ occurs, we generate a set of events $E_i$ over $\Sigma_{\text{IO}}$ in the following way:
We now define information-flow traces $S, I$. The complete input-output (IO) trace is obtained by considering the sequence of transitions from $S$ to $O$. The signature is $\text{Itf}(I, O)$.

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

\[
((\tau_0, \text{In}(\text{Alice, timeline, username, \{\text{Bob}\}))),\]
\[
(\tau_1, \text{Out}(\text{timeline, trustedanalytics, Analytics, Bob})),\]
\[
(\tau_2, \text{Out}(\text{timeline, evinalalytics, Analytics, Bob})),\]
\[
(\tau_3, \text{Out}(\text{timeline, Alice, Service, posts})),\]
\[
(\tau_4, \text{Out}(\text{timeline, Alice, Marketing, ad})).\]

In the following, we label the relation $\Rightarrow_{\pi \cdot P}$ introduced in Section 2.2.2 with the IO trace produced by the execution. We write $S, I \Rightarrow_{\pi \cdot P} S', I'$ if $S, I \Rightarrow S', I'$ and $\pi$ 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 [46] capturing the influence of an input on an output. The signature is $\Sigma_{\text{Itf}} = (D \cup D^2, \{\text{In}, \text{Out}, \text{Itf}, \text{In} \Rightarrow 3, \text{Out} \Rightarrow 4, \text{Itf} \Rightarrow 2\})$ (cf. Figure 5).

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 $\tau$ 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 In($u, f, a, d_i$) in the IO trace at timestamp $\tau$ becomes $\text{In}(u, f, (\tau, a))$ in the information-flow trace;
- Each Out($f, u, p, d$) in the IO trace at timestamp $\tau$ becomes $\text{Out}(f, u, p, \tau)$ 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:

\[
\{(\tau_0, \text{In}(\text{Alice, timeline, username, \{\text{Bob}\}))),\]
\[
(\tau_1, \text{Out}(\text{timeline, trustedanalytics, Analytics, \{\text{Bob}\}))),\]
\[
(\tau_2, \text{Out}(\text{timeline, evinalalytics, Analytics, \{\text{Bob}\}))),\]
\[
(\tau_3, \text{Out}(\text{timeline, Alice, Service, posts})),\]
\[
(\tau_4, \text{Out}(\text{timeline, Alice, Marketing, ad})).\]

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 $\text{Itf}(I, O)$, where $I$ is a UT and $O$ an output identifier (i.e., an output timestamp). Informally, $\text{Itf}(I, O)$ 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 allows us to capture (non)interference [46] 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 $\tau_1$. Consider first the application without proactive checks (Figure 3). At timestamp $\tau_0$, user_timeline retrieves Bob’s posts in the variable posts. Two inputs have influenced the content of posts: the one marked by $\tau_1$, and Alice’s latest input to user_timeline (with value username = “Bob”), which has some UT $\tau_0$. Thus, the information-flow trace must contain $\text{Itf}(\tau_1, \tau_3)$ and $\text{Itf}(\tau_0, \tau_3)$ at timestamp $\tau_3$. Further, username is sent to the two analytics third-parties, which receive the outputs with identifiers $\tau_1$ and $\tau_2$. Through generate_ad, the two inputs that influenced posts also influence ad, which is output with the identifier $\tau_4$. Hence, we have the following information-flow trace:

\[
((\tau_0, \{\ldots\})),\]
\[
(\tau_1, \ldots, \text{Itf}(\tau_0, \text{username})),\]
\[
(\tau_2, \ldots, \text{Itf}(\tau_0, \text{username})),\]
\[
(\tau_3, \ldots, \text{Itf}(\tau_0, \text{username}), \text{Itf}(\tau_0, \text{text})),\]
\[
(\tau_4, \ldots, \text{Itf}(\tau_0, \text{username})).\]

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 $(\tau, a) \rightarrow_{\pi \cdot P \cdot I} \tau_1$, 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, $\pi$ a program, and $P$ a policy.
- Assume that we execute the application defined by $\rightarrow_{\pi \cdot P}$ on $I$. Event $\text{Itf}((\tau, a), \tau_1)$ is in the information-flow trace at timestamp $\tau_1$ if, and only if, $(\tau, a) \rightarrow_{\pi \cdot P \cdot I} \tau_1$.

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

Our Itf event plays a role analogous to the leak modality of SecTL [32] by capturing a hyperproperty [23]: 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 this trace 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 \( \mathcal{I}_{{\Sigma}P} \). We say that the online application \( A = (S, S_0, \rightarrow_{s,P}) \) enforces the privacy policy \( P \subseteq \mathcal{I}_{{\Sigma}P} \) if for any input sequence \( I \) and trace \( \sigma \), we have \( \text{trace}_{s,P}(I, \sigma) \in P \). 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 \( \text{Obs} = \{ \text{In}, \text{Iff} \} \), \( \text{Sup} = \{ \text{Out} \} \).

Our privacy policy definition addresses [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 [7, 19] provides the temporal operators \( \varphi \) (‘once in the past within time interval \( I \)’) and \( \Xi \) (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 [36], graphical representations [62], or natural language generation [79], and might benefit from a trans-disciplinary 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 Iff 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 Iff 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 = Bob from user Alice querying function user_timeline. It tags it with the UT \( i = (t_0, \text{username}) \) 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 Iff 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 3 with proactive checks is shown in Figure 4. Proactive enforcer calls are performed via the primitive check. If the content of posts cannot be used for marketing, a non-personalized ad is 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
\[
\pi(f) = ((\text{code}_i, \text{out}_y, \text{out}_u, \text{out}_p, \text{time}_i))_{1 \leq i \leq k}.
\]
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 \( \text{code}_i \) followed by a single output statement that attempts to output the content of variable \( \text{out}_y \) to user \( \text{out}_u \) for purpose \( \text{out}_p \). The complete source code of the function is obtained by concatenating the \( (\text{code}_i)_{1 \leq i \leq k} \) and their respective output statements. Each \( \text{out}_u \) can take any value in \( U \) or the special value \( \varepsilon \) denoting the user that performed the latest input. The integer \( \text{time}_i \in \mathbb{N} \) is the fixed duration of executing \( \text{code}_i \). In the following, the various components of \( \pi(f) \) will be denoted by \( \pi(f)_i\text{, code,} \ldots, \pi(f)_i\text{, time} \) 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 3 can be decomposed as follows, where \( \epsilon \) denotes an empty source code:
\[
\pi(f) = ((\text{code}_1 \ldots 15), \text{username}, \text{trustedanalytics.com, Analytics}),
\]
\[
(\epsilon, \text{username}, \text{evilanalytics.com, Analytics}),
\]
\[
(\epsilon, \text{posts, me, Service}),
\]
\[
(\epsilon, \text{ad, me, Marketing}).
\]

