Privacy with Formal Guarantees: The Databank

François Hublet and David Basin and Srđan Krstić

ABSTRACT

Data ownership is the users' right to control how their personal data is used. Current approaches for handling user data do not properly enforce data ownership. One major reason for this is that service providers can *both* provision their services' code and execute it on user data that they collected and stored on their own servers.

We propose the Databank model, a new architecture that separates data storage and code execution from code provisioning. Its main component, called a *Databank*, stores and processes user data. Users can define data-usage policies describing how their data may be processed, expressed as temporal constraints on service behavior. Service providers independently deploy code to the Databank written in a programming language that uses dynamic information-flow control (to track data during service execution) and runtime verification (to detect and prevent policy violations) to enforce user policies. Users thus obtain formal guarantees about the use of data they own.

We specify and implement the Databank. We evaluate our implementation on a proof-of-concept application, which attests to its generality, usability, and modest performance overhead.

CCS CONCEPTS

• **Security and privacy → Information flow control**; **Trust frameworks**; **Formal security models**; **Logic and verification**; • **Computer systems organization → Other architectures**; • **Information systems → Web applications**.

KEYWORDS

data protection, information-flow control, runtime verification

1 INTRODUCTION

The modern data economy suffers from a major power imbalance. Many service providers engage in *data barter* [28], offering "free" services and profiting from the collected user data. Such services may store, process, and even sell this data without the users' knowledge or consent. This infringes on users' *data ownership* rights [10], namely, their ability to control how their (personal) data is used.

As a field of privacy, data protection [48] is concerned with enforcing users' data ownership rights. These rights are enforced when each datum is treated according to its owner's preferences, which are typically expressed as a *data-usage policy*. Recent data protection regulations such as the EU's General Data Protection Regulation (GDPR) mandate the enforcement of a particular class of data-usage policies. For instance, user data may be collected and processed solely for purposes that users have consented to. However, such regulations are difficult to audit and enforce: once user data has been collected, there are no *technical* controls restricting how the service provider may use it. Thus, in practice, the regulations only mandate some notion of service provider accountability, where enforcement degenerates to ex-post audit and fines. We believe that this situation is perpetuated by inadequate technical solutions.

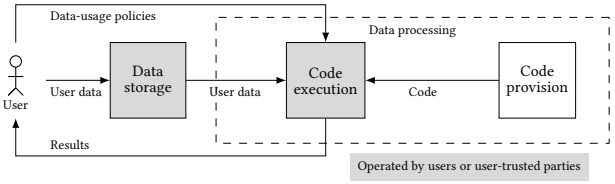

Figure 1: An abstract architecture enforcing data ownership

In the general case, data ownership rights can be enforced at the architectural level (Figure 1) by splitting an application into multiple components and letting users, or user-trusted third parties, run the critical components (e.g., data storage). The trust in a third party may stem from the party's implementation of technical measures (e.g., rigorous audits or use of trusted computing), economic incentives (e.g., serving privacy-conscious users), or even future legal mandates requiring such parties to be accountable only to the users.

The state-of-the-art can be roughly classified based on the trust placed in service providers (Section 2). In the untrusted service-provider setting, variants of the above architecture have been proposed [21, 35, 38, 43, 51, 56], most of which rely on basic access control rather than information-flow control (IFC) mechanisms. Differential privacy approaches [23, 24, 58] can provide good statistical guarantees, but are only applicable to specific data types or use cases. In contrast, in the trusted service-provider setting, the emphasis is on technical solutions that provide formal guarantees [13, 15, 17, 19, 32, 39, 45, 53, 59]. These solutions, however, do not protect users against the threats posed by purposefully non-compliant services. Existing systems also differ in the expressiveness of the supported data-usage policy language.

We propose the *Databank model*, an architecture that separates data storage and code execution from code provisioning. It supports (a fragment of) Metric First-Order Temporal Logic (MFOTL) [14] as its data-usage policy language. Users can specify complex temporal policies in MFOTL constraining the usage of their data by the service without any knowledge of its business logic. Such *application-agnostic* data-usage policies are achieved by encoding critical operations on user data into events that MFOTL policies may refer to.

A component called the *Databank* implements data storage and code execution. Independently of the users' data-usage policies, service providers submit code that the Databank executes on user data. Application code can be written in a fragment of Python, called DPython, which is *policy-agnostic*, i.e., it does not have any policy-oriented syntactic constructs. Still, the program behavior is restricted to enforce users' policies, with invalid executions throwing security exceptions and terminating [52]. This is achieved by DPython's semantics, which is designed to execute critical operations only when they are policy-compliant. DPython semantics combine methods from information-flow control and runtime verification (Section 3), tracking information flows at runtime and preventing outputs that violate user policies. We show that with

Anonymous Submission to ACM CCS 2022, November 14–19, 2022, Los Angeles, CA 2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

appropriate design decisions service providers can develop practical and usable services in DPython.

In contrast to the existing solutions in the untrusted service-provider setting, if operated by a user-trusted third party, the Databank *formally* guarantees that users' data-usage policies are enforced. Unlike systems addressing the trusted service-provider setting, the Databank allows *users* to specify policies and it does not require coordination between users and service providers: users need not understand services' business logics to specify policies, and service providers need not know user's policies to provision their code.

In summary, we specify, develop, and evaluate a new architecture that enforces data ownership. In particular:

(1) We propose the Databank model (Section 4), a web architecture that enforces expressive data-usage policies in the presence of untrusted code. We describe its design and usage.
(2) We specify the Databank component of our architecture by defining the syntax and semantics of DPython. We show that DPython provides formal guarantees for the enforcement of users' data-usage policies. (Section 5).
(3) We develop a Databank prototype (Section 6), implement a realistic event management service as a proof of concept, and report on its performance and our development experience. Overall, our evaluation demonstrates the Databank's practicality and its modest overhead (Section 7).

2 RELATED WORK

We group related approaches into four categories.

Future web architectures. Personal data stores (PDS) are services that provide data storage to users and an API for service providers to access user data under user-defined restrictions. This idea by Krohn et al. [38] has many implementations including the Hub-of-All-Things (HAT) [43], Meeco [40], openPDS [21], and Mydex [35].

The openPDS project [21] introduced the concept of *SafeAnswers*, which are code snippets provided by a third party that are executed on user data within a PDS. Only the execution results are revealed to the third-party. As SafeAnswers lacks any formal analysis, it is generally unclear how much information is disclosed.

Schwarzkopf et al. [54] envision that user data should be stored in PDS and queried by service providers through dynamically updated views. Unlike in the Databank model, Schwarzkopf et al. do not aim to protect users against malicious service providers, and question IFC mechanisms' ability to achieve practical performance, which we show to be possible in our case (Section 7).

The Social Linked Data (Solid) project [51] gives users control over what data leaves their PDS, but does not facilitate trusted code execution. Moreover, Solid cannot prevent third parties from replicating data they have legitimate access to. Therefore, users effectively lose control over shared data.

Riverbed [56] is a Python interpreter with IFC that enforces user-defined data usage policies. Instead of a user-trusted third party, the interpreter is run by the service provider with a Riverbed proxy remotely attesting its execution. The proxy only releases the data if attestation succeeds. Unlike the Databank, Riverbed's IFC handles only explicit information flows. Its policy language is simple and non-temporal: a user can only (dis)allow that her data is aggregated with data of other users or written to permanent storage, and users may specify lists of trusted endpoints and server stacks. Riverbed, as well as the closely related Mitigator [41], does not provide rigorous soundness proofs for the languages they support.

Runtime verification of data-usage policies. Schneider's security automata [52] provide the reference framework for the enforcement of security policies. To enforce policies in programs, inline reference monitors (IRM) [25, 26] have been suggested: given a policy, code is rewritten by introducing checks that trigger specific handler code in case of violations. An alternative and related approach is monitoring-oriented programming (MOP) [11, 12, 47]. In both IRM and MOP, it is programmers, not users, who specify policies.

Basin et al. have used MFOTL to monitor data usage in distributed systems [6]. Following a similar methodology, Arfelt et al. [1] have specified part of the GPDR in MFOTL and checked compliance of system execution logs. This demonstrates the relevance of temporal specifications for the enforcement of data ownership rights. These techniques focus on auditing existing systems, rather than on developing applications that enforce data ownership.

IFC for web applications. The development of programming languages that incorporate IFC has been an area of active research for over twenty years. Examples include Jif [16, 42], FlowCaml [55], and JSFlow [36]. General methodologies have been suggested [37].

IFC systems for database-backed web applications have also been proposed. SELinks [19] extends the multi-tier programming language Links [18] with IFC. LWeb [45] combines a Haskell IFC library with a web programming framework to enforce confidentiality and integrity policies in databases. Storm [39] enforces centralized data-dependent policies using refinement types.

Unlike SELinks and LWeb, our language allows for policy-agnostic programming [3], where programs and policies are specified separately. Jacqueline [59] is an extension of the Django Python web framework that also allows for policy-agnostic programming.

Hails [32] implements the so-called MPVC architecture, which delegates model-policy (MP) and view-controller (VC) components to mutually distrusting parties, using IFC to enforce privacy. Like the Databank, MPVC separates code provisioning and policy enforcement and supports policy-agnostic programming.

All the above systems trust service providers to define data-usage policies on the users' behalf. Even if users could write their own policies, they would need to know the data model and business logic of each individual application.

Other approaches. Another promising approach to enforcing privacy regulations [20] is differential privacy [24], which can provide strong *statistical* privacy guarantees. Being statistical, these guarantees may be practically insufficient or limitedly usable depending on the type of data, the size of datasets, and the queries considered [22, 23, 58]. Program synthesis techniques are emerging [57]. Moreover, differential privacy cannot be used to ensure privacy in applications that handle individual data items.

Another line of work [30, 44] combines fully homomorphic encryption (FHE) [31] with trusted computing techniques to outsource complex computations with reduced information leakage. Such techniques, while not requiring a trusted third-party, must balance between efficiency and security: complete FHE is still prohibitively

slow, while more efficient alternatives [30] only protect implicit flows against honest-but-curious service providers.

3 PRELIMINARIES

We now introduce the two techniques leveraged by our model.

3.1 Information-flow control

Information-flow control (IFC) detects and controls information propagation during program execution. Information flows can be *explicit*, when caused by value assignments, or *implicit*, when caused by control-flow instructions. Besides implicit and explicit flows, other *covert channels* exist [49], e.g., whenever one observes program termination, information flows via the *termination* channel.

Absence of information flow (between two memory locations) is typically expressed as noninterference [33]. If noninterference holds under the assumption that program runs always terminate, it is called *termination-insensitive* noninterference (TINI) [50].

In this paper, we focus on dynamic IFC to enforce TINI. In classical dynamic IFC [2, 29], each value (or rather the memory location containing it) is typically assigned a sensitivity label. The set of such labels L forms a semilattice $(L, \sqcup, \sqsubseteq)$, where $\sqcup$ is the join operation on labels and $\sqsubseteq$ is a pre-order on labels.

We label each value with the *set of all user-inputted values that have affected it*. To represent this set more compactly, we use unique *input labels* instead of the input values themselves. An input label $\ell_x \in \mathcal{L}$ of a value $x \in \mathbb{D}$ inputted by a user $u \in \mathbb{U}$ is obtained by applying a collision-resistant hash function to x combined with the corresponding argument name and time of input, and pairing the resulting hash with u. Thus our dynamic IFC uses the semilattice $(\mathcal{P}(\mathcal{L}), \cup, \subseteq)$, such that all values affected by x are labeled with a set containing ℓ_x. We also use the function $\omega : \mathcal{L} \to \mathbb{U}$ that maps each input label back to the user that provided the input value.

3.2 Runtime verification

Runtime verification (RV) [5, 27] aims to verify a system against its formal specifications (*policies*) during its execution.

We first recap the syntax and semantics of Metric First-Order Temporal Logic (MFOTL) [14], an RV policy specification language. We focus on a since-only MFOTL (sMFOTL) fragment that uses only the temporal operator "since". We then introduce monitors for MFOTL policies and show how they can be used for enforcement.

Notation. Let $\mathrm{Seq}\,(X)$ denote the set of all finite sequences of elements of set X and ϵ an empty sequence. For $a, a' \in \mathrm{Seq}\,(X)$, we write $a \preceq a'$ iff a is a prefix of a', $|a|$ for a's length, and a_i for the a's ith element (1-based), if it exists. Set $\mathbb{B} = \{\mathtt{true}, \mathtt{false}\}$ contains Boolean constants, and $\mathbb{I}$ all (finite or infinite) intervals over $\mathbb{N}$.

sMFOTL. A *signature* $(\mathbb{D}, \mathbb{E}, \iota)$ consists of an infinite domain $\mathbb{D}$, a set of event names $\mathbb{E}$, and an arity function $\iota : \mathbb{E} \to \mathbb{N}$. For a set $\mathbb{V}$ of variables ($\mathbb{V} \cap \mathbb{D} = \emptyset$), the following grammar defines sMFOTL, where t_i, r, x, and I range over $\mathbb{V} \cup \mathbb{D}$, $\mathbb{E}$, $\mathbb{V}$, and $\mathbb{I}$, respectively:

$$\varphi ::= t_1 \approx t_2 \mid r(t_1, ..., t_{\iota(r)}) \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\varphi \mid \varphi\ \mathsf{S}_I\ \varphi.$$

A formula of the form $r(t_1, ..., t_{\iota(r)})$ is called a *predicate*. We derive other operators: truth $\top := \exists x.\ x \approx x$, inequality $t_1 \not\approx t_2 := \neg(t_1 \approx t_2)$, conjunction $\varphi \wedge \psi := \neg(\neg\varphi \vee \neg\psi)$, implication $\varphi \Rightarrow \psi := \neg\varphi \vee \psi$,

$$\begin{array}{ll}
v, i \models_\rho t_1 \approx t_2 & \text{iff } v(t_1) = v(t_2) \\
v, i \models_\rho r(t_1, ..., t_n) & \text{iff } r(v(t_1), ..., v(t_n)) \in D_i \\
v, i \models_\rho \neg\varphi & \text{iff } v, i \not\models_\rho \varphi \\
v, i \models_\rho \varphi \vee \psi & \text{iff } v, i \models_\rho \varphi \text{ or } v, i \models_\rho \psi \\
v, i \models_\rho \exists x.\ \varphi & \text{iff } v[x \mapsto d], i \models_\rho \varphi \text{ for some } d \in \mathbb{D} \\
v, i \models_\rho \varphi\ \mathsf{S}_I\ \psi & \text{iff } v, j \models_\rho \psi \text{ for some } j \le i, \tau_i - \tau_j \in I, \\
 & \text{and } v, k \models_\rho \varphi \text{ for all } k \text{ with } j < k \le i
\end{array}$$

Figure 2: Semantics of sMFOTL

once $\blacklozenge_I\,\varphi := \top\ \mathsf{S}_I\ \varphi$, and historically $\blacksquare_I\,\varphi := \neg\blacklozenge_I\neg\varphi$. We use $\mathbb{V}_\varphi$ for the set of φ's free variables, and call φ closed iff $\mathbb{V}_\varphi = \emptyset$.

A valuation v is a mapping $\mathbb{V}_\varphi \to \mathbb{D}$, assigning domain elements to the free variables of φ. Overloading notation, v can be extended to the domain $\mathbb{V}_\varphi \cup \mathbb{D}$, setting $v(t) = t$ whenever $t \in \mathbb{D}$. Finally, we write $v[x \mapsto d]$ for the function equal to v, except that $v(x)$ is d.

An element of $\mathbb{E} \times \mathrm{Seq}\,(\mathbb{D})$, denoted as $r(d_1, ..., d_{\iota(r)})$, is an *event*, where r is its name and $d_1, ..., d_{\iota(r)}$ are its parameters.

sMFOTL formulas are interpreted over *traces*, which are finite sequences $(\tau_i, D_i)_{i\in\{1,...,k\}}, k \in \mathbb{N}$, where $\tau_i \in \mathbb{N}$ is a timestamp, $D_i \in \mathbb{DB} = \mathcal{P}(\mathbb{E} \times \mathrm{Seq}\,(\mathbb{D}))$ is a finite set of events, and $\tau_i \le \tau_{i+1}$ for all i. The empty trace is denoted by ϵ, and the set of all traces is denoted by $\mathbb{T} \subset \mathrm{Seq}\,(\mathbb{N} \times \mathbb{DB})$. The relation $v,\ i \models_\rho \varphi$, shown in Figure 2, defines the satisfaction of the formula φ with respect to a valuation v, an index $i \le |\rho|$, and a trace ρ.

Monitoring. A *monitor* for a closed formula φ takes a trace as input and checks if φ holds at every index in the trace, i.e., it implements the function $\mathcal{M}_\varphi(\rho) = \forall v, i.\ v, i \models_\rho \varphi$. Note that $\mathcal{M}_\varphi(\epsilon)$ is satisfied for all φ. In practice, a monitor incrementally computes this Boolean verdict while reading the trace.

In this paper, we use monitors for closed sMFOTL formulas φ. Specifically, we use MonPoly [9], a state-of-the-art monitor that implements $\mathcal{M}_\varphi$ for an expressive safety fragment of MFOTL [14].

When monitoring is used to *enforce* a specification, events can be partitioned into two subsets: the sets $\mathbb{CE} \subseteq \mathbb{E}$ and $\mathbb{E} \setminus \mathbb{CE}$ contain the names of *controllable* and *uncontrollable* events, respectively. The former kind can be suppressed by some mechanism running the monitoring tool and overseeing the execution of the system, while the latter can only be observed. A policy φ is enforceable [7] iff for any valuation v, index i, trace $\rho = (\tau_i, D_i)_{i\in\{1,...,k\}}$, set of events $D' \in \mathcal{P}((\mathbb{E} \setminus \mathbb{CE}) \times \mathrm{Seq}\,(\mathbb{D}))$, and timestamp $\tau' \ge \tau_k$, the trace $\rho' = \rho +\!+ [(\tau', D')]$ obtained by appending a set of zero or more uncontrollable events after ρ's last event is such that $v, i \models_\rho \varphi \Rightarrow v, i \models_{\rho'} \varphi$. In other words, enforceable policies can be violated only by controllable events, hence the mechanism is always able to ensure violation-free execution by suppressing events.

4 THE DATABANK MODEL

The Databank model (Figure 3) is an architecture consisting of users $\mathbb{U}$, service providers $\mathbb{P} \subseteq \mathbb{U}$, and the Databank component implementing data storage and code execution components. The Databank has five subcomponents:

(1) applications $A_1, \ldots, A_n$, deployed by service providers;
(2) a proxy P, by which users can access $A_1, \ldots, A_n$;
(3) a database D containing (user) data;
(4) a policy interface I by which users input data-usage policies;
(5) a monitor M.

Each user $u \in \mathbb{U}$ defines their data-usage policy φ_u through I. These policies are sMFOTL formulas defined in terms of events (see below) that are logged by the Databank when performing critical operations. Applications A_i are deployed to the Databank by the service providers. The users can query applications A_i through the proxy P. By default, all user data collected by applications is stored within the database D and can only be used by the applications submitted to the Databank. Whenever an application wants to perform an operation (e.g. an output) that may cause some user's policy to be violated, P logs this operation, and M checks if the execution log complies with $\Phi := \bigwedge_{i \in \mathbb{U}} \varphi_i$, i.e., M implements $\mathcal{M}_\Phi$. If so, the operation is executed; otherwise, execution is aborted.

Applications can access the database directly, but receive and serve user requests only through P. To communicate with users, each application defines a number of *entry points* distinct across applications that are exposed through P. The proxy P checks with M that user outputs do not lead to policy violations. Additionally, applications can call external, service-provider side functions. Since information passed to such functions leaves the Databank, any such external call must also go through P and be approved by M. Finally, besides answering external function calls by Databank applications, service providers can interact with the Databank as users.

In principle, events logged by the Databank—and used by users to specify their policies—may correspond to any relevant operation executed by some A_i, e.g., sending data to other users or service providers, writing to global memory, aggregating data from different users, etc. Here, we focus on privacy and thus consider two events $\mathbb{E} = \{\texttt{Input}, \texttt{Learn}\}$, with $\iota(\texttt{Input}) = 5$ and $\iota(\texttt{Learn}) = 3$:

- $\texttt{Input}(a, e, \omega(\ell_x), \ell_x, x)$ occurs whenever an input x is passed by user $\omega(\ell_x)$ as argument a of entry point e;
- $\texttt{Learn}(v, \omega(\ell_x), \ell_x)$ occurs whenever information about input x by user $\omega(\ell_x)$ flows to some user v.

Note that in both events, the parameter ℓ_x is the input label identifying x, not the input value. In the corresponding predicates of the sMFOTL policies, input labels are represented by either universally or existentially quantified variables. We also define predicates $\texttt{InputV}(a, e, u, x) := \exists \ell. \texttt{Input}(a, e, u, \ell, x)$ and $\texttt{InputL}(a, e, u, \ell) := \exists x. \texttt{Input}(a, e, u, \ell, x)$ as syntactic sugar used in policies where the input labels or values are not important, respectively.

Since we want to enforce data ownership, users may not restrict how their peers' data is used. Therefore, we syntactically restrict policy φ_u to contain only predicates Input and Learn where $\omega(\ell_x) = u$.

Example 4.1. Consider a scenario where the user Alice wants to retrieve a list of her own posts on a social media application A. The posts are inputs to argument text of an entry point post. Suppose that one of Alice's posts p_1 with input label ℓ_1 is three weeks old. Alice specifies the following policy φ_{Alice}: anyone can see her posts that are no older than two weeks, while only she can see the older posts. Alice's policy can be specified in sMFOTL as

$$\forall v, \ell.\ \texttt{Learn}(v, \texttt{Alice}, \ell) \wedge \blacklozenge_{(2w,\infty)} \texttt{InputL}(\texttt{text}, \texttt{post}, \texttt{Alice}, \ell) \Rightarrow v \approx \texttt{Alice}.$$

This policy states that whenever some user v learns one of Alice's inputs, labeled with ℓ, which was input (by Alice) more than two weeks ago to argument text of entry point post, then v is Alice.

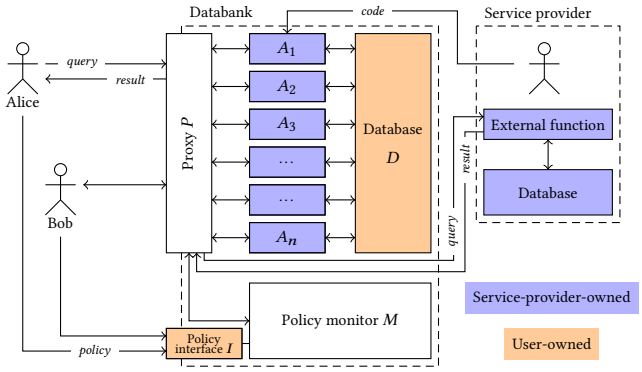

Figure 3: The Databank model

Now suppose that application A shows non-personal ads together with the list of posts. Thus, before serving the page, A calls an external function ads implemented by its own service provider S. This function does not have arguments, so it does not provide S any information about Alice.

The corresponding Databank execution is shown in Figure 4a. Alice queries the entry point list_posts to get the list of her posts (Step 1). After logging the corresponding Input event (Steps 2-3), P executes list_posts of A (Step 4). The function retrieves the requested content from the database (Steps 5-6). Now A wants to call external function ads to retrieve non-personal ads. The proxy P does not log any new event to the monitor, since no information is passed to the external function. Thus the call is performed and the ads a are retrieved (Steps 7-9). The application builds the page π from Alice's posts $\{p_1, \dots p_n\}$ and the ads a. This page is labeled with input labels $\{\ell_1, \dots, \ell_m\}$ representing all user inputs used to compute π. The application computes this page and sends it back to the proxy P to be shown to Alice (Step 10). Finally, P logs that Alice would learn information about inputs corresponding to input labels $\ell_1, \dots, \ell_m$ (Step 11), and the monitor checks that the resulting trace does not violate any user's policy (Step 12). Since only Alice's own data is returned to her, which is allowed by her own policy, no violation is detected, and the page is sent to Alice (Step 13).

Example 4.2. Now suppose that Bob tries to access all of Alice's posts, as shown in Figure 4b. Recall that φ_{Alice} specifies that other users cannot see her old posts. After the initial request (Step 1), the execution is similar up to Step 10. The same page π is computed as before, but now events of the form $\texttt{Learn}(\texttt{Bob}, \texttt{Alice}, \ell_j)$ are logged to the trace (Step 11). In particular, the event $\texttt{Learn}(\texttt{Bob}, \texttt{Alice}, \ell_1)$ is logged. As ℓ_1 is the input label of Alice's post p_1 posted three weeks ago, the formula $\blacklozenge_{(2w,\infty)} \texttt{InputL}(\texttt{text}, \texttt{post}, \texttt{Alice}, \ell_1)$ must hold. Therefore, logging $\texttt{Learn}(\texttt{Bob}, \texttt{Alice}, \ell_1)$ causes Alice's policy to be violated (Step 12). The trace is restored to its previous state (Step 13) by removing all the Learn events logged in Step 10, and the proxy does not deliver π, but rather an "error" page, to Bob (Step 14).

We depict the proxy P as a separate component of the Databank for conceptual clarity. However, in practice, its functionality is incorporated directly into the semantics of the programming language used by the service providers to write applications A_i.

1. Alice → P : get list_posts(*Alice*)
2. $P \to M$: log Input(user, list_posts, Alice, ℓ_0, *Alice*)
3. $M \to P$: true
4. $P \to A$: list_posts(*Alice*)
5. $A \to D$: SELECT * FROM posts WHERE author = *Alice*
6. $D \to A$: $\{p_1, \ldots, p_n\}$
7. $A \to P$: call S.ads()
8. $P \to S$: call ads()
9. $S \to A$: a
10. $A \to P$: $\langle \pi, \{\ell_1, \ldots, \ell_m\} \rangle$ to Alice
11. $P \to M$: log Learn(Alice, Alice, ℓ_1), ..., Learn(Alice, Alice, ℓ_m)
12. $M \to P$: true
13. $P \to$ Alice : π

(a) Databank execution from Example 4.1

1. Bob → P : get list_posts(*Alice*)
2. $P \to M$: log Input(user, list_posts, Bob, ℓ_0, *Alice*)

...

10. $A \to P$: $\langle \pi, \{\ell_1, \ldots, \ell_m\} \rangle$ to Bob
11. $P \to M$: log Learn(Bob, Alice, ℓ_1), ..., Learn(Bob, Alice, ℓ_m)
12. $M \to P$: false
13. $P \to M$: revert_log
14. $P \to$ Bob : "*error*"

(b) Databank execution from Example 4.2

Figure 4: Databank usage examples

Adversary model. The Databank model is designed to be secure against a powerful adversary who may corrupt any number of users, service providers, or both, and may know the code of applications A_i and user policies φ_u. As a user, the adversary can query entry points with arbitrary arguments. As a service provider, he can deploy arbitrary application code to the Databank or alter the code of existing applications. However, he cannot compromise the Databank itself: he cannot alter the policy monitor M, the proxy P, or the policy specification interface I, nor directly read from or write to the database D. He also cannot observe termination, timing, or aborting functions (which is treated as non-termination). All communication channels represented in Figure 3 are secure, i.e., both authentic and confidential. In practice, communication using a protocol such as TLS can be assumed. Hence, communication between honest parties cannot be eavesdropped or modified.

5 PROGRAMMING THE DATABANK

We now describe DPython, a new Python fragment whose special semantics enforces data-usage policies in the Databank model.

We proceed in three steps. First, we present the syntax and semantics of a simpler language called Dmol (*Databank model language*; Section 5.1). Dmol covers all core features of DPython while abstracting away implementation details (e.g. SQL queries and session variables). This allows for conceptually clear definitions and proofs. Second, we show that the Databank running Dmol code enforces Φ (Section 5.2). Finally, we describe DPython and explain how Dmol's semantics and properties can be extended to DPython in a straightforward way (Section 5.3).

In this section, Φ denotes the (global) policy to be enforced, which is the conjunction of all users' data usage policies.

```
def hello(uid) {
  msg = "Hi";
  name = users[uid]["name"];    # `users` is global
  if name != "" {
    msg = msg + ", " + name };
  return msg
}
```

Figure 5: An example Dmol code

5.1 Dmol

Dmol's syntax is given in Figure 6, where *ident* denotes identifiers, *string* denotes string literals, and *const* denotes constant literals from $\mathbb{D}$. The nonterminals Ω_1 and Ω_2 denote standard unary and binary operators. The notation $[\![A]\!]$ means that A is optional, while $(\!|A|\!)^*$ means that A repeats 0 or more times.

A Dmol program is a sequence of zero or more functions (Eq. 1), which can all serve as entry points. Dmol is based on a While language (Eq. 3, 4, 7-10, 14–16) enriched with support for function calls (Eq. 2, 5) and key-value mappings, called *dictionaries* in Python (Eq. 11, 17–20). The functions `len` and `keys` return the length and the keys of a dictionary (Eq. 19, 20). Variables can be global or local, with names taken from two disjoint sets of global ($\mathcal{G}$) and local ($\mathcal{H}$) identifiers respectively.

Two web-specific features are part of Dmol. The `external_call` statement provides calls to external (remote) functions over the network (Eq. 6). It takes a URL identifying the function and a list of arguments. The function `me` gets the crrent user's ID (Eq. 21).

Example 5.1. An example Dmol function is given in Figure 5. This code defines a function that takes a user id as an argument, retrieves the corresponding user's name from a global dictionary `users`, and shows "Hi, *name*", or simply "Hi" if the name is empty.

In the rest of this section, we fix a Dmol program P, encoded by code and argnames functions, which map P's function names to their corresponding code and list of arguments, respectively.

Next, we describe Dmol's semantics in three steps. We first introduce the abstract state machine that defines Dmol's semantics. We then define *memory cells*, which constitute the basic building block of Dmol's memory model. Finally, we use memory cells to describe concrete states and transition relation of the state machine.

State machine. We instantiate the set $\mathbb{D}$ with atomic values of type `int`, `float`, `bool`, and `str`. A *query* $(z, u, e, ((a_1, i_1), \ldots, (a_k, i_k)))$

$$
\begin{aligned}
prog &::= fun; \ldots; fun && (1)\\
fun &::= \texttt{def}\ ident(ident, \ldots, ident)\ \{block\} && (2)\\
block &::= stmt; \ldots; stmt && (3)\\
assign &::= lhs\_expr = expr && (4)\\
&\mid ident = ident(url, expr) && (5)\\
&\mid ident = \texttt{external\_call}(url, expr) && (6)\\
stmt &::= assign && (7)\\
&\mid \texttt{return}\ expr && (8)\\
&\mid \texttt{if}\ expr\ \{block\}[\![\texttt{else}\ \{block\}]\!] && (9)\\
&\mid \texttt{while}\ expr\ \{block\} && (10)\\
&\mid \texttt{del}\ lhs\_expr[expr] && (11)\\
lhs\_expr &::= ident(\!|[expr]|\!)^* && (12)\\
url &::= string && (13)\\
expr &::= const \mid ident && (14)\\
&\mid \Omega_1\ expr && (15)\\
&\mid expr\ \Omega_2\ expr && (16)\\
&\mid \{expr:expr, \ldots, expr:expr\} && (17)\\
&\mid expr[expr] && (18)\\
&\mid \texttt{len}(expr) && (19)\\
&\mid \texttt{keys}(expr) && (20)\\
&\mid \texttt{me}() && (21)
\end{aligned}
$$

Figure 6: Syntax of Dmol programs

is a quadruple of a timestamp z, a user u, the name e of the function e to be used as an entry point, and a sequence of pairs of argument names a_j and argument values (or *inputs*) i_j. The set of queries is denoted by $\mathbb{Q} := \texttt{int} \times \mathbb{U} \times \texttt{str} \times \text{Seq}\,(\texttt{str} \times \mathbb{D})$. An *output* is a pair of a user and a value; thus the set of outputs is $\mathbb{O} := \mathbb{U} \times \mathbb{D}$. The (sub)sequence of values sent to some $u \in \mathbb{U}$ within $o \in \mathbb{O}$ can be extracted as $\text{filter}(o, u) := [v \mid (u, v) \in o]$.

We consider a state machine $(\mathbb{S}, s_0, \rightsquigarrow, \mathsf{in})$ with a set of states $\mathbb{S}$, an initial state $s_0 \in \mathbb{S}$, a deterministic transition relation $\rightsquigarrow$: $\mathbb{S} \times \text{Seq}\,(\mathbb{O}) \to \mathbb{S} \times \text{Seq}\,(\mathbb{O})$, and a function $\mathsf{in} : \mathbb{S} \times \mathbb{Q} \to \mathbb{S}$ that encodes the effect of reading a user query. We write $s, o \rightsquigarrow s', o'$ to denote that given a state s and an output sequence o, one execution step produces the new state s' and output sequence o', such that $o \preceq o'$. We say that a state $s \in \mathbb{S}$ is *final* if, for all $o \in \text{Seq}\,(\mathbb{O})$, $s, o \rightsquigarrow s, o$.

We assume that all executions terminate. Termination can be ensured e.g. by adding timeouts or requiring termination proofs, which we reserve for further work. Assuming termination, for all $s \in \mathbb{S}$ and $q \in \mathbb{Q}$, there is exactly one sequence of pairs $s_1, \varepsilon \rightsquigarrow s_2, o_2 \rightsquigarrow \cdots \rightsquigarrow s_n, o_n$ where $s_1 = \mathsf{in}(s, q)$ and s_n is a final state. We write $\mathsf{run1}(s, q)$ for the sequence $(\mathsf{in}(s, q), \varepsilon), \ldots, (s_n, o_n)$, i.e., a run of the Databank starting in the state s given query q.

The function $\mathsf{run} : \text{Seq}\,(\mathbb{Q}) \to \text{Seq}\,(\text{Seq}\,(\mathbb{S} \times \text{Seq}\,(\mathbb{O})))$ takes a sequence of queries. Then, starting from the initial state s_0, it iteratively applies run1 consuming one query at each call and returns the sequence of run1 outputs for each query. For all q and k such that $q \in \text{Seq}\,(\mathbb{Q})$ and $1 \le k < |\mathsf{run}(q)|$, the final state of the kth sequence in $\mathsf{run}(q)$ is used in the $(k+1)$th application of run1.

Memory cells. A *memory cell* is a pair $\langle v, \overline{\ell} \rangle$ representing a labeled value. For readability we denote memory cells using $\langle \ldots \rangle$ to distinguish them from other pairs. The value v can be either an element of $\mathbb{D}$ or a dictionary $\{k_1 : c_1, \ldots, k_n : c_n\}$ mapping atomic values $k_i \in \mathbb{D}$ to memory cells c_i, while $\overline{\ell}$ is a finite set of input labels. The set of memory cells is denoted by C. Note that any data structure encoded using a dictionary can also be encoded using a memory cell. We call such data structures *labeled*. E.g., a list of memory cells $[c_1, \ldots, c_n]$ can be embedded in C as the *labeled list* $\langle [c_1, \ldots, c_n], \overline{\ell} \rangle := \langle \{1 : c_1, \ldots, n : c_n\}, \overline{\ell} \rangle$. Similarly, we can encode and label, most data structures used in web applications.

Given input labels w and w', two memory cells $c = \langle v, \overline{\ell} \rangle$ and $c' = \langle v', \overline{\ell}' \rangle$ are (w, w')-*indistinguishable*, written $c =_{w,w'} c'$, iff one of the following holds: (i) $w \in \overline{\ell}$ and $w' \in \overline{\ell'}$; or (ii) $v = v'$ and $\overline{\ell} = \overline{\ell}'$; or (iii) v and v' are dictionaries with the same keys, (w, w')-indistinguishable corresponding values, and $\overline{\ell} = \overline{\ell}'$. Intuitively, c and c' are (w, w')-indistinguishable if they are equal except on subcells labeled with w and w', respectively.

Example 5.2. Two (w, w')-indistinguishable memory cells $\langle \{4 : \langle 5, \{w\} \rangle, 6 : \langle 7, \emptyset \rangle\}, \emptyset \rangle$ and $\langle \{4 : \langle \{5 : \langle 9, \emptyset \rangle\}, \{w\} \rangle, 6 : \langle 7, \emptyset \rangle\}, \emptyset \rangle$, are shown in Figure 7 side by side. Each cell has values in squares on the left, and a set of labels on the right. Dashed arrows denote the $=_{w,w'}$ relation. The lowest pair of indistinguishable subcells is indistinguishable due to rule (i), since the LHS is labeled by w and the RHS by w'. The middle pair is indistinguishable—due to (ii)—because the two cells are equal. Finally, the two top cells are indistinguishable by due to rule (iii): they have the same labels, the same dictionary keys, and pairwise indistinguishable values.