To simplify the presentation, we consider non-IO source code written in a Turing-complete language called TTCWhile that manipulates integer values. We fix a set \( \Omega \) of binary operator symbols and a mapping \( \Omega: \Omega \to (\mathbb{D} \times \mathbb{D} \to \mathbb{D}) \) interpreting these symbols. For illustrative purposes, we assume that \( \Omega \) contains at least two symbols \( \ominus \) and \( \oplus \) such that \( \Omega(\ominus) = (x, y) \mapsto x - y \) and \( \Omega(\oplus) = (x, y) \mapsto x + y \). The syntax of TTCWhile programs is given by the following grammar:
\[
e := c | x | e @ e
\]
\[
s := x = e \mid \text{while } x \{b\} | x = \text{check } y \{u|me\} p
\]
\[
b := s | b; b | e
\]
where \( c \in \mathbb{D} \) is a constant, \( @ \in \Omega \) a binary operator symbol, \( x \) a variable name, \( u \) a user name, and \( p \) a purpose.

A source code, represented by the non-terminal \( b \) ("block") is a (possibly empty) sequence of statements. Statements include assignments of expressions \( (e = e) \), while loops \( (\text{while } x \{b\}) \), and the special check instruction \( (x = \text{check } y \{p\}) \). Expressions can be constructed from constants \( (c) \), variables \( (x) \), and binary operators \( (e @ e) \). The instruction \( x = \text{check } y \{p\} \) 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 \( \text{me} \). From assignments and while loops, we define if statements as syntactic sugar, writing \( \text{if } x \{b\} \) as shorthand for \( \text{if } x \{b\} \{\text{while } x \{b\}; \text{true}\} \).

For the rest of this section, we fix a TTCWhile program \( \pi \), an enforceable policy \( P \subseteq \mathbb{T}_\Sigma \), and choose an enforcer \( E \) 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 \( w = (v, \alpha) \) of a value \( v \in \mathbb{D} \) and a finite list \( \alpha = \{\{f_1, \ldots, f_{k_1}\}, \ldots, \{f_1, \ldots, f_{k_l}\}\} \) of finite sets of UTs. The list \( \alpha \) is called a UT history. The UT history \( \alpha \) 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 \( f_i \);
2. The UTs in the first set, i.e., \( \{f_1, \ldots, f_{k_1}\} \), are part of the history of \( x \) independently of any input;
3. For any \( i > 1 \), the content of the \( i \)-th set of the history, \( \{f_1, \ldots, f_{k_i}\} \), 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 \( \{1, \{f_1\}, \{f_2, f_3\}\} \). The value of \( x \) is 1. The history indicates that:

1. Inputs with UTs \( f_1, f_2 \), and \( f_3 \) may have influenced \( x \)’s value;
2. No input has influenced that \( f_1 \) tags \( x \);
3. The input with UT \( f_2 \) has influenced that \( f_2, f_3 \) tag \( x \).

In practice, such a history can be obtained, e.g., by executing the code \( x = 10; \text{if } y \{x = u + v\} \) when \( u, v \) and \( v \) contain inputs \( \{1, \{f_1\}\}, \{0, \{f_2\}\} \), and \( \{1, \{f_3\}\} \) respectively. In this case:

1. The values of the inputs \( y, u, v \) with UTs \( f_2, f_2, f_3 \) respectively have all influenced the final value of \( x \);
2. The value of the input \( y \) with UT \( f_1 \) has influenced the final value of \( x \) irrespective of the value of any other input;
3. The value of the input \( y \) with UT \( f_1 \) has influenced that \( u, v \) (with UTs \( f_2 \) and \( f_3 \)) influence \( x \): if the \( i \)-th block is executed, there is influence of \( u, v \) on \( x \), whereas there is no such influence if the \( i \)-th block is not executed.

When property (1) above holds, the Itf events resulting from outputting \( s \) must all be of the form \( \text{Itf}(f_i, o) \) for some \( f_i \) in \( \alpha \). 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 $S_{\text{TTC}} = S_W \times S_I \times S_E \times S_M$, where $S_W$, $S_I$, $S_E$, and $S_M$ 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 $\sigma \in \mathcal{T}_E = S_E$ that contains at least as many ltl 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

$$\begin{align*}
\tau' &= \tau + \pi(f)_1.t \text{time} \\
\mathcal{E}_\tau' &= \mathcal{E}_\tau \cdot \tau, \{\ln(u, f, (\tau, a_1))\}_{1 \leq k} \\
\{\text{step} = \text{Taint}\}, s_I, s_E, s_M &\rightarrow \{u = u, f = f, t = \tau', i = 1\} \\
\{\text{step} = \text{Track}\}, u = u, f = f, t = \tau', i = 1 &\rightarrow \text{init}((f, u, s_E, \tau', t), s_E, \mathcal{E}_\tau, a_1, \delta, a_1, \delta, \ldots, a_1, \delta).
\end{align*}$$

This uses the three auxiliary functions:

$$\begin{align*}
\text{init}(f, u, s_E, \tau', t) &= \{\text{code} = \pi(f), \text{code, pc} = \varnothing, u = u, c = c_E\} \\
c_{f, u, \sigma, r, t}(u', p) &= \mathcal{E}(\pi, c_E, \{\text{Out}(f, \text{usv}(u, u'), p, t')\} | | (f, t') | | = \varnothing) \\
\text{usv}(u, u') &= \text{if } u' = u \text{ then } u \text{ else } u'.
\end{align*}$$

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 $\tau$ are read in the wrapper’s state. Moreover, the timestamp $\tau$ is incremented by $\pi(f)_1.t \text{time}$ to obtain the timestamp $\tau' = \tau + \pi(f)_1.t \text{time}$ of the next tentative output, and the field $1$, 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 $\text{init}$: its field code receives the source code $\pi(f), \text{code, pc} = \varnothing$; its program counter history, discussed in the next section, is initialized to $\varnothing$; the constant $u = u, c = c_E$; finally, the single-UT enforcement oracle $c$, which will be used to execute check instructions in the Track phase, is set to $c_E$. The function $c$ takes as arguments a UT $f$, a user $u$, and a purpose $p$: it returns 1 if one can perform an output to $u$ for purpose $p$ at timestamp $t'$ with a value influenced by $f$ without violating $P$. To detect violations, $c$ calls the enforcer $E$. The auxiliary function usv makes it possible to refer to the caller user $u$ in $c$ using the special constant $\text{me}$. Finally, the arguments $a_i$ are set in memory, each tagged with a fresh UT $(\tau, a_i)$.

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 6. The five rules assign, while_true, while_false, pop, and check (Figure 6b) are of the form $\pi, pc, m \rightarrow \pi', pc', m'$, where $\pi$ 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 $\llbracket v \rrbracket(m) = (w, a)$ (Figure 6a) 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 $\llbracket e_1 \rrbracket(m) = (v_1, a_1)$ and $\llbracket e_2 \rrbracket(m) = (v_2, a_2)$; then, a tagged value with value $\Omega(\oplus)(v_1, v_2)$...
and UT history $a_1 \cup a_2$ is computed, where $\cup$ is defined by

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

By this definition, the UT history of $[e_1 \varepsilon e_2](m)$ contains all UTs in the histories of $[e_1](m)$ and $[e_2](m)$. This reflects that the inputs that have influenced the evaluation $e_1 \varepsilon e_2$ are those that have influenced the evaluation of $e_1$ or $e_2$. Moreover, the relative order of the UTs within $a_1$ and $a_2$ respectively is preserved in $a_1 \cup a_2$.

The assign rule defines the semantics of assignments $x \leftarrow e$. First, $[e]$ is evaluated to some $(v, a)$. Then, $m(x) \leftarrow e$ receives a new tagged value with value $a$ and UT history $a \cdot e$, where all $(p, c)$ concatenates all UT histories in $pc$ (see Figure 6c). 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 influenced the value of $m(x)$ are those that influenced the result of evaluating $v$ or the control flow. Prepending of all $(p, c)$ to $a$ records that the presence of the UTs from $a$ in $m(x)$ depends on the inputs with UTs in all $(p, c)$, which have influenced the control flow.

The while_true and while_false rules define the semantics of loops while $x \{b\}$. If $m(x) = (v, a)$ and $v = 0$, then while_true is applicable. This rule adds $b$ at the front of the code to be executed and prepends $a$ to $pc$, recording that the inputs corresponding to the UTs in $a$ now influence the control flow. To cover any implicit flows that the loop’s execution can cause, while_true additionally adds the UTs $a$ to $m(a)$ for all variables $a$ that may be modified by $b$.

The set of such variables is computed using the auxiliary function $lhs$ (see Figure 6c). Finally, a special pop instruction is inserted after while. Its semantics is defined by the pop rule, which 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 $k_0 = \text{while } x \{ z = a + b; x = 0 \}$. Consider the initial memory state $m_0 = \{ a \mapsto (3, \{ f_1 \}), b \mapsto (2, \{ f_2 \}), x \mapsto (1, \{ f_3 \}) \}$. Executing $k_0$ on $m_0$ yields:

- $k_0: \{ a \mapsto (3, \{ f_1 \}), b \mapsto (2, \{ f_2 \}), x \mapsto (1, \{ f_3 \}), z \mapsto (0, \{ \}) \}$. Executing $k_0$ on $m_0$ yields:

$$k_0: \{ a \mapsto (3, \{ f_1 \}), b \mapsto (2, \{ f_2 \}), x \mapsto (1, \{ f_3 \}), z \mapsto (0, \{ \}) \}$$

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

6.4.2 Checks. The check rule in Figure 6c defines the semantics of $x \leftarrow \text{check } y u p$. It uses the function $\kappa$ from Figure 6c, calling it with the arguments $(u, p, c)$ where $c$ is the UT history of $m(x)$. The function $\kappa$ is defined so that this call returns 1 if, and only if, $c(u, p, \ell) = 1$ for every UT $\ell$ contained in $c$, where $c$ is the single-UT enforcement oracle defined in the previous section. In other words, $ \kappa(u, p, c) = 1$ if all inputs that have influenced $y$ can be output to $p$ for purpose $p$. This matches the intuitive semantics of check.

While defining the value of $\kappa(u, p, c)$ using $c(u, p, \ell)$ is easy, choosing the right definition for the UT history of $\kappa(u, p, c)$ is more difficult. This is because this definition must both account for all information flows between user inputs and the $c(u, p, \ell)$ 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 $x$ 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 $x = \text{check } y u p$, 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 $c(u, p, c) = 0$, 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 $\kappa(u, p, c)$ to be tagged by any UTs for which $c(u, p, c)$ returns 0. Formally:

$$\forall \ell, \alpha, \beta, \ell, \kappa(u, p, \alpha) = (\alpha, \beta) \wedge \ell \in \text{uts}(\beta) \Rightarrow c(u, p, \ell) = 1$$

where $\text{uts}(\{ s_1, \ldots, s_k \}) = s_1 \cup \ldots \cup s_k$.