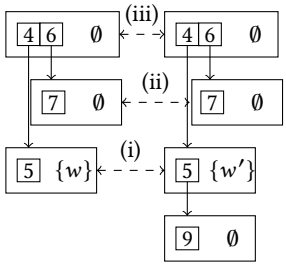

Figure 7: Two (w, w')-indistinguishable memory cells, exemplifying rules (i)–(iii)

Dmol's memory model is based on memory cells and their underlying dictionary structure. Variables, but also traces, are stored within memory cells. A *labeled trace* is an element $\hat{\rho} \in C$ where:

(1) $\hat{\rho}$ is a dictionary whose keys are int timestamps;
(2) For every such timestamp z, $\hat{\rho}[z]$ is a dictionary with the keys "Input", and "Learn";
(3) For every z, $\hat{\rho}[z]$["Input"] is a dictionary with arbitrary keys and values of the form $[a_j, e_j, \ell_{x_j}, \ell_{x_j}, x_j]$ (labels omitted), representing $\texttt{Input}(a_j, e_j, \omega(\ell_{x_j}), \ell_{x_j}, x_j)$;
(4) For every z, $\hat{\rho}[z]$["Learn"] is a dictionary with keys representing users, and for such user v, $\hat{\rho}[z]$["Learn"][v] is a dictionary whose values are of the form $[\omega(\ell_{x_j}), \ell_{x_j}]$, representing $\texttt{Learn}(v, \omega(\ell_{x_j}), \ell_{x_j})$.

State space. We now refine the notion of state introduced above to reflect the state of a Dmol program more precisely.

Every state $s \in \mathbb{S}$ is a septuple $\langle \sigma, u, j, m, z, \hat{\rho}, \lambda \rangle$ where:

- σ is either:
  - A sequence of statements to be executed next;
  - Ret c, denoting that the current entry point returned $c \in C$;
  - Stop, denoting that the current entry point aborted.
- u is the user querying the current entry point.
- j is the *label stack*, a list of sets of input labels; the union of these sets, denoted by all j, is the set of input labels of the instruction pointer.
- $m = \text{mem}(s)$ is the memory state, which is a mapping from $\mathcal{G} \cup \mathcal{H}$ to C. Uninitialized variables have value $\langle \bot, \emptyset \rangle$.
- z is the current timestamp, updated at each entry point call.
- $\hat{\rho}$ is a labeled trace.
- λ is either $\bot$ or a *caller state* from which execution will resume when the current function returns.

The initial state is $s_0 = \langle \text{Stop}, \bot, [], \langle \{\}, \emptyset \rangle, 0, \hat{\epsilon}, \bot \rangle$, where $\hat{\epsilon}$ denotes an empty labeled trace.

A *core memory location* is an expression of the form $i[v_1] \ldots [v_k]$ where $i \in \mathcal{G} \cup \mathcal{H}$ and for all $p \in \{1, \ldots, k\}$, $v_p \in \mathbb{D}$. The *longest core memory location* (LCML) of an assignment $i[v_1] \ldots [v_n]$=e' is the longest $i[v_1] \ldots [v_k]$, $k \le n$, such that all $(v_p)_{1 \le p \le k}$ are constant.

The *domain* of a memory state m, denoted by dom m, is the set of all core memory locations defined in m, i.e., the set of all $i[v_1] \ldots [v_k]$ such that $i \in \mathcal{G} \cup \mathcal{H}$, v_1 is a key of $m[i]$, v_2 is a key of $m[i][v_1]$, …, and v_k is a key of $m[i][v_1] \ldots [v_k]$.

Given a memory state m and $V \subseteq \mathcal{G} \cup \mathcal{H}$, we denote by $m|_V$ the memory state obtained from m by setting $m[v]$ to $\langle \bot, \emptyset \rangle$ for all $v \notin V$.

When two memory states m and m' have disjoint initialized variables, we write $m \cup m' := v \mapsto$ if $m[v] \neq \langle \bot, \emptyset \rangle$ then $m'[v]$ else $m[v]$.

Indistinguishability on the above state space is defined inductively as follows: for all $w, w' \in \mathcal{L}$, for all $s, s' \in \mathbb{S}$ such that $s = \langle \sigma, u, j, m, z, \hat{\rho}, \lambda \rangle$ and $s' = \langle \sigma', u', j', m', z', \hat{\rho}', \lambda' \rangle$, then $s \sim_{w,w'} s'$ holds iff either (1) $w \in$ all j and $w' \in$ all j', or (2) all of these hold: (i) $\sigma = \sigma'$, $u = u'$, $z = z'$ (ii) $|j| = |j'|$ and for all $1 \leq i \leq |j|$, $\langle \bot, j_i \rangle =_{w,w'} \langle \bot, j'_i \rangle$ (iii) $m =_{w,w'} m'$ (iv) $\hat{\rho} =_{w,w'} \hat{\rho}'$ (v) $\lambda \sim_{w,w'} \lambda'$.

Transition relation. The in function loads the code of the called function, resets the label stack, the local memory, the caller state and the timestamp, sets all arguments, and logs the corresponding `Input` events to the labeled trace. Its pseudocode is as follows:

```
1: function in(⟨σ0, u0, j0, m, z0, ρ̂, λ0⟩, ⟨z, u, e, ((a1, i1), ..., (an, in))⟩)
2:     if (a1, ..., an) = argnames(e) then                ▷ if argument names are valid
3:         m' ← m |_G [a1 ↦ ⟨i1, ℓi1⟩, ..., an ↦ ⟨in, ℓin⟩]   ▷ set argument values
4:         c ← ⟨[⟨ap, ∅⟩, ⟨e, ∅⟩, ⟨u, ∅⟩, ⟨ℓip, ∅⟩, ⟨ip, {ℓip}⟩], ∅⟩ | 1 ≤ p ≤ n], ∅⟩
5:         ρ̂, n ← ρ̂.copy(), |ρ̂| + 1
6:         ρ̂[n] ← ⟨{"ts" : ⟨z, ∅⟩, "Input" : c, "Learn" : ⟨{}, ∅⟩}, ∅⟩   ▷ log Input events
7:         return ⟨code(e), u, [[]], m', z, ρ̂, ⊥⟩          ▷ return updated state
8:     else
9:         return ⟨σ0, u0, j0, m, z0, ρ̂, λ0⟩
```

We suppose given two sets of unary and binary operators Ω_1 and Ω_2 which can be used both as symbols and as functions. We consider two functions implemented via straightforward static analysis:

- assigned(s), where s is a sequence of statements, returns the set of LCMLs in the assignments' LHS reachable from s.
- called(s, u), where s is a sequence of statements and $u \in \mathbb{U}$, returns the set of all owners of external functions reachable from s, to which u is added iff s contains a return statement.

For every external function f, we denote by $\Theta(f)$ the *owner* of f, i.e., the service provider hosting function f.

The semantics of Dmol expressions is presented in Figure 9, while the semantics of instructions is in Figure 10. Auxiliary functions used in this semantics are described in Figure 11. This defines the relation $\rightsquigarrow$, whose transitive closure is denoted by $\rightsquigarrow^*$.

We now review the main features of Dmol's semantics.

Inputs and outputs. Whenever an entry point is called, the function in reads user inputs, generates their fresh input labels, constructs the corresponding memory cells, and adds them to the system state as local variables. It also updates the timestamp in the system state with the current time.

Two types of Dmol instructions produce outputs: `return` instructions of entry points (Eq. 30–31) and external function calls (Eq. 40–40). When a `return` e instruction in an entry point called by a user u is executed in state s with output sequence o, the expression e is first evaluated to a memory cell $\langle v, \overline{\ell} \rangle$, and we have a transition $s, o \rightsquigarrow s', (o +\!\!+ [(u, v)])$ where s' is a final state with the local variables deallocated. External function calls also modify the output sequence, which otherwise remains unmodified.

Label propagation. Dmol's semantics leverages dynamic IFC [2, 29] to propagate input labels during execution. Labels are attached to values stored in memory, the instruction pointer, and the trace.

Example 5.3. Figure 8 describes the execution of the Dmol code from Figure 5 on two examples. It focuses on σ, j, and the values of the local variables `uid`, `name`, and `msg`.

	σ	j	`uid`	`name`	`msg`	Rule
s_1	*msg* = "Hi"; *name* =...	[∅]	⟨1, {ℓ_1}⟩	undef.	undef.	-
s_2	*name* = *users*[*uid*]...	[∅]	⟨1, {ℓ_1}⟩	undef.	⟨"Hi", ∅⟩	43
s_3	`if` *name* != "" {...	[∅]	⟨1, {ℓ_1}⟩	⟨"B", {ℓ_1, ℓ_2}⟩	⟨"Hi", ∅⟩	43
s_4	*msg* = *msg* + ", " +...	[{ℓ_1, ℓ_2}]	⟨1, {ℓ_1}⟩	⟨"B", {ℓ_1, ℓ_2}⟩	⟨"Hi", ∅⟩	33
s_5	`Pop;return` *msg*	[{ℓ_1, ℓ_2}]	⟨1, {ℓ_1}⟩	⟨"B", {ℓ_1, ℓ_2}⟩	⟨"Hi, B", {ℓ_1, ℓ_2}⟩	43
s_6	`return` *msg*	[∅]	⟨1, {ℓ_1}⟩	⟨"B", {ℓ_1, ℓ_2}⟩	⟨"Hi, B", {ℓ_1, ℓ_2}⟩	47
s_7	`Ret` ⟨"Hi, B", {ℓ_1, ℓ_2}⟩	[∅]	⟨1, {ℓ_1}⟩	⟨"B", {ℓ_1, ℓ_2}⟩	⟨"Hi, B", {ℓ_1, ℓ_2}⟩	30

(a) First execution of `hello`: `uid` = 1, `name` is not empty

	σ	j	`uid`	`name`	`msg`	Rule
s'_1	*msg* = "Hi"; *name* =...	[∅]	⟨3, {ℓ_3}⟩	undef.	undef.	-
s'_2	*name* = *users*[*uid*]...	[∅]	⟨3, {ℓ_3}⟩	undef.	⟨"Hi", ∅⟩	43
s'_3	`if` *name* != "" {...	[∅]	⟨3, {ℓ_3}⟩	⟨"", {ℓ_3, ℓ_4}⟩	⟨"Hi", ∅⟩	43
s'_4	`Pop;return` *msg*	[{ℓ_1, ℓ_2}]	⟨3, {ℓ_3}⟩	⟨"", {ℓ_3, ℓ_4}⟩	⟨"Hi", {ℓ_3, ℓ_4}⟩	34
s'_5	`return` *msg*	[∅]	⟨3, {ℓ_3}⟩	⟨"", {ℓ_3, ℓ_4}⟩	⟨"Hi, B", {ℓ_3, ℓ_4}⟩	47
s'_6	`Ret` ⟨"Hi, B", {ℓ_1, ℓ_2}⟩	[∅]	⟨3, {ℓ_3}⟩	⟨"", {ℓ_3, ℓ_4}⟩	⟨"Hi, B", {ℓ_3, ℓ_4}⟩	30

(b) Second execution of `hello`: `uid` = 3, `name` is empty

Figure 8: Label propagation

In the first execution, `uid` initially contains the value 1 labeled with the input label ℓ_1, and `name` and `msg` are uninitialized. After `msg` is initialized (line 2), the name of user with uid 1 is read from a global dictionary. This results in the name "B" labeled with ℓ_2 being loaded into `name` (line 3). The memory cell for `name` is labeled with both ℓ_1 (since it labels `uid`, which was used in the query) and ℓ_2 (which labels value "B" in the database). When performing the `if` test (line 4), labels ℓ_1 and ℓ_2 labeling the variable `name` are added to the input labels of the instruction pointer. Since `name` is not the empty string, the then branch is executed (line 5), propagating the instruction pointer's labels to `msg` to capture implicit flows. The execution finally leaves the `if` block, removing the labels ℓ_1 and ℓ_2 from the instruction pointer, and the return instruction is performed (line 6).

In the second execution, `uid` is initially 3 and the retrieved `name` is empty (line 5). Thus, the then branch is not executed. Nevertheless, `name`'s input labels (here ℓ_3, ℓ_4) are propagated to `msg`, as `msg` $\in$ assigned(b_1), where b_1 is the block in the then branch.

Through label propagation, Dmol's semantics enforce TINI [50]:

Lemma 5.4 (TINI). *Consider $q \in \mathsf{Seq}(\mathbb{Q})$ and its copy q' where a single input i in q with input label w has been replaced by some other input i' with input label w'. Now choose $1 \leq j \leq |q|$ and consider the sequences of pairs of states and outputs $r = \mathsf{run}(q)_j$ and $r' = \mathsf{run}(q')_j$ produced by the jth query for q and q'. Then there exists a nondecreasing mapping $\mu : \{1, \ldots, |r|\} \to \{1, \ldots, |r'|\}$ such that for all $k \in \mathrm{dom}\ \mu$, $\mathsf{fst}(r_k) \sim_{w,w'} \mathsf{fst}(r'_{\mu(k)})$.*

Proof. See Appendix A.1. □

Intuitively, this means that for every increasing sequence of states in one run, we can find an increasing sequence of states in the other run that are pairwise indistinguishable from the first ones.

Example 5.5. For the two sequences of states s and s' of length $|s| = 7$ and $|s'| = 6$ from Figure 8, we can choose $\mu = (1\ 2\ 3\ 4\ 4\ 5\ 6)$.

Logging. For a query $(z, u, e, ((a_1, x_1), \ldots, (a_k, x_k)))$, the function in logs all events `Input`$(a_j, e, u, \ell_{x_j}, x_j)$, for $1 \leq j \leq k$, at the current timestamp to the labeled trace in the system state. The return instructions of entry points and the external function calls log `Learn` events. Specifically, for every input label ℓ labeling the return value of an entry point called by u (respectively, one of the

$$\frac{c \in \mathbb{D}}{[[c,u,m,z]] = \langle c, \emptyset\rangle} \qquad \frac{i \in \operatorname{dom} m}{[[i,u,m,z]] = m[i]} \qquad \frac{i \in \mathcal{G} \cup \mathcal{H} \setminus \operatorname{dom} m}{[[i,u,m,z]] = \langle \bot, \emptyset\rangle} \tag{22}$$

$$\frac{\omega \in \Omega_1 \qquad [[e_1,u,m,z]] = \langle v_1, \overline{\ell}_1\rangle}{[[\omega\ e_1,u,m,z]] = \langle \omega(v_1), \overline{\ell}_1\rangle} \qquad \frac{\omega \in \Omega_2 \qquad [[e_1,u,m,z]] = \langle v_1, \overline{\ell}_1\rangle \qquad [[e_2,u,m,z]] = \langle v_2, \overline{\ell}_2\rangle}{[[e_1\ \omega\ e_2,u,m,z]] = \langle \omega(v_1,v_2), \overline{\ell}_1 \cup \overline{\ell}_2\rangle} \tag{23}$$

$$\frac{\forall i \in \{1,\dots,n\},\ [[e_i,u,m,z]] = c_i \qquad \forall i \in \{1,\dots,n\},\ [[k_i,u,m,z]] = \langle \gamma_i, \overline{\ell}_i\rangle \quad \forall i \in \{1,\dots,n\},\ \gamma_i \in \mathbb{D}}{[[\{k_1 : e_1, \dots, k_n : e_n\}, u, m, z]] = \langle \{\gamma_1 \mapsto c_1, \dots, \gamma_n \mapsto c_n\}, \cup_{p=1}^{n} \overline{\ell}_p\rangle} \tag{24}$$

$$\frac{\forall i \in \{1,\dots,n\},\ [[e_i,u,m,z]] = c_i \qquad \forall i \in \{1,\dots,n\},\ [[k_i,u,m,z]] = \langle \gamma_i, \overline{\ell}_i\rangle \quad \exists i \in \{1,\dots,n\},\ \gamma_i \notin \mathbb{D}}{[[\{k_1 : e_1, \dots, k_n : e_n\}, u, m, z]] = \langle \bot, \cup_{p=1}^{n} \overline{\ell}_p\rangle} \tag{25}$$

$$\frac{[[e_1,u,m,z]] = \langle v_1, \overline{\ell}_1\rangle \qquad v_2 \in \operatorname{dom} \langle v_1, \overline{\ell}_1\rangle \qquad [[e_2,u,m,z]] = \langle v_2, \overline{\ell}_2\rangle \qquad v_1\,[v_2] = \langle v', \overline{\ell}'\rangle}{[[e_1[e_2],u,m,z]] = \langle v', \overline{\ell}_1 \cup \overline{\ell}_2 \cup \overline{\ell}'\rangle} \tag{26}$$

$$\frac{[[e_1,u,m,z]] = \langle v_1, \overline{\ell}_1\rangle \qquad [[e_2,u,m,z]] = \langle v_2, \overline{\ell}_2\rangle \qquad v_2 \notin \operatorname{dom} \langle v_1, \overline{\ell}_1\rangle}{[[e_1[e_2],u,m,z]] = \langle \bot, \overline{\ell}_1 \cup \overline{\ell}_2\rangle} \tag{27}$$

$$\frac{[[e_1,u,m,z]] = \langle v_1, \overline{\ell}_1\rangle \qquad \operatorname{dom} \langle v_1, \overline{\ell}_1\rangle = \{k_1, \dots, k_n\} \qquad k_1 < \dots < k_n}{[[\mathtt{keys}(e_1),u,m,z]] = \langle \{0 \mapsto \langle k_1, \emptyset\rangle, \dots, n-1 \mapsto \langle k_n, \emptyset\rangle\}, \overline{\ell}_1\rangle} \tag{28}$$

$$\frac{[[e_1,u,m,z]] = \langle v_1, \overline{\ell}_1\rangle \qquad |\langle v_1, \overline{\ell}_1\rangle| = n}{[[\mathtt{len}(e_1),u,m,z]] = \langle n, \overline{\ell}_1\rangle} \qquad \frac{}{[[\mathtt{me},u,m,z]] = \langle u, \emptyset\rangle} \tag{29}$$