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

- Call $c(u, p, \ell)$ for every UT $\ell$ in $c$ 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 $c(u, p, \ell)$ 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 $\kappa$ in Figure 6c formalizes this.

Example 4.6. Consider again the tagged value $\{ 1, \{ f_1 \}, \{ f_2 \} \}$. Fix some $u_0$ and $p_0$, and let $\alpha = \{ f_1 \}, \{ f_2 \}$.

If $c(u_0, p_0, f_1) = 0$, then we have $\kappa(u_0, p_0, c) = (0, \{ \})$: the first check performed is on $f_1$, which immediately fails. By invariant (2) of UT histories (see Section 4.3), we know that the presence of $f_1$ in the first set of $\alpha$ has not been influenced by the value of any input. Hence, the output of $\kappa(u_0, p_0, c)$ does not depend on the value of any input and the empty UT history in $\{ 0, \{ \} \}$ correctly accounts for all information flows (a). Moreover, since the history is empty, condition (b) is trivially fulfilled.

If $c(u_0, p_0, f_2) = 1$, then $\kappa(u_0, p_0, c) = (c(u_0, p_0, f_2) = 1) \wedge (c(u_0, p_0, f_2) = 1)$. The first call of $c(u_0, p_0, \cdot)$ on $f_1$ is successful, hence the value of $\kappa(u_0, p_0, c)$ depends on the output of $c(u_0, p_0, \cdot)$ on $f_2$. The content of $\kappa(u_0, p_0, c)$ now depends on the value of the input with UT $f_1$ (since this value influences the presence of $f_1$ in the history), which is indicated by having $f_1$ in the history $\{ f_1 \}$ (a). But since $c(u_0, p_0, f_1) = 1$, (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.

6.4.3 Track Rule. The rule for the Track phase is

$$s_j \text{code, } \{ \} s_M \rightarrow^* e, p', c' s_M' \quad \text{TRACK,}$$

where $\rightarrow^*$ denotes the transitive closure of $\rightarrow$. After executing the code in $s_j \text{code}$ (1), the wrapper moves the system into a Control state using a •-transition, updating the memory state (2).
We now state and discuss sufficient conditions on $P$ where a control phase is not processed, and the output is either performed or suppressed, with the trace approximation being updated accordingly (Step 5 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 $nextW$ and $nextI$ (Step 5, not shown in Figure 2).

### 4.7 Control

The rule defining the semantics of the CONTROL phase is

\[
\begin{align*}
\sigma' &= \pi(f), out_u = true & \text{if} & \pi(f), out_u = true \\
p &= \pi(f), out_p = true & \text{if} & \pi(f), out_p = true \\
b = E(s, (u, N(f, u', p, r, a))) &= E(s, (u, N(f, u', p, r, a))) \\
s_f = E(s', (u, N(f, u', p, r, a))) &= E(s', (u, N(f, u', p, r, a))) \\
\{\text{control} = true, u = f, r = t, i = 1\} & & & \text{if} & i = |\pi(f)| \\
\{\text{track} = true, u = f, r = t, i = i + 1\} & & & \text{otherwise}
\end{align*}
\]

where $N(f, u, p, r, a) = \{\text{Out}, (f, u, p, r)\} \cup \{\text{If}, (f, r) | f \in uts(a)\}$

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 $N(f, u', p, r, a)$ 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 $b = 0$ (output complies with $P$) or $b \neq 0$ (output violates $P$), the output is either performed or suppressed, with the trace approximation being 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 $nextW$ and $nextI$ (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) IfT-monotonicity and (2) independence of past outputs.

1. We say that a policy $P$ is $a$-monotonic for some action $a \in A$ if, for any trace $\sigma$ that violates $P$, adding additional events in $\sigma$ cannot restore compliance with $P$. Formally, $P$ is $a$-monotonic iff for all $\sigma = ((\sigma_0, D_0)), and $\sigma' = ((\sigma', D'))$, we have

\[
\forall i. D_i \subseteq D'_i \land D'_i - D_i \subseteq \{a(\bar{a}) | \bar{a} \in D''(a)\} \land \sigma \notin P \Rightarrow \sigma' \notin P.
\]

Example 4.7. The policies in Figure 1 are IfT-monotonic, as they are of the form $\Phi_\psi = \exists \psi(i, o, p, r, a) Out (f, u, p, o) \land \text{IfT} (i, o, \Rightarrow \psi)$, where $\psi$ contains no IfT event.

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

2. We say that a policy $P$ is independent of past outputs if compliance with $P$ does only depend on past In events, but not on past Out and IfT 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

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

If $P$ is enforceable, IfT-monotonic, and independent of past outputs, and the online application $A_{TTC} = (S_{TTC}, S_{TTC}, \rightarrow_{\pi, P})$ terminates, then $A_{TTC}$ enforces the privacy policy $P \subseteq T_{\Sigma P}$.

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 3, the HTML representations of a sequence of events are incrementally generated and added to the page 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 only the case if the enforcer’s verdict on a UT history $\alpha \not\parallel \beta$ is no stricter than the conjunction of its verdicts on $\alpha$ and $\beta$ 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 $\Phi_\psi$, 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/HOL proof assistant. The formalization, which has approximately 4,300 lines of Isar code, is openly available [57]. More details are provided in Appendix A.

### 5 IMPLEMENTATION

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

WebTTC features a Python-like language called PythonTTC that extends TTCWhile with support for basic web programming primitives. PythonTTC’s syntax is presented in Figure 7. Figures 3 and 4 show examples of PythonTTC code. Beyond standard Python features (functions, dictionaries, lists, sets, l. 1–8 and 9–13), PythonTTC

\[
\begin{align*}
\text{prog} &::= \text{fun} \ldots \text{fun} \ldots \text{fun} \ldots \text{fun} \ldots \text{fun} \ldots \\
\text{fun} &::= \text{@route}(\text{str}) \text{ def} (\text{id}, \ldots, \text{id}) \text{ block} \\
\text{block} &::= \text{stmt} \ldots \text{stmt} \\
\text{stmt} &::= \text{if exp} \text{ block} \ldots \text{else block} \\
\text{while exp} \text{ block} &::= \text{for id in exp block} \\
\text{return exp} \text{ block} &::= \text{send(const, const, exp)} \ldots \text{check(const, const, exp)} \ldots \text{check(const, const, exp)} \ldots \text{check(const, const, exp)} \ldots
\end{align*}
\]
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 3), 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 EnFoly [56] for enforcement. WebTTC consists of approximately 3,500 lines of Python code. We provide a ready-to-use image [58] containing all artifacts and tests.

The framework discharges our model’s assumptions as follows:

1. The assumptions about the privacy policies (independence of past outputs, monotonicity, enforceability) are verified by restricting policies to a ‘safe’ syntactic fragment of MFOTL on which those requirements are fulfilled [7, 56].

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 function’s 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., [44]) 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] implementation 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 3. 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 implement 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 code 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
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 \( P_u = P_{u0} \), \( P_u = P_{u1} \), \( P_u = P_{u2} \), or \( P_u = P_{u3} \), where, for \( i \in \{0, \ldots, 3\} \), \( P_{ui} \) is identical to policy \( P1 \) 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 \( O_{\text{Fast}} \) the ratio between the latencies of WebTTC and Flash.

The results of our experiments are 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 for large \( u \) and \( n \), with the overhead versus Flash being 1–10 for view_papers, or even higher for view_index. The high overhead is caused by the large number of checks and more complex UT propagation induced when combining many inputs into a single 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 \( P_{\text{Conf}} \) and \( P_{\text{HIPAA}} \) 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 [28, 29] gave rise to a rich IFC literature. Many IFC languages have been developed, including JIF [61], FlowCam [72], Jeeves [78], JSFlow [54], WebTTC [64], and others [5, 18, 20, 21, 25, 45, 47, 59, 60, 65, 71, 75]. Our work builds on two ideas from this field: dynamic information-flow [4, 37, 67, 77].

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Conf: view_paper</td>
<td>Conf: view_paper</td>
<td>Conf: view_paper</td>
</tr>
<tr>
<td>Conf: view_papers</td>
<td>Conf: view_papers</td>
<td>Conf: view_papers</td>
</tr>
<tr>
<td>Conf: submit_paper</td>
<td>Conf: submit_paper</td>
<td>Conf: submit_paper</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( n )</th>
<th>( P_u )</th>
<th>( O_{\text{Fast}} )</th>
<th>( O_{\text{Fast}} )</th>
<th>( O_{\text{Fast}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>4 4 4</td>
<td>6 6 6</td>
<td>9 9 9</td>
<td>18 18 18</td>
</tr>
<tr>
<td>1024</td>
<td>4 4 4</td>
<td>6 6 6</td>
<td>9 9 9</td>
<td>18 18 18</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Conf: view_paper</td>
<td>Conf: view_paper</td>
<td>Conf: view_paper</td>
</tr>
<tr>
<td>Conf: view_papers</td>
<td>Conf: view_papers</td>
<td>Conf: view_papers</td>
</tr>
<tr>
<td>Conf: submit_paper</td>
<td>Conf: submit_paper</td>
<td>Conf: submit_paper</td>
</tr>
<tr>
<td>Conf: HIPAA: view_index</td>
<td>Conf: HIPAA: view_index</td>
<td>Conf: HIPAA: view_index</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( n )</th>
<th>( P_u )</th>
<th>( O_{\text{Fast}} )</th>
<th>( O_{\text{Fast}} )</th>
<th>( O_{\text{Fast}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>4 4 4</td>
<td>6 6 6</td>
<td>9 9 9</td>
<td>18 18 18</td>
</tr>
<tr>
<td>1024</td>
<td>4 4 4</td>
<td>6 6 6</td>
<td>9 9 9</td>
<td>18 18 18</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Conf: view_paper</td>
<td>Conf: view_paper</td>
<td>Conf: view_paper</td>
</tr>
<tr>
<td>Conf: view_papers</td>
<td>Conf: view_papers</td>
<td>Conf: view_papers</td>
</tr>
<tr>
<td>Conf: submit_paper</td>
<td>Conf: submit_paper</td>
<td>Conf: submit_paper</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( n )</th>
<th>( P_u )</th>
<th>( O_{\text{Fast}} )</th>
<th>( O_{\text{Fast}} )</th>
<th>( O_{\text{Fast}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>4 4 4</td>
<td>6 6 6</td>
<td>9 9 9</td>
<td>18 18 18</td>
</tr>
<tr>
<td>1024</td>
<td>4 4 4</td>
<td>6 6 6</td>
<td>9 9 9</td>
<td>18 18 18</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Conf: view_paper</td>
<td>Conf: view_paper</td>
<td>Conf: view_paper</td>
</tr>
<tr>
<td>Conf: view_papers</td>
<td>Conf: view_papers</td>
<td>Conf: view_papers</td>
</tr>
<tr>
<td>Conf: submit_paper</td>
<td>Conf: submit_paper</td>
<td>Conf: submit_paper</td>
</tr>
<tr>
<td>Conf: HIPAA: view_index</td>
<td>Conf: HIPAA: view_index</td>
<td>Conf: HIPAA: view_index</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( n )</th>
<th>( P_u )</th>
<th>( O_{\text{Fast}} )</th>
<th>( O_{\text{Fast}} )</th>
<th>( O_{\text{Fast}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>4 4 4</td>
<td>6 6 6</td>
<td>9 9 9</td>
<td>18 18 18</td>
</tr>
<tr>
<td>1024</td>
<td>4 4 4</td>
<td>6 6 6</td>
<td>9 9 9</td>
<td>18 18 18</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Conf: view_paper</td>
<td>Conf: view_paper</td>
<td>Conf: view_paper</td>
</tr>
<tr>
<td>Conf: view_papers</td>
<td>Conf: view_papers</td>
<td>Conf: view_papers</td>
</tr>
<tr>
<td>Conf: submit_paper</td>
<td>Conf: submit_paper</td>
<td>Conf: submit_paper</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( n )</th>
<th>( P_u )</th>
<th>( O_{\text{Fast}} )</th>
<th>( O_{\text{Fast}} )</th>
<th>( O_{\text{Fast}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>4 4 4</td>
<td>6 6 6</td>
<td>9 9 9</td>
<td>18 18 18</td>
</tr>
<tr>
<td>1024</td>
<td>4 4 4</td>
<td>6 6 6</td>
<td>9 9 9</td>
<td>18 18 18</td>
</tr>
</tbody>
</table>
Table 3: Comparison of related work