Figure 9: Evaluation of Dmol expressions

$$\frac{\begin{array}{c}\text{assigned}\,(s) = A \\ \text{called}\,(s,u) = U \qquad \mathrm{fp}(\hat{\rho}, z, \overline{\ell}_1, \text{all}\ j, u) = \langle \hat{\rho}', \top, \overline{\ell}^*\rangle \\ \text{value}\,([[e_1,u,m,z]]) = \langle v_1, \overline{\ell}_1\rangle \qquad \hat{\rho}'' = \text{sanitize}(\hat{\rho}', U, \overline{\ell}^*, z)\end{array}}{\begin{array}{l}\langle \mathtt{return}\ e_1; s, u, j, m, z, \hat{\rho}, \bot\rangle, o \\ \rightsquigarrow \langle \text{Ret}\ \langle v_1, \overline{\ell}_1 \cup \overline{\ell}^*\rangle, u, j, \text{adopt}(A, \text{all}\ j, m), F, z, \hat{\rho}'', \bot\rangle, o{+}{+}[(v_1, u)]\end{array}} \tag{30}$$

$$\frac{\begin{array}{c}\text{value}\,([[e_1,u,m,z]]) = \langle v_1, \overline{\ell}_1\rangle \\ \text{assigned}\,(s) = A \qquad \mathrm{fp}(\hat{\rho}, z, \overline{\ell}_1, \text{all}\ j, u) = \langle \hat{\rho}', \bot, \overline{\ell}^*\rangle \\ \text{called}\,(s,u) = U \qquad \hat{\rho}'' = \text{sanitize}(\hat{\rho}', U, \overline{\ell}^*, z)\end{array}}{\langle \mathtt{return}\ e_1; s, u, j, m, z, \hat{\rho}, \bot\rangle, o \rightsquigarrow \langle \text{Stop}, u, j, \text{adopt}(A, \text{all}\ j, m), z, \hat{\rho}'', \bot\rangle, o} \tag{31}$$

$$\frac{\lambda \neq \bot \qquad \text{assigned}\,(s) = A \qquad [[e_1,u,m,z]] = \langle v_1, \overline{\ell}_1\rangle}{\begin{array}{l}\langle \mathtt{return}\ e_1; s, u, j, m, z, \hat{\rho}, \lambda\rangle, o \\ \rightsquigarrow \langle (\text{Ret}\ \langle v_1, \overline{\ell}_1 \cup \text{all}\ j\rangle) :: t, u, j, \text{adopt}(A, \text{all}\ j, m), z, \hat{\rho}, \lambda\rangle, o\end{array}} \tag{32}$$

$$\frac{\begin{array}{c}\text{assigned}\,(b_2) = A \qquad [[e_1,u,m,z]] = \langle \top, \overline{\ell}_1\rangle \\ \text{called}\,(b_2,u) = U \qquad \hat{\rho}' = \text{sanitize}(\hat{\rho}, U, \overline{\ell}_1 \cup \text{all}\ j, z)\end{array}}{\begin{array}{l}\langle \mathtt{if}\ e_1\ \{b_1\}\ \mathtt{else}\ \{b_2\}; s, u, j, m, z, \hat{\rho}, \lambda\rangle, o \\ \rightsquigarrow \langle b_1; \text{Pop}; s, u, \text{push}(j, \overline{\ell}_1, b_1), \text{adopt}(A, \overline{\ell}_1 \cup \text{all}\ j, m), z, \hat{\rho}', \lambda\rangle, o\end{array}} \tag{33}$$

$$\frac{\begin{array}{c}\text{assigned}\,(b_1) = A \qquad [[e_1,u,m,z]] = \langle v_1, \overline{\ell}_1\rangle \\ \text{called}\,(b_1,u) = U \qquad v_1 \neq \top \qquad \hat{\rho}' = \text{sanitize}(\hat{\rho}, U, \overline{\ell}_1 \cup \text{all}\ j, z)\end{array}}{\begin{array}{l}\langle \mathtt{if}\ e_1\ \{b_1\}\ \mathtt{else}\ \{b_2\}; s, u, j, m, z, \hat{\rho}, \lambda\rangle, o \\ \rightsquigarrow \langle b_2; \text{Pop}; s, u, \text{push}(j, \overline{\ell}_1, b_2), \text{adopt}(A, \overline{\ell}_1 \cup \text{all}\ j, m), z, \hat{\rho}', \lambda\rangle, o\end{array}} \tag{34}$$

$$\frac{[[e_1,u,m,z]] = \langle \top, \overline{\ell}_1\rangle}{\begin{array}{l}\langle \mathtt{while}\ e_1\ \{b_1\}; s, u, j, m, z, \hat{\rho}, \lambda\rangle, o \\ \rightsquigarrow \langle b_1; \text{Pop}; \mathtt{while}\ e_1\ \{b_1\}; s, u, \text{push}(j, \overline{\ell}_1, b_1), m, z, \hat{\rho}', \lambda\rangle, o\end{array}} \tag{35}$$

$$\frac{\begin{array}{c}\text{assigned}\,(b_1) = A \qquad v_1 \neq \top \\ \text{called}\,(b_1,u) = U \qquad [[e_1,u,m,z]] = \langle v_1, \overline{\ell}_1\rangle \qquad \hat{\rho}' = \text{sanitize}(\hat{\rho}, U, \overline{\ell}_1 \cup \text{all}\ j, z)\end{array}}{\begin{array}{l}\langle \mathtt{while}\ e_1 : b_1; s, u, j, m, z, \hat{\rho}, \lambda\rangle, o \\ \rightsquigarrow \langle \text{Pop}; s, u, \text{push}(j, \overline{\ell}_1, b_1), \text{adopt}(A, \overline{\ell}_1 \cup \text{all}\ j, m), z, \hat{\rho}', \lambda\rangle, o\end{array}} \tag{36}$$

$$\frac{\begin{array}{c}\forall p \in \{1,\dots,n\},\ [[e_p,u,m,z]] = \langle v_p, \overline{\ell}_p\rangle \\ \lambda' = \langle i{=}f(e_1,\dots,e_n); s, u, j, m, z, \hat{\rho}, \lambda\rangle\end{array}}{\lambda', o \rightsquigarrow \langle \text{code}(f), u, j, \text{set}(m|_{\mathcal{G}}, \text{argnames}(f), (v_1,\dots,v_n)), z, \hat{\rho}, \lambda'\rangle, o} \tag{37}$$

$$\frac{}{\begin{array}{l}\langle \text{Ret}\ \langle v', \overline{\ell}'\rangle, u, j', m', z, \hat{\rho}', \langle i{=}f(e_1,\dots,e_n); s, u, j, m, z, \hat{\rho}, \lambda\rangle\rangle, o' \\ \rightsquigarrow \langle s, u, j, (m'|_{\mathcal{G}} \cup m|_{\mathcal{H}})[i \mapsto \langle v', \overline{\ell}' \cup \text{all}\ j\rangle], z, \hat{\rho}', \lambda\rangle, o'\end{array}} \tag{38}$$

$$\frac{}{\begin{array}{l}\langle \text{Stop}, u, j', m', z, \hat{\rho}', \langle i{=}f(e_1,\dots,e_n); s, u, j, m, z, \hat{\rho}, \lambda\rangle\rangle, o' \\ \rightsquigarrow \langle \text{Stop}, u, j, m'|_{\mathcal{G}} \cup m|_{\mathcal{H}}, z, \hat{\rho}', \lambda\rangle, o'\end{array}} \tag{39}$$

$$\frac{\begin{array}{c}\text{called}\,(s,u) = U \qquad \mathrm{fp}(\hat{\rho}, z, \overline{\ell}_1, \text{all}\ j, u) = \langle \hat{\rho}', \top, \overline{\ell}^*\rangle \\ \text{value}\,([[e_1,u,m,z]]) = \langle v_1, \overline{\ell}_1\rangle \qquad \hat{\rho}'' = \text{sanitize}(\hat{\rho}', U, \overline{\ell}^*, z)\end{array}}{\begin{array}{l}\langle i = \text{external\_call}(r, e_1); s, u, j, m, z, \hat{\rho}, \lambda\rangle, o \\ \rightsquigarrow \langle s, u, \text{pushbot}(j, \overline{\ell}^*), m[i \mapsto \langle \text{call}\,(r, v_1), \overline{\ell}^*\rangle], z, \hat{\rho}'', \lambda\rangle, o{+}{+}[(v_1, \Theta(r))]\end{array}} \tag{40}$$

$$\frac{\begin{array}{c}\text{called}\,(s,u) = U \qquad \mathrm{fp}(\hat{\rho}, z, \overline{\ell}_1, \text{all}\ j, u) = \langle \hat{\rho}', \bot, \overline{\ell}^*\rangle \\ \text{value}\,([[e_1,u,m,z]]) = \langle v_1, \overline{\ell}_1\rangle \qquad \hat{\rho}'' = \text{sanitize}(\hat{\rho}', U, \overline{\ell}^*, z)\end{array}}{\langle i = \text{external\_call}(r, e_1); s, u, j, m, z, \hat{\rho}, \lambda\rangle, o \rightsquigarrow \langle \text{Stop}, u, j, m, z, \hat{\rho}'', \lambda\rangle, o} \tag{41}$$

$$\frac{[[e_1,u,m,z]] = \langle v_1, \overline{\ell}_1\rangle}{\langle i = e_1; s, u, j, m, z, \hat{\rho}, \lambda\rangle, o \rightsquigarrow \langle s, u, j, m[i \mapsto \langle v_1, \overline{\ell}_1 \cup \text{all}\ j\rangle], z, \hat{\rho}, \lambda\rangle, o} \tag{42}$$

$$\frac{\begin{array}{c}[[e',u,m,z]] = \langle v', \overline{\ell}'\rangle \\ \forall p \in \{1,\dots,n\},\ [[e_p,u,m,z]] = \langle v_p, \overline{\ell}_p\rangle \quad i\,[v_1]\dots[v_{n-1}] \in \operatorname{dom} m \quad v_n \in \mathbb{D}\end{array}}{\begin{array}{l}\langle i[e_1]\dots[e_n] = e'; s, u, j, m, z, \hat{\rho}, \lambda\rangle, o \\ \rightsquigarrow \langle s, u, j, \text{adoptall}(i, (v_1,\dots,v_{n-1}), \\ \qquad (\text{all}\ j \cup \overline{\ell}_1, \overline{\ell}_2, \dots, \overline{\ell}_n), m)[i[v_1]\dots[v_n] \mapsto \langle v', \overline{\ell}'\rangle], z, \hat{\rho}, \lambda\rangle, o\end{array}} \tag{43}$$

$$\frac{\begin{array}{c}q < n \\ [[e',u,m,z]] = \langle v', \overline{\ell}'\rangle \qquad i[v_1]\dots[v_{q-1}] \in \operatorname{dom} m \\ \forall p \in \{1,\dots,q\},\ [[e_p,u,m,z]] = \langle v_p, \overline{\ell}_p\rangle \quad i[v_1]\dots[v_q] \notin \operatorname{dom} m \vee v_q \notin \mathbb{D}\end{array}}{\begin{array}{l}\langle i[e_1]\dots[e_n] = e'; s, u, j, m, z, \hat{\rho}, \lambda\rangle, o \\ \rightsquigarrow \langle s, u, j, \text{adoptall}(i, (v_1,\dots,v_{q-1}), (\text{all}\ j \cup \overline{\ell}_1, \overline{\ell}_2, \dots, \overline{\ell}_q), m), z, \hat{\rho}, \lambda\rangle, o\end{array}} \tag{44}$$

$$\frac{\begin{array}{c}[[e',u,m,z]] = \langle v', \overline{\ell}'\rangle \\ \forall p \in \{1,\dots,n\},\ [[e_p,u,m,z]] = \langle v_p, \overline{\ell}_p\rangle \quad i\,[v_1]\dots[v_{n-1}] \in \operatorname{dom} m \wedge v_n \in \mathbb{D}\end{array}}{\begin{array}{l}\langle \mathtt{del}\ i[e_1]\dots[e_n]; s, u, j, m, z, \hat{\rho}, \lambda\rangle, o \\ \rightsquigarrow \langle s, u, j, \text{adoptall}(i, (v_1,\dots,v_{n-1}), \\ \qquad (\text{all}\ j \cup \overline{\ell}_1, \overline{\ell}_2, \dots, \overline{\ell}_n), m) \setminus \{i[v_1]\dots[v_n]\}, z, \hat{\rho}, \lambda\rangle, o\end{array}} \tag{45}$$

$$\frac{\begin{array}{c}q \leq n \\ [[e',u,m,z]] = \langle v', \overline{\ell}'\rangle \qquad i[v_1]\dots[v_{q-1}] \in \operatorname{dom} m \\ \forall p \in \{1,\dots,q\},\ [[e_p,u,m,z]] = \langle v_p, \overline{\ell}_p\rangle \quad i[v_1]\dots[v_q] \notin \operatorname{dom} m \vee v_q \notin \mathbb{D}\end{array}}{\begin{array}{l}\langle \mathtt{del}\ i[e_1]\dots[e_n]; s, u, j, m, z, \hat{\rho}, \lambda\rangle, o \\ \rightsquigarrow \langle s, u, j, \text{adoptall}(i, (v_1,\dots,v_{q-1}), (\text{all}\ j \cup \overline{\ell}_1, \overline{\ell}_2, \dots, \overline{\ell}_q), m), z, \hat{\rho}, \lambda\rangle, o\end{array}} \tag{46}$$

$$\frac{}{\langle \text{Pop}; s, u, h :: j', m, z, \hat{\rho}, \lambda\rangle, o \rightsquigarrow \langle s, u, j', m, z, \hat{\rho}, \lambda\rangle, o} \tag{47}$$

$$\frac{}{\langle \text{Ret}\ c, u, j, m, z, \hat{\rho}, \bot\rangle, o \rightsquigarrow \langle \text{Ret}\ c, u, j, m, z, \hat{\rho}, \bot\rangle, o} \tag{48}$$

$$\frac{}{\langle \text{Stop}, u, j, m, z, \hat{\rho}, \bot\rangle, o \rightsquigarrow \langle \text{Stop}, u, j, m, z, \hat{\rho}, \bot\rangle, o} \tag{49}$$

Figure 10: Small-step semantics of Dmol instructions

$$\text{value}(\langle v, \bar{\ell}\rangle) = \begin{cases} \langle v, \bar{\ell}\rangle & \text{if } v \in \mathbb{D} \\ \langle \bot, \bar{\ell}\rangle & \text{otherwise} \end{cases}$$

$$\text{adopt1}(\bar{\ell}', \langle d, \bar{\ell}\rangle) = \langle d, \bar{\ell} \cup \bar{\ell}'\rangle$$

$$\text{adopt}(A, \bar{\ell}, m) = m[k \mapsto \text{adopt1}(\bar{\ell}, m[k]) \mid k \in A]$$

$$\text{adoptall}(k, (v_i)_{1\le i<n}, (\bar{\ell}_i)_{1\le i\le n}, m) = \text{adopt}(\{k\}, \bar{\ell}_1, \text{adopt}(\{k[v_1]\}, \bar{\ell}_2, \ldots, \text{adopt}(\{k[v_1]\ldots[v_{n-1}]\}, \bar{\ell}_n, m))$$

$$\text{sanitize}(\hat{\rho}, U, \bar{\ell}, z) = \hat{\rho}[z[\text{“Input”}][v] \mapsto \text{adopt1}(\bar{\ell}, \hat{\rho}[z][\text{“Input”}][v]) \mid v \in U]$$

$$\text{pushbot}(j'{+}{+}t, \bar{\ell}) = j'{+}{+}[t \cup \bar{\ell}]$$

$$\text{push}(j'{+}{+}t, \bar{\ell}, b) = \begin{cases} \bar{\ell} :: (j'{+}{+}t) & \text{if } b \text{ does not contain a } \texttt{return} \\ j'{+}{+}[t \cup \bar{\ell}] & \text{otherwise} \end{cases}$$

Figure 11: Auxiliary functions used in program semantics

arguments of the external function by service provider u) an event $\texttt{Learn}(u, \omega(\ell), \ell)$ is logged. Other instructions do not log events.

Example 5.6. The output sequence and the trace produced by the execution from Figure 8a are shown in Figure 12. The output sequence is extended by the `return` instruction, which also logs two `Learn` events to the trace. The `Input` event corresponding to `Alice` querying `hello` with 1 is logged by `in` and is already present in s_1.

Enforcement. Conceptually, Dmol has access to a monitor $\tilde{\mathcal{M}}_\Phi$ implemented as a Dmol function. The monitor takes a labeled trace as input and returns an atomic `bool` value within a memory cell. A monitor implemented in Dmol exploits the label propagation mechanism to properly label verdicts. For our prototype, we use a labeled variant of MonPoly's [9] monitoring algorithm (see Section 6).

The monitor $\tilde{\mathcal{M}}_\Phi$ is called before any critical operation, specifically, before the `return` instructions of entry points and external function calls. If it detects that executing the next instruction would violate Φ, then execution is stopped by entering a special (final) state with $\sigma = \text{Stop}$, and the current entry point never returns. Our approach is therefore non-Truman [46], i.e., as soon as some part of the output triggers a violation, nothing is returned. In contrast, so-called Truman approaches (which show only non-violating parts of the output) may produce inconsistencies that leak information.

By leading the Databank to block or not block the execution, the monitor's verdict can give rise to implicit flows, which in turn must be logged to the trace. Algorithm 1 describes the auxiliary function fp used in rules 30–31 and 40–41, which performs a fixpoint computation to account for monitoring-related implicit flows.

The function fp is passed a trace ($\hat{\rho}$), a timestamp (z), the input labels of the return value or external function argument ($\bar{\ell}$), instruction pointer labels ($\bar{\ell}'$), and the current user's id (u). First, the input labels of the result are concatenated with those of the instruction pointer (to account for implicit information flows) and stored in μ (line 2). Variables μ', storing the set of input labels, and $\hat{\rho}$, storing

		Output sequence	Events logged to the labeled trace $\hat{\rho}$
System states	s_1	ε	[Input(uid, hello, Alice, ℓ_1, 1)]
	…	…	…
	s_6	ε	[Input(uid, hello, Alice, ℓ_1, 1)]
	s_7	[(Alice, "Hi, Bob")]	[Input(uid, hello, Alice, ℓ_1, 1), Learn(Alice, $\omega(\ell_1)$, ℓ_1), Learn(Alice, $\omega(\ell_2)$, ℓ_2)]

Figure 12: The output sequence and the trace when `Alice` queries the entry point `hello` with value `1` for `uid`

Algorithm 1 Auxiliary function fp

```
1:  function fp(ρ̂, z, ℓ̄, ℓ̄′, u)
2:      μ ← ℓ̄′ ∪ ℓ̄″
3:      μ′ ← ∅
4:      ρ̃ ← ρ̂
5:      ρ̃[z]["Learn"][u] = ⟨{}, ∅⟩
6:      while μ ⊈ μ′ do
7:          ρ̃ ← ρ̃.copy()
8:          c ← ⟨{⟨[⟨ω(ℓ), ∅⟩, ⟨ℓ, ∅⟩], ∅⟩ | ℓ ∈ μ \ μ′}, μ⟩
9:          // ⊎ computes the union of both dictionaries and labels
10:         ρ̃[z]["Learn"][u] ← ρ̃[z]["Learn"][u] ⊎ c
11:         μ′ ← μ
12:         ⟨b, π⟩ ← M̃_Φ(ρ̃)
13:         if b = ⊤ then
14:             μ ← μ ∪ π
15:         else
16:             return ⟨ρ̂, ⟨⊥, μ⟩⟩
17:     return ⟨ρ̃, ⟨⊤, μ⟩⟩
```

a copy of the current trace, are initialized (lines 3–4). Then, we log a `Learn` event to $\hat{\rho}$ for each input label in $\mu \setminus \mu'$ (line 7–10), update μ' (line 11), and ask the monitor for its verdict (line 12). If no violation is found, the set of input labels π returned by the monitor is added to μ (line 14). These steps are repeated until either no new input label is output by the monitor, or a violation is detected. If the monitor detects a violation, the function fp returns with the labeled verdict $\langle \bot, \mu\rangle$ and the original trace (l. 16). Otherwise, since μ is strictly increasing and the number of labels in the system at a given point in time is finite, the loop's execution always terminates. In this case, the trace is updated and the verdict $\langle \top, \mu\rangle$ is returned along with the updated trace (l. 17).

5.2 Formal guarantees

Dmol enforces a class of system properties defined by users' sMFOTL policies over `Input` and `Learn` predicates. Specifically, a *system property* is a subset of $\text{Seq}\,(\text{Seq}\,(\mathbb{S} \times \text{Seq}\,(\mathbb{O})))$, whereas a *trace property* is a subset of $\mathbb{T}$. We say that the state machine *enforces* a system property P iff, for any $q \in \text{Seq}\,(\mathbb{Q})$, $\text{run}(q) \in P$.

When users write their sMFOTL policies, they assume some "ideal" semantics of events: `Input` is logged when some input takes place, and `Learn` is logged iff a user learns some information about an input. Such an event semantics can be formalized as an *abstract trace*, which might differ from the actual trace stored in individual states. The system property to be enforced then consists of all executions whose abstract trace is a model of Φ. Here, for each $X := \text{run}(q)$, where $q \in \text{Seq}\,(\mathbb{Q})$ and $k := |q|$, the abstract trace $\text{atrace}(X)$ is constructed using the following two rules:

R1) For a query $q_m := (z, u, e, ((a_1, i_1), \ldots, (a_k, i_k)))$ called in the final state s, all events $\texttt{Input}(a_j, e, u, \ell_{x_j}, i_j)$, for $1 \le j \le k$, are logged to $\text{atrace}(X)$ at timestamp z in $\text{in}(s, q_m)$

R2) Consider $X_j = ((s_1, o_1), \ldots, (s_m, o_m))$, the jth sequence of X. If there exists $q' \in \text{Seq}\,(\mathbb{Q})$ and $u \in \mathbb{U}$ such that (i) q' equals q except for some input i_{ab} at position ab in q, which differs from input i'_{ab} at position ab in q'; and (ii) $\text{filter}(o_m, u)$ cannot be observed in $\text{run}(q')_j$, then $\texttt{Learn}(u, \omega(\ell_{i_{ab}}), \ell_{i_{ab}})$ is logged to $\text{atrace}(X)$ at the current timestamp in q_j.

Rule R1 provides a similar rule for logging `Input` events as before. Rule R2 describes the conditions under which an adversary can learn information about the value of some input i_{ab} according to

TINI. Namely, when the adversary receives a sequence of outputs that she *could not have observed* for *at least one* value of i_{ab}, we consider that she has learnt information about i_{ab}.

Given our termination-insensitive approach, a sequence of values $\text{filter}(o_m, u)$ *cannot be observed* in some $((s'_1, o'_1), \ldots, (s'_{m'}, o'_{m'}))$ iff there does not exist $1 \leq c \leq m'$ such that $\text{filter}(o_m, u) \preceq \text{filter}(o'_c, u)$, i.e., iff no prefix of a sequence of values received by u in s' is equal to $\text{filter}(o_m, u)$.

For some Φ, we finally the define system property $\tilde{P}$ as:

$$\tilde{P} = \left\{X \in \text{Seq}\left(\text{Seq}\left(\mathbb{S} \times \text{Seq}\left(\mathbb{Q}\right)\right)\right) \mid \forall v, i.\ v, i \models_{\text{atrace}(X)} \Phi\right\}.$$

Given $e \in \mathbb{E}$, we say that a policy φ is e-decreasing iff for all ρ and ρ' where ρ' is obtained by removing some occurrences of events with name e from ρ, we have $v, i \models_\rho \varphi \Rightarrow v, i \models_{\rho'} \varphi$. Here, since `Learn` provides only an *overapproximation* of information flows, we require Φ to be `Learn`-decreasing in order to ensure correctness.

We are now ready to show our main theorem:

Theorem 5.7 (Correctness). *If Φ is enforceable and* `Learn`*-decreasing, Dmol's semantics enforce $\tilde{P}$,*

Proof. See Appendix A.2 □

Observe how the above theorem matches the adversary model given in the previous section. First, the inputs that users can learn are captured by our definition of atrace, which is similar to TINI; this approach is sound since we assumed that users have no access to termination, error, and timing channels. Theorem 5.7 shows that user-specified policies are never violated on the abstract trace. Hence, even if the adversary corrupts users, she cannot learn more than what these users are allowed to learn. Second, the code deployed by service providers is arbitrary Dmol code. Therefore, an adversary corrupting service providers cannot get any more information than the agents it corrupts. Third and lastly, communication channels and other Databank components are considered trusted, and are therefore "baked into" Dmol's semantics.

5.3 From Dmol to DPython

We straightforwardly adapted the semantics of Dmol to implement DPython, a Python fragment with primitives for web programming and interaction with a database. DPython's syntax is given in Figure 13. Next, we discuss differences between Dmol and DPython.

In DPython, functions (Eq. 51) can optionally be marked as entry points queryable by users under a specific URL string. This is achieved using the decorator `route` from an imported *app* object. Unlike in Dmol, all program variables are now global. Global information can be stored in the database (using SQL queries, see below), or in user-specific session variables (Eq. 76, 77). Inputs can also be passed to functions through GET and POST arguments (74, 75). Sets, lists, and tuples are available in addition to dictionaries (Eq. 66–68). The label-propagation mechanisms are as in Dmol.

Databases can also be encoded as (labeled) dictionaries: a first level of keys maps table names to tables, which are mappings from pairs of row ids and field names to field values. In DPython, the database can be queried in SQL using the function `sql`. It takes two arguments: a parametric SQL query string and a list of arguments. Parameters of the form `?0`, `?1`, …, `?k` indicate which arguments from the list are to be inserted into the query. Requiring prepared

$$
\begin{aligned}
prog &::= \left|\begin{array}{l}\texttt{from apps import}\langle app\rangle\\ fun\\ \ldots\\ fun\end{array}\right. && (50)\\
fun &::= \left|\begin{array}{l}[\![\texttt{@}\langle app\rangle\texttt{.route(}url\texttt{)}]\!]\\ \texttt{def } ident\texttt{(}ident\texttt{,}\\ \quad\ldots\texttt{,}ident\texttt{):}\\ \quad block\end{array}\right. && (51)\\
block &::= \left|\begin{array}{l}stmt\\ \ldots\\ stmt\end{array}\right. && (52)\\
assign &::= lhs\_expr = expr && (53)\\
stmt &::= assign && (54)\\
&\mid \texttt{return } expr && (55)\\
&\mid \left|\begin{array}{l}\texttt{if } expr\texttt{:}\\ \quad block\\ [\![\texttt{else:}\\ \quad block]\!]\end{array}\right. && (56)\\
&\mid \left|\begin{array}{l}\texttt{while } expr\texttt{:}\\ \quad block\end{array}\right. && (57)\\
&\mid \texttt{del } lhs\_expr\texttt{[}expr\texttt{]} && (58)\\
lhs\_expr &::= ident(\!|[expr]|\!)^* && (59)\\
url &::= string && (60)\\
sql\_query &::= string && (61)
\end{aligned}
$$

$$
\begin{aligned}
expr &::= const \mid ident && (62)\\
&\mid \Omega_1\ expr && (63)\\
&\mid expr\ \Omega_2\ expr && (64)\\
&\mid \{expr\texttt{:}expr, \ldots, expr\texttt{:}expr\} && (65)\\
&\mid \texttt{set(}expr, \ldots, expr\texttt{)} && (66)\\
&\mid \texttt{[}expr, \ldots, expr\texttt{]} && (67)\\
&\mid \texttt{(}expr, \ldots, expr\texttt{)} && (68)\\
&\mid expr\texttt{[}expr\texttt{]} && (69)\\
&\mid \texttt{len(}expr\texttt{)} && (70)\\
&\mid \texttt{keys(}expr\texttt{)} && (71)\\
&\mid ident\texttt{(}expr, \ldots, expr\texttt{)} && (72)\\
&\mid \texttt{external\_call(}url\texttt{,}expr\texttt{)} && (73)\\
&\mid \texttt{get(}string\texttt{)} && (74)\\
&\mid \texttt{post(}string\texttt{)} && (75)\\
&\mid \texttt{get\_session(}string\texttt{)} && (76)\\
&\mid \texttt{set\_session(}string\texttt{,}string\texttt{)} && (77)\\
&\mid \texttt{me()} && (78)\\
&\mid \texttt{sql(}sql\_query\texttt{,}expr\texttt{)} && (79)
\end{aligned}
$$

Figure 13: Syntax of DPython programs

SQL statements allows the static identification of the tables involved in the query, which is required for detecting implicit flows. The `SELECT`, `INSERT`, and `DELETE` SQL statements with `JOIN`, `ORDER BY`, and `WHERE` expressions are derived from the *sql_query* nonterminal.

Example 5.8. Example DPython code given in Figure 14 corresponds to the Dmol code from Figure 5.

6 IMPLEMENTATION

We now describe our Databank implementation, consisting of a DPython compiler, a user interface, a monitor, and a database. The source code and a ready-to-use VirtualBox image are publicly available [4]. The prototype consists of about 3,500 lines of Python 3 code. Dmol's web programming is implemented with the Flask 1.1.2 [34] web framework, and SQLite 3.31.1 is used as the database.

The compiler transforms DPython code into plain Python 3 code, making extensive use of Python's parsing and unparsing libraries. We use the `ast.parse` function to extract the abstract syntax tree (AST) of the DPython code. We then build its control flow graph, generate the compiled Python 3 AST, and finally use the function `unparse` from the `astunparse` library to generate the compiled Python 3 code. The Python 3 code produced by the compiler on the code from Figure 14 is given in Appendix B.

We have modified MonPoly's [9] monitoring algorithm [8] to account for information flow. To find violations, MonPoly first negates the given formula and attempts to satisfy it by computing

```
1 @app.route("hello/<int:uid>")
2 def hello(uid):
3   msg = "Hi"
4   name = sql("SELECT␣name␣FROM␣users␣WHERE␣uid␣=␣?0",[uid])[0][0]
5   if name != "":
6     msg = msg + ",␣" + name
7   return msg
```

Figure 14: An example DPython code

valuations for every subformula as a table. These tables are read from the trace (for predicates) or computed using table operations (using joins, anti-joins, projections, and unions) on the tables of subformulas. Our monitoring algorithm adds input labels to all tables and overloads table operations to propagate labels. The resulting monitor outputs the same verdicts as MonPoly, with the input labels corresponding to the inputs that influenced the verdict.

We provide users with an interface to login and manage their data-usage policies. Each user is identified by A unique ID, while their policies regulate data usage independently of the application.

Each database table in the Databank is extended with labels at the value and table levels. Each table required by the application logic is implemented by two tables: a main table containing values, and an auxiliary table storing corresponding input labels.

7 EVALUATION

We evaluate our Databank prototype by answering the following research questions:

- RQ1. Is DPython expressive enough to develop realistic web applications? How does using the Databank/DPython affect the development process and application design?
- RQ2. How does the Databank scale with respect to policies and application workloads? How much overhead does it incur?

To answer these questions, we develop a proof of concept system with functionalities similar to the real-world application Meetup.

Proof of concept. We develop an event management application where users can create, manage, and delete events. Each event has a name, a description, and possibly a custom location. Users can further organize events into categories, invite other users to attend their events, accept event invitations, request attendance in events they see, and approve or deny attendance requests to their events. Depending on their membership level (free user, premium user, or administrator), users experience different functional restrictions.This application is representative of typical web applications used by small- to middle-size communities. It is not, however, built for large communities, as it does not support horizontal resource scaling, load balancing, data replication, etc.

The Databank application has 1,215 lines of DPython code, which compile into 5,441 lines of Python 3 code. Overall, the code defines 43 entry points and 11 logical database tables, compiling into 22 physical tables that also store input labels. Note that the application has no client-side code, which is outside of this study's scope.

We have also developed a functionally equivalent application using Python 3, Flask, and the Object-Relational Mapper (ORM) SQLAlchemy. This baseline has 636 lines of Python 3 code and 544 lines of additional HTML templates.

Among existing approaches, Riverbed [56] is most similar to ours, but its source code is not publicly available. Note, however, that our proof of concept provides more functionality than Minitwit, the Twitter-like web application used—along with two other non-web examples—to evaluate Riverbed [56]. For completeness, we have also ported the Minitwit application to DPython [4]. It has 170 lines of DPython code, which compile into 1021 lines of Python 3 code.

$$\varphi_u^0 := \top$$

$$\varphi_u^1 := \forall v, \ell.\, \texttt{Learn}(v, u, \ell) \wedge \blacklozenge_{(2w,\infty)}\, \texttt{InputL}(\texttt{text}, \texttt{post}, u, \ell) \Rightarrow v \approx u.$$

$$\begin{aligned}\varphi_u^2 := \forall v, \ell.(&\texttt{Learn}(v, u, \ell) \Rightarrow (\neg \exists a.\blacklozenge \texttt{InputL}(a, \texttt{add\_event}, u, \ell)) \vee v \approx u \vee \\ &\neg(\neg(\texttt{InputV}(\texttt{ID}, \texttt{add\_friend}, v) \vee \\ &\quad \texttt{InputV}(\texttt{user}, \texttt{delete\_friend}, v)) \,\mathsf{S}\, \texttt{InputV}(\texttt{user}, \texttt{delete\_friend}, v))) \vee \\ &(\texttt{Learn}(v, u, \ell) \Rightarrow (\neg \exists a.\blacklozenge \texttt{InputL}(a, \texttt{add\_event}, u, \ell)) \vee v \approx u \vee \\ &\neg \blacklozenge \texttt{InputL}(\texttt{user}, \texttt{delete\_friend}, v))\end{aligned}$$

Figure 15: Data-usage policies for a user u

7.1 RQ1: Expressiveness and design decisions

Expressiveness and succinctness. The DPython language is sufficient to develop the case study described above. The lack of `for` loops, external module imports, a template system, an ORM, and `UPDATE` SQL queries can lead to less succinct or modular code compared to the baseline. However, these are due only to limitations in our prototype and are not inherent to the Databank model.

Testing and deployment. DPython's semantics is user-controlled. Furthermore, many expressions that would throw errors in Python evaluate to a default value. This makes DPython code slightly more challenging to debug than standard Python code; ideally, DPython code should always be tested using a standard Python interpreter first. Additional testing is required to ensure that policy violations are handled gracefully and to avoid inconsistent application states.

Design decisions. While it is easy to syntactically port Python code to DPython, in practice, the development of our Databank application led to different design decisions compared to the baseline in order to preserve usability despite heterogenous user policies. Next, we discuss one such case of a Databank-specific design.

On pages showing data owned by multiple users (e.g., listing events from a specific category), a violation of a single user's data-usage policy may disallow the entire page to be served. Note that this is intended, as it follows what the data-usage policy really prescribes. In general, strictness of data usage policies affects the usability of Databank applications, but users are now free to find their own balance between privacy and usability, or even negotiate it with the service provider more transparently. On the other hand, service providers may introduce features that facilitate such negotiation. For instance, we introduced a directed *friendship* relation within the logic of our Databank application. Users may add and remove other users as their friends, which is then used as a content filter: only data owned by the friends of the current user are shown. Therefore, by managing their friends, users can ensure to see events that are not restricted to them by data-usage policies.

A similar design decision is already present in the Minitwit application, where "following" a user replaces friendship, and user timelines only aggregate posts of accounts a user follows.

7.2 RQ2: Runtime performance

Experimental setup. We assess the performance of our Databank application and its baseline using the following input workloads:

- INSERT, DELETE, or VIEW a single event in the system;
- LIST all events in a system; and
- COMPLEX workload: A user u_1 logs in, creates a new event e and invites u_2 to e; then u_2 logs in and accepts the invitation.

We consider the performance of three different systems: (D) the

(a) **Overhead** $O_{(D)/S}$: $u = 10$, $p = \varphi_u^1$, $r = 100$

n	0		25		50		100		250		500		Plot
S	(F)	(P)	(F)	(P)	(F)	(P)	(F)	(P)	(F)	(P)	(F)	(P)	
Insert	.38	.02	.56	-.01	.62	.11	.67	.16	.71	.23	.98	.45	
Delete	n.a.	n.a.	.87	.03	.88	.13	1.06	.26	1.13	.26	.95	.25	
View	n.a.	n.a.	**.77**	**.07**	.88	.13	.90	.18	**1.06**	**.29**	.83	.13	16d
List	.09	.05	.37	.19	**.65**	.49	1.18	1.04	2.80	2.79	**6.48**	5.21	16f
Complex	**.16**	.06	.32	.24	.34	.19	.78	.73	.73	.15	**1.93**	.72	16g

(b) **Overhead** $O_{(D)/S}$: $n = 50$, $p = \varphi_u^1$, $r = 20$

u	2		10		50		100		500		Plot
S	(F)	(P)	(F)	(P)	(F)	(P)	(F)	(P)	(F)	(P)	
Insert	.43	-.06	.62	.11	.69	.06	.45	.03	.88	.01	
Delete	.78	.13	.88	.13	.96	.20	1.19	.22	.88	.13	
View	.81	.13	.88	.13	.88	.16	.76	.19	1.16	.10	
List	.79	.66	.65	.49	.64	.42	.46	.35	.98	.46	16e
Complex	.72	.47	.34	.19	.44	.34	.71	.41	.70	.43	16h

(c) **Overhead** $O_{(D)/S}$: $n = 50$, $u = 10$, $r = 20$

p	φ_u^0		φ_u^1		φ_u^2		Plot
S	(F)	(P)	(F)	(P)	(F)	(P)	
Insert	.57	-.02	.62	.11	.56	.05	
Delete	.86	.13	.88	.13	.88	.16	
View	.82	.14	.88	.13	.87	.16	
List	.63	.51	.65	.49	.82	.44	
Complex	.39	.28	.34	.19	.64	.37	16i

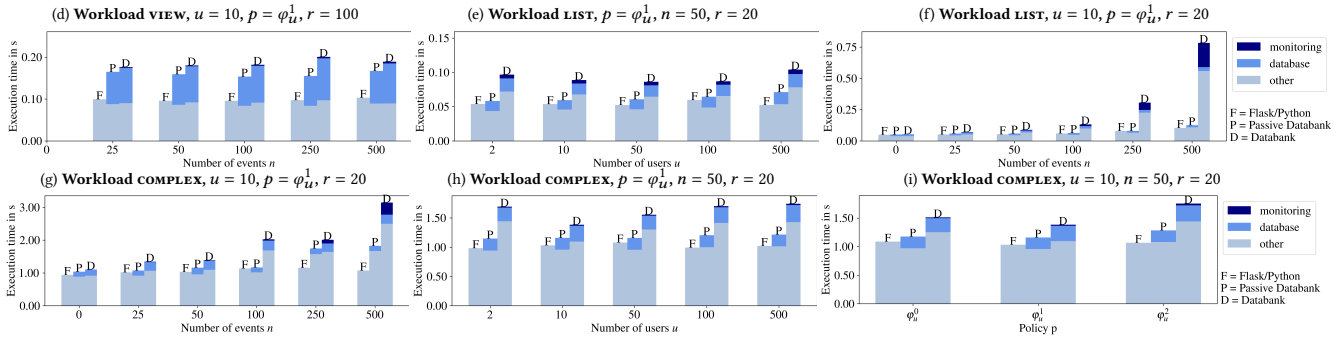

Figure 16: Overhead and execution time of our Databank application averaged over r repetitions

DPython application run on the Databank; (P) the DPython application run on a *passive* Databank, with both label propagation and monitoring turned off; and (F) Flask/Python baseline application.