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>User-defined</td>
<td>✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗</td>
<td>✗ ✗ ✗ ✗ ✗</td>
</tr>
<tr>
<td>Fine-grained</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Time-dependent</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Proofs</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Noninterference</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
<tr>
<td>Code available</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
</tr>
</tbody>
</table>

*Fine-grained restrictions defined at the data-model level only.*

and monitored execution [48, 49] where a monitor integrated into the program’s semantics ensures 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 [21], FlowCam [72], JSFlow [5], Lifty [65], and Storm [59] use type systems, while UrFlow [18] 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 [45], LWeb [64], and Estrela [11] rely on policies defined at the data-model level. Hails [45] and LWeb [64] are based on Haskell monads, Estrela [11] on modifying database queries, and Jeeves [78] and Jacqueline [77] on a lambda 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 [21], and JSFlow [5] support static reasoning about interference between inputs and outputs in a fine-grained way, thus fulfilling [R2]. Additionally, in SIF [21] and JSFlow [5], 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 [25] uses a type system to guaran-

te 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 noninterference 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 [9, 12, 13, 17, 27, 30, 54, 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 [22] and SecLTL [31] can be used to monitor noninterference. However, hyperproperty monitors usually take the set of all system traces as an input [2, 16, 24, 38–40, 50], a set which cannot be computed in reasonable time for general programs.

For classical (non-hyper-)properties, efficient monitors [7, 8, 52, 70] and enforcers [56] exist and have been used to verify data usage [6] and GDPR [3] restrictions in real-world settings.

Differential privacy [35] is a promising approach to enforcing privacy regulations [26], 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 [33, 34, 76].

The combination of fully homomorphic encryption [43] with trusted computing allows for outsourcing complex computations with reduced information leakage [41, 63]. However, these approaches are still prohibitively slow for general computation, while efficient alternatives [41] 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 [56]. 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 features of modern software systems, like communicating components, provides another avenue for future research. With more features and a broader GDPR coverage, we also plan to conduct studies involving external developers and users, exploring how TTC can help develop of new generation of user-centric, private-by-design software systems.


A FORMAL DEFINITIONS AND PROOFS

Our Isabelle/HOL formalization [57] 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 | 4.6–4.6.2 | 1145 | TTCWhile
TTC.thy | 4.4–4.5, 4.6.7–4.8 | 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 · concatenate on tuples and lists, i.e., α :: [α₁, ..., αₙ] = [α₁, ..., αₙ] and [α₁, ..., αₙ] · [β₁, ..., βₘ] = [α₁, ..., αₙ, β₁, ..., βₘ]. We fix a set of states S, an initial state S₀, a transition relation →, and a Boolean predicate wf_in_seq₂ on input sequences.

Definition A.1 (Well-formed input sequences, wf_in_seq²). An input sequence I is well-formed, writtenwf_in_seq² I, if wf_in_seq² I holds and the timestamps in I are strictly increasing.

Definition A.2 (IO trace, ttrans). We define ⇒ inductively as:
with the output we observe. In other words, we can reach state $I$ if we can find $S$, assume that in the transition following state $(\tau,S)$ we tell us that the value of $1$ cannot have been interfered with output $t_S$ at time $\tau$, except that $1 \in S$. Thus, observing $1\in S$ tells us that the value of $i$ cannot have been $d'$. 

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

$$a \in \text{dom } A \quad I = H \cdot (u, f, A, \tau) \quad \Rightarrow \quad S_0, I \rightarrow S_1, I_1$$

$$S_1 \in \text{out}(f, u, p, d, \tau') \rightarrow S'$$

We say that input $(\tau, a)$ interferes with output $\tau_1$, written $(\tau, a) \rightsquigarrow_1 \tau_1$, if there exists $I', d', \sigma_1', \tau_2$ such that

$$I' = H \cdot (u, f, A[a := d'], \tau) \quad \Rightarrow \quad \tau_2 \geq \tau_1$$

When the transition relation is $\rightarrow_{\pi,p}$, we denote by $\rightsquigarrow_{\pi,p,l}$ the corresponding interference relation.

**Definition A.4 (Information-flow traces, enforcement).** We say that an application enforces a property $P$, written $P$, iff, for any input sequence $I$ such that $\text{wf}_\text{in_seq'} I$ and $S_0, I \rightarrow_\text{IO_NOP}$, we have $\Delta_I(\sigma) \in P$, where

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

$$\rho(\text{out}(f, u, p, d), \tau) = \{\text{Out}(f, u, p, \tau)\}$$

$$\{\text{Itf}((\tau', a), \tau) \mid (\tau', a) \rightsquigarrow_1 \tau\}$$

$$\Delta_I = \text{map } \delta_I.$$
Definition A.7. Let $t$ be a UT. The relations $\equiv \subseteq \mathcal{H}^2$ and $\approx \subseteq \mathcal{T}^2$ are defined as follows:

\[
\begin{align*}
\emptyset & \equiv [t] \\
h & : h \equiv h' : t' \\
\alpha & : \equiv \alpha' \\
\langle \alpha, \alpha \rangle & \approx \langle \alpha, \alpha' \rangle \\
\langle \alpha, \alpha' \rangle & \equiv \langle \alpha, \alpha' \rangle
\end{align*}
\]

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

\[
m \equiv m' \equiv \forall v \in \mathrm{Var}. m(v) \equiv m'(v).
\]

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 $t$, the relations $\equiv_t$ and $\approx_t$ are equivalence relations.