We vary the number n of already created events, the number u of users, and, for (D) only, the user policy $p \in \{\varphi_u^0, \varphi_u^1, \varphi_u^2\}$ (see below, identical for all users in $\mathbb{U}$). Every workload is executed using a headless browser on a higher-end laptop (Intel Core i5, 32 GB RAM). Besides the total execution time, for (P) and (D) we measure database querying time. For (D), we also show monitoring time.

Policies. We have formalized three data-usage policies shown in Figure 15. Policy φ_u^0 allows all data uses; φ_u^1 is the policy from Section 4; and φ_u^2 requires that to see an event, a user must have been added as a friend by the user who created the event, and this friendship relationship should not yet have been revoked.

Results. Execution times and overheads are summarized in Figure 16. The overhead of system x with execution time t_x compared to system y with execution time t_y is $O_{x/y} = \frac{t_x}{t_y} - 1$. We show values averaged over $r = 100$ repetitions for every experiment in Figure 16a, and over $r = 20$ repetitions for each experiment in Figures 16b and 16c. The tables show the overheads $O_{(D)/(F)}$ and $O_{(D)/(P)}$, while the plots show selected absolute figures.

For the three simple workloads, INSERT, DELETE, and VIEW, all three systems exhibit constant execution time when varying n, p, and u. The VIEW workload is shown in Figure 16d. The time spent on monitoring is below 5%. The Databank application (D) has an overhead of 77%–106% compared to (F) and 7%–29% compared to (P). Hence, the majority of the overhead (30%–100%) comes from the lack of features essential for web application performance (e.g., ORM, or a template system). However, the performance of the Databank does not impede usability, as for all values of the parameters, the pages are always loaded in under 0.2 seconds (Figure 16d).

For the LIST workload, the three systems' execution time is linear with respect to n (Figure 16f), and constant with respect to p (Figure 16c) and u (Figure 16e). This is expected since the data to be shown grows linearly with n. (D)'s overhead increases from 65% for $n = 50$, to 648% for $n = 500$. Namely, n also linearly increases the number of input labels to be propagated and logged, causing the additional work for label propagation and monitoring. The performance of (P) confirms this, as it shows negligible overhead with respect to (F). Such a situation can be avoided by limiting the number of items shown simultaneously, e.g. by introducing pagination.

When executing on the COMPLEX workload, the overhead is unaffected by changes in u (Figure 16h) and p (Figure 16i), but grows with n, reaching 193% for $n = 500$ (against 16% for $n = 2$), of which 40% are caused by the lack of efficient language features, while the rest is due to label propagation and monitoring. For all values of the parameters, (D) remains usable, as the average time for COMPLEX remains below 3 seconds (much less than human users would need).

Overall, our evaluation demonstrates the scalability of the Databank model. In none of the cases considered did the overhead threaten usability. Moreover, this overhead could be further reduced (to under 40%, see runtime on LIST for $n \leq 25$) by avoiding loading large batches of data and introducing pagination whenever possible.

8 CONCLUSION AND FUTURE WORK

We introduced the Databank model, an architecture that enforces data ownership rights against malicious service providers. This is achieved by delegating data storage and code execution to a user-trusted third party, the Databank. The Databank runs code written in DPython, a novel Python fragment with formally specified semantics. We developed a Databank prototype and evaluated its performance on a proof of concept application.

To the best of our knowledge, this is the first web architecture providing formal guarantees on policy enforcement, allowing users to specify application-agnostic data-usage policies, and allowing service providers to write policy-agnostic code.

In the future, we will generalize the model to a decentralized setting, extend DPython to support all Python 3 constructs, improve the label propagation precision using static IFC, add transactional memory semantics for DMOL's entry points, and add more events users can utilize to write policies (e.g., for purpose-based usage [1]).

REFERENCES

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

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

[3] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. 2013. Faceted Execution of Policy-Agnostic Programs. In *8th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, Prasad Naldurg and Nikhil Swamy (Eds.). ACM, 15–26. https://doi.org/10.1145/2465106.2465121

[4] Anonymous Author(s). 2021. Databank (Git repository). https://github.com/databank-anon/databank.

[5] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. *Lectures on Runtime Verification*. LNCS, Vol. 10457. Springer, Chapter Introduction to runtime verification, 1–33. https://doi.org/10.1007/978-3-319-75632-5_1

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

[7] David Basin, Vincent Jugé, Felix Klaedtke, and Eugen Zălinescu. 2013. Enforceable Security Policies Revisited. *ACM Trans. Inf. Syst. Secur.* 16, 1, Article 3 (June 2013), 26 pages. https://doi.org/10.1145/2487222.2487225

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

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

[10] Andrea Boerding, Nicolai Culik, C. Doepke, T. Hoeren, Tim Juelicher, Charlotte Roettgen, and Max V. Schoenfeld. 2018. Data Ownership—A Property Rights Approach from a European Perspective. *Journal of Civil Law Studies* 11 (2018), 5.

[11] Feng Chen and Grigore Roşu. 2005. Java-MOP: A Monitoring Oriented Programming Environment for Java. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Nicolas Halbwachs and Lenore D. Zuck (Eds.). Springer, 546–550. https://doi.org/10.1007/978-3-540-31980-1_36

[12] Feng Chen and Grigore Roşu. 2003. Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. *Electron. Notes Theor. Comput. Sci.* 89, 2 (2003), 108–127. https://doi.org/10.1016/S1571-0661(04)81045-4

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

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

[15] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. 2007. Secure web application via automatic partitioning. In *21st ACM Symposium on Operating Systems Principles (SOSP)*, Thomas C. Bressoud and M. Frans Kaashoek (Eds.). ACM, 31–44. https://doi.org/10.1145/1294261.1294265

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

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

[18] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects (FMCO, Vol. 4709)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer, 266–296. https://doi.org/10.1007/978-3-540-74792-5_12

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

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

[21] Yves-Alexandre de Montjoye, Erez Shmueli, Samuel S. Wang, and Alex Sandy Pentland. 2014. openPDS: Protecting the Privacy of Metadata through SafeAnswers. *PLOS ONE* 9, 7 (07 2014), 1–9. https://doi.org/10.1371/journal.pone.0098790

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

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

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

[25] Ulfar Erlingsson. 2004. *The inlined reference monitor approach to security policy enforcement.* Ph.D. Dissertation.

[26] Úlfar Erlingsson and Fred B. Schneider. 2000. IRM Enforcement of Java Stack Inspection. In *2000 IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 246–255. https://doi.org/10.1109/SECPRI.2000.848461

[27] Yliès Falcone, Srđan Krstić, Giles Reger, and Dmitriy Traytel. 2021. A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.* 23, 2 (2021), 255–284. https://doi.org/10.1007/s10009-021-00609-z

[28] Maryam Farboodi and Laura Veldkamp. 2021. *A Growth Model of the Data Economy.* Technical Report. National Bureau of Economic Research.

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

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

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

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

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

[34] Miguel Grinberg. 2018. *Flask web development: developing web applications with python.* O'Reilly Media, Inc.

[35] William Heath, David Alexander, and Phil Booth. 2013. Digital Enlightenment, Mydex, and restoring control over personal data to the individual. In *Digital Enlightenment Forum Yearbook*. 253–269.

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

[37] Stefan Heule, Deian Stefan, Edward Z. Yang, John C. Mitchell, and Alejandro Russo. 2015. IFC Inside: Retrofitting Languages with Dynamic Information Flow Control. In *Principles of Security and Trust*, Riccardo Focardi and Andrew Myers (Eds.). Springer, 11–31.

[38] Maxwell Krohn, Alex Yip, Micah Brodsky, Robert Morris, Michael Walfish, et al. 2007. A world wide web without walls. In *6th ACM Workshop on Hot Topics in Networking (Hotnets)*, Constantine Dovrolis, Vern Paxson, and Stefan Savage (Eds.). ACM SIGCOMM.

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

[40] Tuukka Lehtiniemi. 2017. Personal data spaces: An intervention in surveillance capitalism? *Surveillance & Society* 15, 5 (2017), 626–639. https://doi.org/10.24908/ss.v15i5.6424

[41] Miti Mazmudar and Ian Goldberg. 2020. Mitigator: Privacy policy compliance using trusted hardware. *Proc. Priv. Enhancing Technol.* 2020, 3 (2020), 204–221. https://doi.org/10.2478/popets-2020-0049

[42] Andrew C Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2001. Jif: Java information flow. Software release.

[43] Irene C L Ng et al. 2013. Making Value Creating Context Visible for New Economic and Business Models: Home Hub-of-all-Things (HAT) as Platform for Multi-Sided Market Powered by Internet-of-Things. In *Panel Session at The Future of Value Creation in Complex Service Systems Minitrack of Hawaii International Conference on Systems Science (HICSS), January*. 7–10.

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

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

[46] Shariq Rizvi, Alberto O. Mendelzon, S. Sudarshan, and Prasan Roy. 2004. Extending Query Rewriting Techniques for Fine-Grained Access Control. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, Gerhard Weikum, Arnd Christian König, and Stefan Deßloch (Eds.). ACM, 551–562. https://doi.org/10.1145/1007568.1007631
[47] Grigore Roşu. 2012. On Safety Properties and Their Monitoring. *Sci. Annals of Computer Science* 22, 2 (2012), 327–365. https://doi.org/10.7561/SACS.2012.2.327
[48] Stefano Rodotà. 2009. Data protection as a fundamental right. In *Reinventing Data Protection?* Springer, 77–82.
[49] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE J. Sel. Areas Commun.* 21, 1 (2003), 5–19. https://doi.org/10.1109/jsac.2002.806121
[50] Andrei Sabelfeld and David Sands. 2001. A per model of secure information flow in sequential programs. *High. Order Symb. Comput.* 14, 1 (2001), 59–91.
[51] Andrei Vlad Sambra et al. 2016. *Solid: a platform for decentralized social applications based on linked data.* Technical Report. MIT CSAIL & Qatar Computing Research Institute.
[52] Fred B. Schneider. 2000. Enforceable Security Policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (Feb. 2000), 30–50. https://doi.org/10.1145/353323.353382
[53] Daniel Schoepe, Daniel Hedin, and Andrei Sabelfeld. 2014. SeLINQ: Tracking Information across Application-Database Boundaries. In *19th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 25–38. https://doi.org/10.1145/2628136.2628151
[54] Malte Schwarzkopf, Eddie Kohler, M Frans Kaashoek, and Robert Morris. 2019. Position: GDPR Compliance by Construction. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*, Vijay Gadepally, Timothy Mattson, Michael Stonebraker, Fusheng Wang, Gang Luo, Yanhui Laing, and Alevtina Dubovitskaya (Eds.). LNCS, Vol. 11721. Springer, 39–53. https://doi.org/10.1007/978-3-030-33752-0_3
[55] Vincent Simonet. 2003. The Flow Caml System (version 1.00): Documentation and user's manual. http://www.normalesup.org/~simonet/soft/flowcaml/manual/
[56] Frank Wang, Ronny Ko, and James Mickens. 2019. Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 615–630. https://www.usenix.org/conference/nsdi19/presentation/wang-frank
[57] Yuxin Wang, Zeyu Ding, Yingtai Xiao, Daniel Kifer, and Danfeng Zhang. 2021. DPGen: Automated Program Synthesis for Differential Privacy. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 393–411. https://doi.org/10.1145/3460120.3484781
[58] Benjamin Weggenmann and Florian Kerschbaum. 2021. Differential Privacy for Directional Data. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 1205–1222. https://doi.org/10.1145/3460120.3484734
[59] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, Dynamic Information Flow for Database-Backed Applications. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chandra Krintz and Emery Berger (Eds.). ACM, 631–647. https://doi.org/10.1145/2908080.2908098

A PROOFS OF THEOREMS

A.1 Termination-insensitive noninterference

First, note that functions fp and sanitize are side-effect-free, and one can easily check that for $w, w' \in \mathcal{L}$ and $t, t' \in \mathbb{S}$ such that $t \sim_{w,w'} t$, $\langle \sigma, u, j, m, z, \hat{\rho}, \lambda\rangle := t$, $\langle \sigma', u, j', m', z, \hat{\rho}', \lambda'\rangle := t'$:

- For all $U \subseteq \mathbb{U}$ and $\overline{\ell}, \overline{\ell}' \subseteq \mathcal{L}$ such that $\langle \bot, \overline{\ell}\rangle =_{w,w'} \langle \bot, \overline{\ell}'\rangle$,
$$\mathsf{sanitize}(\hat{\rho}, U, \overline{\ell}, z) =_{w,w'} \mathsf{sanitize}(\hat{\rho}', U, \overline{\ell}', z);$$
- For all $u \in \mathbb{U}$ and $\overline{\ell}, \overline{\ell}' \subseteq \mathcal{L}$ such that $\langle \bot, \overline{\ell}\rangle =_{w,w'} \langle \bot, \overline{\ell}'\rangle$,
$$\mathsf{fst}(\mathsf{fp}(\hat{\rho}, z, \overline{\ell}, u)) =_{w,w'} \mathsf{fst}(\mathsf{fp}(\hat{\rho}, z, \overline{\ell}, u));$$
$$\mathsf{snd}(\mathsf{fp}(\hat{\rho}, z, \overline{\ell}, u)) =_{w,w'} \mathsf{snd}(\mathsf{fp}(\hat{\rho}, z, \overline{\ell}, u)).$$

Thus, these two functions behave like regular Dmol operators.

We will need four technical lemmas:

Lemma A.1. *Let $s, s' \in \mathbb{S}$ and $w, w' \in \mathcal{L}$ such that $s \sim_{w,w'} s'$, $u \in \mathbb{U}$, $z \in \mathbb{N}$ and e a Dmol expression.*

Then $[\![e, u, \mathrm{mem}(s), z]\!] =_{w,w'} [\![e, u, \mathrm{mem}(s'), z]\!]$.

Proof. Let $m = \mathrm{mem}(s)$, $m' = \mathrm{mem}(s')$, $\langle v, j\rangle = [\![e, u, m, z]\!]$ and $\langle v', j'\rangle = [\![e, u, m', z]\!]$.

The proof is by induction over expressions.

If $e = c$, then $\langle v, j\rangle = \langle c, \emptyset\rangle = \langle v', j'\rangle$ and therefore $\langle v, j\rangle =_{w,w'} \langle v', j'\rangle$.

If $e = i$ with $i \in \mathrm{dom}\ m$, then $m \sim_{w,w'} m'$ yields $i \in \mathrm{dom}\ m' = \mathrm{dom}\ m$ and $\langle v, j\rangle = m[i] =_{w,w'} m'[i] = \langle v', j'\rangle$.

If $e = i$ with $i \notin \mathrm{dom}\ m$, then similarly $i \notin \mathrm{dom}\ m'$, hence $\langle v, j\rangle = \langle \bot, \emptyset\rangle = \langle v', j'\rangle$ and $\langle v, j\rangle =_{w,w'} \langle v', j'\rangle$.

If $e = \omega\ e_1$ with $\omega \in \Omega_1$, assume that $[\![e_1, u, m, z]\!] =: \langle v_1, i_1\rangle =_{w,w'} \langle v_1', i_1'\rangle := [\![e_1, u, m', z]\!]$. If $w \notin i_1$ and $w' \notin i_1'$, we first consider the case where v_1 and v_1' are not dictionaries. In this case, we get $\langle v_1, i_1\rangle = \langle v_1', i_1'\rangle$. By rule 23 (left), $v = \omega(v_1) = \omega(v_1') = v'$ and $j = i_1 = i_1' = j'$. Hence $\langle v, j\rangle =_{w,w'} \langle v', j'\rangle$ by case 2 in the definition of $=_{w,w'}$. Then, while still assuming $w \notin i_1$, we consider the case where v_1 and v_1' are dictionaries. In that case, $\mathrm{dom}\ v_1 = \mathrm{dom}\ v_1'$. Since the result of applying ω to a dictionary argument only depends on the keys of this argument, we obtain $v = v'$, $j = j'$, and conclude as above. If $w \in i_1$ and $w' \in i_1'$, since $j = i_1$ and $j' = i_1'$ by rule 23 (left), we have $\langle v, j\rangle =_{w,w'} \langle v', j'\rangle$ by case 1.

If $e = e_1\ \omega\ e_2$, the proof is similar to the previous case, modulo a straightforward case distinction over $w \in i_1 \wedge w' \in i_1'$ and $w \in i_2 \wedge w' \in i_2'$.

If $e = \{k_1{:}e_1, \ldots, k_n{:}e_n\}$ with, for all $1 \le p \le n$,

$$[\![e_p, u, m, z]\!] =: c_p =_{w,w'} c_p' := [\![e_p, u, m', z]\!]$$
$$[\![k_p, u, m, z]\!] =: \langle \gamma_p, \iota_p\rangle =_{w,w'} \langle \gamma_p', \iota_p'\rangle := [\![k_p, u, m', z]\!].$$

If $w \notin j = \cup_{p=1}^{n} \iota_p$, then for all $1 \le p \le n$, $w \notin \iota_p$ and $w' \notin \iota_p'$. If for some $1 \le p \le n$, γ_p is a dictionary, then γ_p' is as well. In any case, $\iota_p = \iota_p'$ for all $1 \le p \le n$, and therefore $j = j'$. If some γ_p and γ_p' are dictionaries, then $v = \bot = v'$ by rule 25 and we get $\langle v, j\rangle =_{w,w'} \langle v', j'\rangle$. Otherwise, $c_p =_{w,w'} c_p'$ for all $1 \le p \le n$, and $\langle v, j\rangle =_{w,w'} \langle v', j'\rangle$ by case 3 and rule 24. If $w \in j$, then there exists $1 \le p \le n$ such that $w \in \iota_p$; for this same p, we also have $w' \in \iota_p'$, yielding $w' \in j'$. Finally, $\langle v, j\rangle =_{w,w'} \langle v', j'\rangle$ by case 1.