Lemma A.9. If $\langle \alpha, \alpha \rangle \equiv \langle \alpha', \alpha' \rangle$ but $\langle \alpha, \alpha \rangle \neq \langle \alpha', \alpha' \rangle$, then $\ell \in \mathrm{uts}(\alpha) \cap \mathrm{uts}(\alpha')$.

Lemma A.10. If $\langle \alpha, \alpha \rangle \equiv \langle \alpha', \alpha' \rangle$, then $\alpha \equiv \alpha'$.

Lemma A.11. If $\langle \alpha, \alpha \rangle \equiv \langle \alpha', \alpha' \rangle$ and $\beta \equiv \beta'$, then $\langle \alpha, \beta : \alpha \rangle \equiv \langle \alpha', \beta : \alpha' \rangle$.

Lemma A.12. If $\alpha \equiv_1 \alpha'$ and $\ell \in \mathrm{uts}(\alpha) \cap \mathrm{uts}(\alpha')$, then for all $\alpha, \alpha', \beta, \beta'$, we have $\langle \alpha, \beta : \alpha \rangle \equiv_0 \langle \alpha', \beta : \alpha' \rangle$.

Lemma A.13. If $\alpha_1 \equiv \alpha_1'$ and $\alpha_2 \equiv \alpha_2'$, then $\alpha_1 \uplus \alpha_2 \equiv \alpha_1' \uplus \alpha_2'$.

Lemma A.14. $\mathrm{uts}(\alpha \uplus \beta) = \mathrm{uts}(\alpha) \cup \mathrm{uts}(\beta)$.

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

Lemma A.16. If $m \equiv m'$ and $\alpha \equiv \alpha'$, then sanitize$(m, b, \alpha) \equiv m'$ sanitize$(m', b, \alpha')$.

Lemma A.17. If $\alpha \equiv \alpha'$, then $\kappa(u, p, \alpha) \equiv \kappa(u, p, \alpha')$.

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

\[
k_1 = u = \mathrm{check} \ x \ u_1 \ p_1; \text{if } x \{ z = a \}; v = \mathrm{check} \ z \ u_1 \ p_1; w = u = v.
\]

This program first checks $x$, assigns $a$ to $z$ if $x$ is non-zero, and checks $z$. The two verdicts are written into $u$ and $v$, respectively. 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 are then used by $w$. Hence, when $x$ 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 $\kappa$ must ensure that, after executing $k_1$, $w$ is tagged with the initial UT of $x$.

Next, we prove the additional property of $x$ from Section 4.6.2:

Theorem A.18 ($k_\text{accept}$). If $\kappa_c(u, p, \alpha) = \langle \alpha', \alpha' \rangle$ (where $\kappa_c$ is $\kappa$ defined as in Section 4.6.2 using the single-UT enforcement oracle $c$) and $\ell \in \mathrm{uts}(\alpha')$, then $c(u, p, \ell) = 1$.

Example A.19. With the code $k_1$ introduced above, our definition gives $\kappa(u_1, p_1, \{ \{ z \} \} = 0, \{ \} \} if c(u_1, p_1, \ell)) = 0, or (1, \{ \{ z \} \}) if c(u_1, p_1, \ell)) = 1$. In the former case, $\ell$ 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, $\ell$ 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 $(v_1 = v_2, \{ \{ \ell \} \})$, this information flow is correctly tagged by the UTs. In both cases, $\kappa(u_1, p_1, \{ \{ \ell \} \})$ is only tagged with $\ell_1$ if $c(u_1, p_1, \ell)) = 1$.

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 $\pi$ 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 $\pi, pc, m \rightarrow^{*} \epsilon, pc', m'$.

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

Definition A.22 (Well-formedness, $\text{wf}$ code). We say that a code $\pi$ is well-formed, written $\text{wf$ code} \ \pi$, if it contains no $\text{pop}$ statements inside its $\text{while}$ blocks.

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

Lemma A.23 (eval1_ind). If $m \equiv m'$, $\| e \| (m) = w$ and $\| e \| (m') = w'$, then $w \equiv w'$.

We then show the following properties for TTCWhile programs:

Lemma A.24 (Determinism, step_deterministic). If $\pi, pc, m \rightarrow^{*} e, pc_1, m_1 \wedge \pi, pc, m \rightarrow^{*} e, pc_2, m_2$,
then $pc_1 = pc_2 \wedge m_1 = m_2$.

Lemma A.25 (Noninterference, $\text{ni}$). If $\pi, pc, m_1 \rightarrow^{*} e, pc_1, m_1 \wedge \pi, pc, m_2 \rightarrow^{*} e, pc_2, m_2$,
then $m_1 \equiv m_2$.

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 PL Locale. We fix a program transition relation $\rightarrow^*$ as above, a program $\pi$, an enforcer $\mathcal{E}$, a property $P$, an initial program counter history $pc_0$, and a null program $\epsilon$.