If $e = e_1[e_2]$ with

$$[\![e_1, u, m, z]\!] =_{w,w'} [\![e_1, u, m', z]\!]$$
$$[\![e_2, u, m, z]\!] =_{w,w'} [\![e_2, u, m', z]\!],$$

assume first that e_1 evaluates to a dictionary. If the LHS applies rule 26, we have

$$[\![e_1, u, m, z]\!] =: \langle \{k_1 \mapsto \langle v_1, i_1\rangle, \ldots, k_n \mapsto \langle v_n, i_n\rangle\}, i'\rangle$$
$$[\![e_2, u, m, z]\!] =: \langle k_p, y\rangle,\ 1 \le p \le n.$$

If $w \notin j = i_p \cup i' \cup y$, then in particular $w \notin i'$ and $w \notin y$, and we obtain

$$[\![e_1, u, m', z]\!] =: \langle \{k_1 \mapsto \langle v_1', i_1'\rangle, \ldots, k_n \mapsto \langle v_n', i_n'\rangle\}, i''\rangle \quad w' \notin i''$$
$$[\![e_2, u, m', z]\!] =: \langle k_p', y'\rangle,\ 1 \le p \le n \quad w' \notin y'$$
$$\langle v_p, i_p\rangle =_{w,w'} \langle v_p', i_p'\rangle\ \forall 1 \le p \le n.$$

Moreover, $w \notin i_p$. As $\langle v_p, i_p\rangle =_{w,w'} \langle v_p', i_p'\rangle$, we get $v_p = v_p'$ and $i_p = i_p'$. Finally, $w \notin y$, which implies $k_p = k_{p'}'$ and $y = y'$. Hence,

$$v' = v_p' = v_p = v$$
$$j' = i_p' \cup i'' \cup y' = i_p \cup i \cup y = j$$

and $\langle v', j' \rangle =_{w,w'} \langle v, j \rangle$. If $w \in j$, then $w' \in i_p$ or $w' \in i'$ or $w' \in y$, and $\langle v, j \rangle =_{w,w'} \langle v', j' \rangle$.

In the other case corresponding to $e = e_1[e_2]$ (rule 27), the proof is similar.

If e_1 evaluates to an atomic cell, let $[[e_1, u, m, z]] =: \langle v_1, i_1 \rangle$ with $v_1 \in \mathbb{D}$ and $[[e_1, u, m', z]] =: \langle v_1', i_1' \rangle$. If $w \notin i_1$ and $w' \notin i_1'$, then $v_1 = v_1'$, $i_1 = i_1'$ and v_1' is atomic. Hence rule 27 applies in both cases, and we get $\langle v, j \rangle = \langle \bot, i_1 \rangle =_{w,w'} \langle \bot, i_1' \rangle = \langle v', j' \rangle$. If $w \in i_1$ and $w' \in i_1'$, then $w \in j$ and $w' \in j'$ and $\langle v, j \rangle =_{w,w'} \langle v', j' \rangle$.

If $e = \texttt{keys}\ e_1$ or $e = \texttt{len}\ e_1$, the proof for unary operators can be reused, since keys and len conform to our definition of unary operators as arity-1 total function of values and keys of dictionaries.

If $e = \texttt{me}$, then $\langle v, j \rangle = \langle v', j' \rangle$ and therefore $\langle v, j \rangle =_{w,w'} \langle v', j' \rangle$. □

Lemma A.2. *Let $w, w' \in \mathcal{L}$. Let m and m' be memory states with $m \sim_{w,w'} m'$, and $k \in \operatorname{dom} m \cap \operatorname{dom} m'$. Then either (i) $m[k] =_{w,w'} m'[k]$ or (ii) there exists $k' \prec k$ such that w is a label of $m[k']$ and w' is a label of $m'[k']$.*

Proof. Let $n \in \mathbb{N}$, $i \in \mathcal{G} \cup \mathcal{H}$, $(v_1, \ldots, v_n) \in \mathbb{D}^n$ such that $k = i[v_1] \ldots [v_n]$. The proof is by induction on n.

If $n = 0$, as $k \in \operatorname{dom} m \cap \operatorname{dom} m'$, we have $k = i \in \operatorname{dom} m \cap \operatorname{dom} m'$ and $m[k] =_{w,w'} m'[k']$ by definition of $\sim_{w,w'}$. Hence, $m[k] =_{w,w'} m'[k]$.

Let $n > 0$. If the property holds for all $n' < n$, then as $k \in \operatorname{dom} m \cap \operatorname{dom} m'$, we have $k^* := i[v_1] \ldots [v_{n-1}] \in \operatorname{dom} m \cap \operatorname{dom} m'$. By IH, we either have $m[k^*] =_{w,w'} m'[k^*]$ or there exists $k' \prec k^*$ such that w is a label of $m[k']$ and w' is a label of $m'[k']$. In the first case, as $m[k^*]$ is a dictionary, then either w is a label of $m[k^*]$ and w' is a label of $m'[k^*]$, in which case we have (ii) with $k^* \prec k$, or $m[k] = m[k^*][v_n] =_{w,w'} m'[k][v_n] = m'[k]$ and we have (i). In the second case, $k' \prec k^* \prec k$, and we also have (ii). □

Lemma A.3. *Let $w, w' \in \mathcal{L}$, and m, m' two memory states such that $m \sim_{w,w'} m'$. Then:*

1. *For all $V \subseteq \mathcal{G} \cup \mathcal{H}$, $m|_V \sim_{w,w'} m'|_V$.*
2. *For all memory states n, n' such that $n \sim_{w,w'} n'$ and the initialized variables of m and n are disjoint, we have $m \cup n \sim_{w,w'} m' \cup n'$.*
3. *For all $k \in \operatorname{dom} m \cap \operatorname{dom} m'$, $v \in \mathbb{D}$ and $c, c' \in C$ such that $c =_{w,w'} c'$, we have*

$$m[k \mapsto c] \sim_{w,w'} m'[k \mapsto c'].$$

If $m[k]$ and $m'[k]$ are dictionaries, we also have

$$m[k[v] \mapsto c] \sim_{w,w'} m'[k[v] \mapsto c'].$$

Moreover, for all $k \in \mathcal{G} \cup \mathcal{H}$ and c, c' as above, we have

$$m[k \mapsto c] \sim_{w,w'} m'[k \mapsto c'].$$

4. *For all $k \in \operatorname{dom} m \cap \operatorname{dom} m'$, $k' \preceq k$, $\langle v, i \rangle, \langle v_1', i_1' \rangle, \langle v_2', i_2' \rangle \in C$ such that $w \in i$, $w \in i_1'$ and $w' \in i_2'$, we have*

$$m[k' \mapsto \langle v_1', i_1' \rangle][k \mapsto \langle v, i \rangle] \sim_{w,w'} m'[k' \mapsto \langle v_2', i_2' \rangle]$$
$$m[k' \mapsto \langle v_1', i_1' \rangle] \sim_{w,w'} m'[k' \mapsto \langle v_2', i_2' \rangle].$$

Proof. (1) Let $V \subseteq \mathcal{G} \cup \mathcal{H}$. Since $\operatorname{dom} m = \operatorname{dom} m'$ by $m \sim_{w,w'} m'$, there follows $\operatorname{dom} m|_V = \operatorname{dom} m \cap V = \operatorname{dom} m' \cap V = \operatorname{dom} m|_V$. Moreover, for all $k \in \operatorname{dom} m|_V = \operatorname{dom} m'|_V \subseteq \operatorname{dom} m = \operatorname{dom} m'$, we have $v|_V[k] = v[k] =_{w,w'} v'[k] = v'|_V[k]$. Hence, $m|_V \sim_{w,w'} m'|_V$.

(2) For all $i \in \mathcal{G} \cup \mathcal{H}$, either $m[i]$ or $n[i]$ is initialized. If $m[i] \neq \langle \bot, \emptyset \rangle$, then, since $m \sim_{w,w'} m'$, $m'[i] \neq \langle \bot, \emptyset \rangle$ and $m[i] =_{w,w'} m'[i]$. Hence, $(m \cup n)[i] = m[i] =_{w,w'} m'[i] = (m' \cup n')[i]$. If $m[i] = \langle \bot, \emptyset \rangle$, then, since $m \sim_{w,w'} m'$, $m'[i] = \langle \bot, \emptyset \rangle$. Moreover, as $m \sim_{w,w'} m'$, and $n[i] =_{w,w'} n'[i]$. Hence, $(m \cup n)[i] = n[i] =_{w,w'} n'[i] = (m' \cup n')[i]$. We conclude that $m \cup n \sim_{w,w'} m' \cup n'$.

(3) Let $k = i[v_1] \ldots [v_n] \in \operatorname{dom} m \cap \operatorname{dom} m'$, $v \in \mathbb{D}$ and $(c, c') \in C^2$ such that $c =_{w,w'} c'$. Let $j \in \operatorname{dom} m = \operatorname{dom} m'$. If $i \neq j$, then

$$m[k \mapsto c][j] = m[j] =_{w,w'} m'[j] = m'[k \mapsto c'][j].$$

Now, assume $i = j$. If $n = 0$, $m[k \mapsto c][j] = c =_{w,w'} c' = m'[k \mapsto c'][j]$. Assume $n > 0$. If there is some $k' \prec k$ such that w is a label of $m[k']$ and w' is a label of $m'[k']$, select the shortest such k' and let $N < n$ such that $k' = i[v_1] \ldots [v_N]$. Otherwise, let $N := n$. We will prove by induction on $p \in \{N, \ldots, 0\}$:

$$[P(p)] \qquad m[k \mapsto c][i[v_1] \ldots [v_p]] =_{w,w'} m'[k \mapsto c'][i[v_1] \ldots [v_p]].$$

For $p = n = N$, we have

$$m[k \mapsto c][i[v_1] \ldots [v_n]] = c =_{w,w'} c' = m'[k \mapsto c'][i[v_1] \ldots [v_n]]$$

and therefore $P(n)$. For $p = N < n$, we know that w (resp. w') labels $m[k']$ (resp. $m'[k']$); therefore w (resp. w') labels $m[k \mapsto c][k']$ (resp. $m'[k \mapsto c'][k']$), as the dictionary's labels are not affected by the update, and

$$m[k \mapsto c][i[v_1] \ldots [v_N]] =_{w,w'} m'[k \mapsto c'][i[v_1] \ldots [v_N]].$$

For $p < n$, assume $P(p+1)$ and let

$$\langle x_p, y_p \rangle := m[i[v_1] \ldots [v_p]]$$
$$\langle x_p', y_p' \rangle := m'[i[v_1] \ldots [v_p]].$$

Since $m \sim_{w,w'} m'$ and our choice of N ensures that there is no prefix k' of $i[v_1] \ldots [v_n]$ such that $m[k']$ would be labelled by w and $m'[k']$ by w', Lemma A.2 yields $\langle x_p, y_p \rangle =_{w,w'} \langle x_p', y_p' \rangle$.

If $w \in y_p$, w labels $m[k \mapsto c][i[v_1] \ldots [v_p]]$, which proves $P(p)$. Otherwise, x_p and x_p' are dictionaries, $\operatorname{dom} x_p = \operatorname{dom} x_p'$, $y_p = y_p'$ and for all $k' \in \operatorname{dom} x_p$, $x_p[k'] =_{w,w'} x_p'[k']$. As $k \in \operatorname{dom} m$, we have $v_{p+1} \in \operatorname{dom} x_p$, and the only change that the update causes to $\langle x_p, y_p \rangle$ and $\langle x_p', y_p' \rangle$ concerns fields $x_p[v_{p+1}]$ and $x_p'[v_{p+1}]$. But then

$$m[k \mapsto c][i[v_1] \ldots [v_p][v_{p+1}]] =_{w,w'} m'[k \mapsto c'][i[v_1] \ldots [v_p][v_{p+1}]]$$

by IH, which proves $P(p)$. Taking $p = 0$, we get

$$m[k \mapsto c][j] =_{w,w'} m'[k \mapsto c'][j].$$

Hence $m[k \mapsto c][j] =_{w,w'} m'[k \mapsto c'][j]$. Since $\operatorname{dom} m$ and $\operatorname{dom} m'$ have not changed, we conclude that $m[k \mapsto c] \sim_{w,w'} m'[k \mapsto c]$.

To prove $m[k[v] \mapsto c] \sim_{w,w'} m'[k[v] \mapsto c]$, we additionally set $\hat{k} := k[v]$ and $v_{n+1} := v$ and consider the proof above.

Only the new base case $p = n + 1 = N$ needs to be checked again. But since

$$m[\hat{k} \mapsto c][i[v_1] \dots [v_{n+1}]] = c =_{w,w'} c'$$
$$= m'[\hat{k} \mapsto c'][i[v_1] \dots [v_{n+1}]]$$

then $P(n+1)$ is easily proven and the conclusion follows. The case of $k \in \mathcal{G} \cup \mathcal{H}$ is a straightforward consequence of the definition of memory indistinguishability.

(4) Let $k \in \text{dom } m \cap \text{dom } m'$, $k' \preceq k$, $\langle v, i\rangle, \langle v'_1, i'_1\rangle, \langle v'_2, i'_2\rangle \in C$ such that $w \in i \cap i'_1$, $w' \in i'_2$. Clearly, $\langle v'_1, i'_1\rangle =_{w,w'} \langle v'_2, i'_2\rangle$, hence by sublemma 2

$$m[k' \mapsto \langle v'_1, i'_1\rangle] \sim_{w,w'} m[k' \mapsto \langle v'_2, i'_2\rangle].$$

Now, if $k = k'$, $m[k' \mapsto \langle v'_1, i'_1\rangle][k \mapsto \langle v, i\rangle] = m[k' \mapsto \langle v, i\rangle]$, and the proof is the same. Otherwise, $k' \prec k$, and we have $k = k'[v_1] \dots [v_p]$ for some $p \in \mathbb{N}$, $p > 1$, and we easily see that

$$m[k' \mapsto \langle v'_1, i'_1\rangle][k \mapsto \langle v, i\rangle]$$
$$= m[k' \mapsto \langle v'_1[v_1[v_2] \dots [v_p] \mapsto \langle v, i\rangle], i'_1\rangle].$$

But $\langle v'_1[v_1[v_2] \dots [v_p] \mapsto \langle v, i\rangle], i'_1\rangle =_{w,w'} \langle v'_2, i'_2\rangle$, hence sublemma 2 yields

$$m[k' \mapsto \langle v'_1, i'_1\rangle][k \mapsto \langle v, i\rangle] =_{w,w'} m[k' \mapsto \langle v'_2, i'_2\rangle].$$

□

Lemma A.4. *Let $s, s' \in \mathbb{S}$ and $w, w' \in \mathcal{L}$. Let $s \sim_{w,w'} s'$, $A \subseteq \text{dom}\,(\text{mem}(s)) \cap \text{dom}\,(\text{mem}(s'))$ and $\overline{\ell}, \overline{\ell}' \subseteq \mathcal{L}$ such that $\langle \bot, \overline{\ell}\rangle =_{w,w'} \langle \bot, \overline{\ell}'\rangle$.*

Then $\text{adopt}(A, \overline{\ell}, \text{mem}(s)) \sim_{w,w'} \text{adopt}(A, \overline{\ell}', \text{mem}(s'))$.

Proof. Given the definition of adopt, it is sufficient to prove the result for $A = \{k\}$.

Let $m = \text{mem}(s)$, $m' = \text{mem}(s')$, $k \in \text{dom } m \cap \text{dom } m'$, and $A = \{k\}$. Let $\langle v, i\rangle = m[k]$ and $\langle v', i'\rangle = m'[k]$. First, by Lemma A.3, $\langle v, i\rangle =_{w,w'} \langle v', i'\rangle$. We remark that we have $\langle v, i \cup \overline{\ell}\rangle =_{w,w'} \langle v', i' \cup \overline{\ell}'\rangle$: in the case where $w \in i \cup \overline{\ell}$, clearly $w' \in i' \cup \overline{\ell}'$, and the result follows; otherwise, $\overline{\ell} = \overline{\ell}'$ (since $w \notin \overline{\ell}$ and $\langle \bot, \overline{\ell}\rangle =_{w,w'} \langle \bot, \overline{\ell}'\rangle$), $i = i'$ (since $w \notin i$ and $m[k] =_{w,w'} m'[k]$ by $m \sim_{w,w'} m'$ and Lemma A.1), $\phi(v) = \phi(v')$ where ϕ is the identity on atomic values and keys on dictionaries (for the same reason), and, if v and v' are dictionaries, their values are pairwise equivalent.

Using Lemma A.3, we get $m[k \mapsto \langle v, i \cup \overline{\ell}\rangle] \sim_{w,w'} m'[k \mapsto \langle v', i' \cup \overline{\ell}'\rangle]$. Hence, $\text{adopt}(A, \overline{\ell}, m) \sim_{w,w'} \text{adopt}(A, \overline{\ell}', m')$. □

We now sketch the proof of

Lemma 5.4. *Consider $q \in \text{Seq}\,(\mathbb{Q})$ and its copy q' where a single input i in q with input label w has been replaced by some other input i' with input label w'. Now choose $1 \le k \le |q|$ and consider the sequences of pairs of states and outputs $r = \text{run}(q)_k$ and $r' = \text{run}(q')_k$ produced by the kth query in the two scenarios. Then there exists a nondecreasing mapping $\mu : \{1, \dots, |r|\} \to \{1, \dots, |r'|\}$ such that for all $1 \le k \le |r|$, $\text{fst}(r_k) \sim_{w,w'} \text{fst}(r'_{\mu(k)})$.*

Proof sketch. Let q, q', i, i', w, w' as above.

The initial state of any run is σ_0. Moreover, function run reuses the final state returned by run1 on the previous query to initialize the next call to run1. Hence, by straightforward induction, it is sufficient to prove the following: for any $1 \le k \le |q|$, for $r = \text{run}(q)_k$, $r' = \text{run}(q')_k$, *and* $\text{fst}(r_1) \sim_{w,w'} \text{fst}(r'_1)$, there exists a nondecreasing mapping $\mu : \{1, \dots, |r|\} \to \{1, \dots, |r'|\}$ such that for all $1 \le k \le |r|$, $\text{fst}(r_k) \sim_{w,w'} \text{fst}(r'_{\mu(k)})$.