Further, we define $S_{0,TTC} = \{ \text{step} = \text{Taint} \}, \{ \}, (x \mapsto \{ [ ] \})$, 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).**

$$\text{wf\_in\_seq2} \ I \equiv \forall i \in \{ 1 .. |\pi(f)| \}, u, f, A, \tau, u', f', A', \tau'. \ I_i = (u, f, A, \tau) \land I_{i+1} = (u', f', A', \tau') \Rightarrow r' > r + \sum_{i=1}^{\pi(f)} \pi(f)_i \cdot \text{time}. $$

Finally, we assume:

1. $P$ is independent of past outputs and Itf-monotonic,
2. $\mathcal{E}$ is an enforcer for $P$,
3. For all $f, i \in \{ 0 .. |\pi(f)| - 1 \}$, $\pi(f)_i \cdot \text{time} > 0$,
4. For all $f, i \in \{ 0 .. |\pi(f)| - 1 \}$, $\pi(f)_i \cdot \text{time} > 0$,
5. **Noninterference:** for all $f, i \in \{ 0 .. |\pi(f)| - 1 \}$, $m_1, m_2, pc_1, pc_2, m'_1$, and $m'_2$,
6. **Determinism:** for all $f, i \in \{ 0 .. |\pi(f)| - 1 \}$, $m, pc_1, pc_2, m_1$, and $m_2$.

Assumption (1)–(5) are identical.

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 $f, i \in \{ 0 .. |\pi(f)| - 1 \}$, $\pi(f)_i \cdot \text{time} > 0$, and that every $\pi(f)_i$ has a strictly positive running time.

If $P$ is enforceable, Itf-monotonic, and independent of past outputs, then $\mathcal{T}_\pi$ enforces the privacy policy $P \subseteq T_{\Sigma_p}$.

**Proof.** Immediate corollary of TTCWhile.PL.

B POLICIES THAT DEPEND ON PAST OUTPUTS

Consider the code in Figure 8. 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. Next, 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 (I. 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 I. 9–10 to succeed. The adversary was thus able to break $P_{\text{Alice}}$ by choosing malicious code and $P_{\text{Eve}}$. To prevent the previous leak, we must either adopt a coarser propagation of UTs when covering the implicit flow I. 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 propagated (‘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 [56], which studies MFOTL from the perspective of enforcement.

**Definition C.1.** MFOTL formulae over $\Sigma = (D, A, I)$ are defined by the grammar

$$\varphi ::= a(t_1, ..., t_{i(r)}) | \neg \varphi | \varphi \lor \varphi | \exists x. \varphi | \mathbf{\bullet} \varphi | \mathbf{\circ} \varphi | \varphi \cdot \varphi | \varphi \cup \varphi,$$

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

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

1. By definition of $\Sigma_{TTC}$ and Itf-monotonicity of policies in $\Pi$.
2. By Lemma A.27.
def foo():
    if secret == 1:
        y = adv_input
    return y

def bar():
    y = 1
    if check(adv_input, me, ...):
        y = 0
    return y

The MFOTL formula \( \Psi \phi \) is defined over valuations.

Definition C.2. A valuation \( \nu : \text{Var} \cup \Delta \to \Delta \) such that \( \nu(d) = d \) for all \( d \in \Delta \).

We write \( \nu(x \mapsto d) \) for the valuation obtained from \( \nu \) by setting \( \nu(x) = d \). We can now define the semantics of MFOTL.

Definition C.3. Fix a signature \( \Sigma \). Given \( k \in \mathbb{N} \), a trace \( \tau = ((t_i, D_i))_{1 \leq i \leq k} \) over \( \Sigma \), a timeout \( t \), an MFOTL formula \( \phi \) over \( \Sigma \), we define the satisfaction relation \( \models \) as follows:

\[
\begin{align*}
\nu, i \models_{\sigma} a(t_1, ..., t_n) & \iff (r, (\nu(t_1), ..., \nu(t_n))) \in D_i \\
\nu, i \models_{\sigma} \neg \phi & \iff \nu, i \not\models_{\sigma} \phi \\
\nu, i \models_{\sigma} \exists x. \phi & \iff \nu(x \mapsto d), i \models_{\sigma} \phi \quad \text{for} \ d \in \Delta \\
\nu, i \models_{\sigma} \phi \lor \psi & \iff \nu, i \models_{\sigma} \phi \quad \text{or} \ \nu, i \models_{\sigma} \psi \\
\nu, i \models_{\sigma} \psi \land \psi & \iff i > 1 
\text{and} \ \nu, i-1 \models_{\sigma} \phi \text{ and } t_i - t_{i-1} \in I \\
\nu, i \models_{\sigma} \phi \lor \psi & \iff i + 1 \leq |\sigma|, \nu, i+1 \models_{\sigma} \phi, \text{ and } t_{i+1} - t_i \in I \\
\nu, i \models_{\sigma} \exists f. \psi & \iff \nu, j \models_{\sigma} \psi \text{ for some } j \leq i, t_i - t_j \in I, \\
\text{and } \nu, k \models_{\sigma} \phi \text{ for all } k, j < k \leq i \\
\nu, i \models_{\sigma} \phi & \iff o, j \models_{\sigma} \psi \text{ for some } |\sigma| \geq j \geq i, t_j - t_i \in I, \\
\text{and } \nu, k \models_{\sigma} \phi \text{ for all } k, j > k \geq i
\end{align*}
\]

Each MFOTL formula \( \phi \) over \( \Sigma \) defines a policy \( P_{\phi} \subseteq \mathbb{T}_{\Sigma} \) as follows:

Definition C.4. The MFOTL formula \( \phi \) over \( \Sigma \) defines the policy

\[
P_{\phi} = \{ \sigma \in \mathbb{T}_{\Sigma} \mid \exists \nu, v, 1 \models_{\sigma} \phi \}.
\]

Sufficient enforceability conditions are established in [56].
• The patient ID from model Treatment is visible to the patient and prescribing entity;
• The patient ID from model Diagnosis is visible to the patient and recognizing entity;
• The shared information from model BusinessAssociate-Agreement is visible to the covered entity and business associate;
• The standard, first party, second party, and purpose from model Transaction is visible to the first party and second party;
• The email from model UserProfile is visible to the user itself.

We perform tests with Individual only, hence we consider:

$$\psi_{\text{HIPAA}} = \forall f', u, p, o, l, f, a. \text{Out}(f', u, p, o) \land \text{Itf}(l, o) \land \Diamond \text{ln}(f, a, l)$$

$$\Rightarrow f = "\text{Individual}" \land u = \langle u \rangle.$$