Let k, r, r' as above. For $1 \le i \le |r|$, denote by $P(i)$ the following property $P(i)$:

"one can construct $\mu_i : \{1, \dots, |i|\} \to \{1, \dots, |r'|\}$ such that for all $1 \le s \le i$,

(1) $\text{fst}(r_s) \sim_{w,w'} \text{fst}(r'_{\mu(s)})$, *and*
(2) the callers in $\text{fst}(r_s)$ *and* $\text{fst}(r_{\mu(s)})$ *are either both $\bot$ or contain two states λ and λ' such that there exists $1 \le t < s$ with $\lambda = \text{fst}(r_t)$, $\lambda' = \text{fst}(r'_{\mu(t)})$, and $P(t)$ holds using mapping $\mu_i|_{\{1,\dots,t\}}$* and
(3) either
- *the sequences of instructions to be executed next in* $\text{fst}(r_s)$ *and* $\text{fst}(r'_{\mu(s)})$ *are equal, or*
- *they contain* Ret o *and* Ret o' *with $o =_{w,w'} o'$, or*
- *one of them contains* Stop *and the other contains* Stop *or* Ret o *(resp.* Ret o'*) labeled by w (resp. w')*

and $w \notin$ all j_i."

We will construct μ incrementally on increasing subintervals $\{1, \dots, i\}$ of $\{1, \dots, |r|\}$ in two steps. We will first check that $P(1)$ holds, and then that for all $1 \le i < |r|$ such that $P(i)$ holds, there exists $i' > i$ such that $P(i')$ holds. This will ensure $P(|r|)$, which implies TINI.

For $1 \le i \le |r|$, define $\langle \sigma_i, u, j_i, m_i, z, \hat{\rho}_i, \lambda_i\rangle := \text{fst}(r_i)$.

Since we assumed $\text{fst}(r_1) \sim_{w,w'} \text{fst}(r'_1)$ and in loads the same code on both sides, $P(1)$ holds trivially by choosing $\mu_1 = \{1 \mapsto 1\}$.

Let now $1 \le i < |r|$, and assume $P(i)$. Let $i' = \mu(i)$. We proceed by case distinction on σ_i.

- $\sigma_i =$ `while` e_1: b_1;t.
  Let $[\![e_1, u, m_i, z]\!] =: \langle v, \overline{\ell}\rangle$ and $[\![e_1, u, m'_{i'}, z]\!] = \langle v', \overline{\ell}'\rangle$. By lemma A.1, $\langle v, \overline{\ell}\rangle =_{w,w'} \langle v', \overline{\ell}'\rangle$.
  Consider three cases:
  (1) If $w \notin \overline{\ell}$ and $w' \notin \overline{\ell}'$, then the truth values of v and v' are the same: if both v and v' are atomic, then $v = v'$; otherwise, both are dictionaries and their truth value is $\bot$. Thus, the next states (produced by rule 35 or 36) will be such that $\sigma_{i+1} = \sigma'_{i'+1}$. Moreover, the label stacks will still be indistinguishable, since adding the label sets of indistinguishable memory cells to indistinguishable input label stacks preserves indistinguishability. If $v_1 = \top$, memory is not modified. Otherwise, we get $m_{i+1} = \text{adopt}(A, \overline{\ell} \cup \text{all } j_i)$ and $m'_{i'+1} = \text{adopt}(A, \overline{\ell}' \cup \text{all } j'_{i'})$ where $A = \text{assigned}(b_1)$. As $\langle v, \overline{\ell}\rangle =_{w,w'} \langle v', \overline{\ell}'\rangle$, then $\langle \bot, \overline{\ell}\rangle =_{w,w'} \langle \bot, \overline{\ell}'\rangle$. Moreover, we have $\langle \bot, \text{all } j_i\rangle =_{w,w'} \langle \bot, \text{all } j'_{i'}\rangle$ since $j_i \equiv_{w,w'} j'_{i'}$. We get $\langle \bot, \overline{\ell} \cup \text{all } j_i\rangle =_{w,w'} \langle \bot, \overline{\ell}' \cup j'_{i'}\rangle$. Finally, we use lemma A.4 to obtain $\tilde{m}_{i+1} \sim_{w,w'} \tilde{m}'_{i'+1}$. The labeled trace is modified using sanitize. Since the arguments passed to sanitize are pairwise (w, w')-indistinguishable, the two resulting traces are (w, w')-indistinguishable as well (see remark above). Choosing $\mu_{i+1} = \mu_i \cup \{i + 1 \mapsto i' + 1\}$, we obtain $P(i+1)$.

(2) If $w \in \overline{\ell}$, $w' \in \overline{\ell}'$ and b_1 cannot return or call external functions, let $p > i$ (resp. $p' > i'$) be the index of the next Pop in r (resp. r'). Since $w \in \overline{\ell}$ (resp. $w' \in \overline{\ell}'$) is part of all label stacks during the iteration in r (resp. r') and all core memory locations that can be updated during the iteration are added these labels in both runs, (w, w')-indistinguishability of the memory is propagated by lemma A.3. Similarly, every table of the labeled trace (storing `Learn` events for some user u) which is extended with new rows during the iteration in r *or* r' ends up being labeled with *both* w in r and w' in r'. Namely, if the table is extended in r (resp. r'), the label w (resp. w') present in the label stack will be added to the table by fp; if the table is not extended in r but is extended in r', then w will be added to the labels of the table when some `then`, `else` or `while` branch executed in r' but not in r, and containing a return instruction or a call to an external function owned by u, will be skipped. This yields $r_{p+1} \sim_{w,w'} r'_{p'+1}$. Moreover, for all $i < q < p$, we have $r_q \sim_{w,w'} r'_{p'}$ since w labels j_q and w' labels $j'_{p'}$. We construct $\mu_{p+1} = \mu_i \cup \{i+1 \mapsto p', \ldots, p \mapsto p', p+1 \mapsto p'+1\}$, which yields $P(p+1)$.

(3) If $w \in \overline{\ell}$, $w' \in \overline{\ell}'$ and b_1 can return, let $p > i$ (resp. $p' > i'$) be the index of the state at which the current function in r (resp. r') returns (i.e., the state immediately preceding the Ret state). As $w \in \overline{\ell}$ and $w' \in \overline{\ell}'$, rules 35 and 36 guarantee that w (resp. w') will remain part of the input label stack until p (resp. p') in r (resp. r'). All memory locations that can be updated during the iteration are added label w (resp. w'), and we get $r_{p+1} \sim_{w,w'} r'_{p'+1}$ by repeatedly applying lemma A.3. Since rules 30–32 append all j_p (resp. all $j'_{p'}$ to the labels of the return value, return value o from r (o' from r') is labeled by w (resp. w'), yielding $o =_{w,w'} o'$. Hence, we can construct $\mu_{p+1} = \mu_i \cup \{i+1 \mapsto p', \ldots, p \mapsto p', p+1 \mapsto p'+1\}$ to obtain $P(p+1)$.

If $\sigma_i = \texttt{if}\ e_1\texttt{:}\ b_1\ \texttt{else:}\ b_2\texttt{;}t$, the proof is similar.

- $\sigma_i = k[e_1]\ldots[e_n]\texttt{=}e'\texttt{;}t$

  We first consider the case in which all memory locations accessed are already initialized (rule 43).

  By Lemma A.1, we get $\langle v, \overline{\ell}\rangle := [\![e', u, m_i, z]\!] =_{w,w'} [\![e', u, m_{i'}, z]\!] =: \langle v', \overline{\ell}'\rangle$ and for all $p \in \{1, \ldots, n\}$, $\langle v_p, \overline{\ell}_p\rangle := [\![e_p, u, m_i, z]\!] =_{w,w'} [\![e_p, u, m'_{i'}, z]\!] =: \langle v'_p, \overline{\ell}'_p\rangle$.

  If $w \notin \bigcup_{1\le p\le n} \overline{\ell}_p$, then $v_p = v'_p$ (if all memory locations exist, then v_p is atomic): the same memory location $i\,[v_1]\ldots[v_n]$ is modified in both r and r'. By Lemma A.3, we get $r_{k+1} \sim_{w,w'} r'_{k'+1}$.

  Now, assume $w \in \bigcup_{1\le p\le n} \overline{\ell}_p$. Let $p_0 = \min\{1 \le p \le n \mid w \in \overline{\ell}_p\}$. By $P(i)$ and Lemma A.1, we have $v_p = v'_p$ for all $1 \le p < p_0$. As above, in both function runs, only sublocations of $l^* = k[v_1]\ldots[v_{p_0-1}]$ are modified. But through adoptall, l^* adopts $\overline{\ell}_{p_0} \ni w$ in r and $\overline{\ell}'_{p_0} \ni w'$ in r'. Hence, we once again get $r_{i+1} \sim_{w,w'} r_{i'+1}$ by Lemma A.3.

  This proves that in this case, we have $P(i+1)$ by choosing $\mu_{i+1} = \mu_i \cup \{i+1 \mapsto i'+1\}$.

  If one or both locations accessed do not yet exist (rule 44), the two previous subcases can be easily adapted. In the first case, $P(i)$ implies that for all $1 \le p < n$, dom $m_i[k[v_1]\ldots[v_p]] =$ dom $m'_{i'}[k[v_1]\ldots[v_p]]$. The nonexistence must occur in both runs and at the same level, and the memory changes will thus be the same. In the second case, for some $1 \le p < n$ it might happen that w (resp. w') labels $m_i[k[v_1]\ldots[v_p]]$ in r (resp. $m'_{i'}[i[v_1]\ldots[v_p]]$ in r'), but this always happens *before* the first level at which the keys of the two memory locations differ, in which case all modified locations are sublocations of the first such index p_0, and the application of adoptall and Lemma A.3 yields the same conclusion.

  If $\sigma_i = \texttt{del}\ i[e_1]\ldots[e_n]\texttt{;}t$, the proof is similar.

- $\sigma_i = \texttt{return}\ e_1\texttt{;}t$

  We have $\sigma_{k+1} = \text{Ret}\ \langle v, \overline{\ell} \cup \text{all}\ j_i\rangle =: \text{Ret}\ o$ and $\sigma'_{k'+1} = \text{Ret}\ \langle v', \overline{\ell}' \cup \text{all}\ j'_{i'}\rangle =: \text{Ret}\ o'$ where $\langle v, \overline{\ell}\rangle = [\![e_1, u, m_i, z]\!]$ and $\langle v', \overline{\ell}'\rangle = [\![e_1, u, m'_{i'}, z]\!]$.

  By $P(i)$ and Lemma A.1, we have $\langle v, \overline{\ell}\rangle =_{w,w'} \langle v', \overline{\ell}'\rangle$. Moreover, $j_i \equiv_{w,w'} j'_{i'}$. This yields $o =_{w,w'} o'$. Simultaneously, memory is updated, adding the labels in all j_k (resp. all $j'_{k'}$) to all locations in $A = \text{assigned}(t)$. This is similar to the first case of `while` and can be similarly proved to preserve memory indistinguishability. Hence, choosing $\mu_{i+1} = \mu_i \cup \{i+1 \mapsto i+1\}$, we obtain $P(i+1)$.

- $\sigma_i = \text{Ret}\ \langle v, \overline{\ell}\rangle$, $\sigma'_{i'} = \text{Ret}\ \langle v', \overline{\ell}'\rangle$

  Since $i < |r|$, then $\lambda_i \neq \bot$, hence by $P(i)$ there exists $1 \le t < i$ such that $\lambda_i = \text{fst}(r_t)$, $\lambda_{i'} = \text{fst}(r_{\mu(t)})$ and $P(t)$ holds. Then the next instruction in both s_t and $s'_{\mu(t)}$ is some $\iota\texttt{=}f(e_1,\ldots,e_n)\texttt{;}\sigma$. By $P(i)$, we also have $\langle v, \overline{\ell}\rangle =_{w,w'} \langle v', \overline{\ell}'\rangle$, hence the memory updates in r_i and $r'_{i'}$ preserve indistinguishability by Lemma A.3. Choosing $\mu_{i+1} = \mu_i \cup \{i+1 \mapsto i'+1\}$, we obtain $P(i+1)$.

- $\sigma_i = \text{Pop}\texttt{;}t$

  Clearly, setting $\mu_{i+1} = \mu_i \cup \{i+1 \mapsto i'+1\}$ yields $P(i+1)$.

- $\sigma_i = \iota\texttt{=}f(e_1,\ldots,e_n)\texttt{;}t$

  In this case, the next transition in both r and r' ensures $\lambda_{i+1} = \text{fst}(\sigma_i)$ and $\lambda'_{i'+1} = \text{fst}(\sigma'_{i'})$. As before, Lemmas A.1 and A.3 guarantee that indistinguishability is preserved when local arguments are set to values obtained by evaluating the same expression against m_i and $m'_{i'}$, respectively, which are (w, w')-indistinguishable by virtue of $P(i)$. We construct $\mu_{i+1} = \mu_i \cup \{i+1 \mapsto i'+1\}$ to obtain $P(i+1)$.

- $\sigma_i = \iota\texttt{=external\_call}(r,e_1)\texttt{;}t$

  This case is similar to the case of standard assignments, as external function calls behave like regular (n-ary) operators.

□

A.2 Correctness of enforcement

In this subsection, we prove correctness of Dmol's semantics by showing that the following theorem holds.

Theorem 5.7. *If Φ is enforceable and* `Learn`*-decreasing, Dmol's semantics enforce $\tilde{P}$.*

Each system state contains a trace which can be retrieved via a function trace : $\mathbb{S} \to \mathbb{T}$. The relation $\rightsquigarrow$ only extends traces and

output sequences, i.e., for all s, s', o, o' with $s, o \rightsquigarrow s', o'$, we have $\mathsf{trace}(s) \preceq \mathsf{trace}(s')$ and $o \preceq o'$. Similarly, function in can only extend the trace.

For all system property P and trace property T, we say that T *captures* P iff $\mathsf{trace} \circ \mathsf{last\_state} \circ \mathsf{run}(q) \in T \implies \mathsf{run}(q) \in P$ for all $q \in \mathsf{Seq}(\mathbb{Q})$. Informally, T captures P iff the trace property T is stronger than the system property P: whenever T holds on the trace, P holds on the entire system.

An immediate consequence of the above definitions is the following lemma:

Lemma A.5. *If we have:*

(1) A trace property T capturing P,
(2) A monitor $M : \mathbb{T} \to \mathbb{B}$ with $M^{-1}(\top) \subseteq T$ and
(3) $M(\mathsf{trace} \circ \mathsf{last\_state} \circ \mathsf{run}(q)) = \top$ for every $q \in \mathsf{Seq}(\mathbb{Q})$,

then the state machine enforces P.

Proof. Let T, P and M such that 1)–3) hold. Let $q \in \mathsf{Seq}(\mathbb{Q})$. By 3), we have $M(\mathsf{trace} \circ \mathsf{last\_state} \circ \mathsf{run}(q)) = \top$. Hence, by 2), $\mathsf{trace} \circ \mathsf{last\_state} \circ \mathsf{run}(q) \in T$. By 1), this implies that $\mathsf{run}(q) \in P$. □

Using this theorem, one can show that the machine satisfies some system property P in three steps: (i) find a trace property T such that T is stronger than P (ii) exhibit a sound monitor for T (iii) show that this monitor never finds any violation when inspecting the state machine's trace.

In our case, the trace property we consider is

$$\tilde{T} = \left\{\rho \in (\mathbb{N} \times \mathbb{DB})^* \mid \forall v, i.v, i \models_\rho \Phi\right\}.$$

We now use Lemma A.5 to that if Φ is enforceable and `Learn`-decreasing, then Dmol's semantics enforce $\tilde{P}$.

Lemma A.6. *If Φ is* `Learn`*-decreasing, $\tilde{T}$ captures $\tilde{P}$.*

Proof. Assume that Φ is `Learn`-decreasing.

Since sMFOTL semantics are used to define both $\tilde{T}$ and $\tilde{P}$, showing that the same events are logged by trace and atrace would be sufficient to prove the theorem.

For `Input`, the events logged are clearly the same.

For `Learn` events, the fact that Φ is `Learn`-decreasing actually allows us to prove a weaker condition, namely that *at least* those events logged in $\tilde{P}$ are logged in $\tilde{T}$.

Now, let $q \in \mathsf{Seq}(\mathbb{Q}), X := \mathsf{run}(q)$, and consider the jth sequence of X, denoted by $X_j = ((s_1, o_1), \ldots, (s_m, o_m))$. Assume that an event $\pi = \mathtt{Learn}(u, \omega(\ell_x), \ell_x)$ has been logged in $\mathsf{atrace} \circ \mathsf{run}(q)$ at the timestamp of q_j. Then, by definition of atrace, we obtain an alternative query sequence q' and a position ab such that:

(i) q' is equal to q up to their respective inputs i_{ab} and i'_{ab},
(ii) $\mathsf{filter}(o_m, u)$ could not have been observed in $\mathsf{run}(q')_j$.

Let $X' := \mathsf{run}(q')$ and consider $X'_j = ((s'_1, o'_1), \ldots, (s'_{m'}, o'_{m'}))$. Let μ as constructed in Lemma 5.4. Then condition (ii) above implies that for all $1 \le d \le m'$, $\mathsf{filter}(o_m, u) \not\preceq \mathsf{filter}(o'_d, u)$. On the other hand, since $o_1 = \varepsilon$, we have $\mathsf{filter}(o_1, u) \preceq \mathsf{filter}(o'_{\mu(1)}, u)$. Hence, there exists $1 \le c_0 < m$, $1 \le d_0 \le m'$ such that $\mathsf{filter}(o_{c_0}, u) \preceq \mathsf{filter}(o'_{d_0}, u)$, but for all $1 \le d \le m'$, $\mathsf{filter}(o_{c_0+1}, u) \not\preceq \mathsf{filter}(o'_d, u)$. Select the lexicographically smallest such pair (c_0, d_0).

Clearly, the instruction executed on the transition $s_{c_0}, o_{c_0} \rightsquigarrow s_{c_0+1}, o_{c_0+1}$ performs at least one output to u. Hence, the instruction pointer in s_{c_0} points to a `return` or `external_call` instruction, and the user who made the current query is u. Let $d_1 = \mu(c_0)$. There are three cases:

(1) If $d_0 \le d_1 < m'$, we have $o_{c_0} \preceq o'_{d_0} \prec o'_{d_1}$, but $o_{c_0+1} \not\preceq o'_{d_1+1}$. Consider two subcases:
   (a) If $o_{c_0} = o'_{d_1}$, then either both s' and s_1 execute the same `return` or `external_call` instruction with different values, or they execute different instructions. In the first case, by definition of μ, ℓ_x labels the return value (resp. the arguments passed to the external function), and π is logged to the trace; in the latter, ℓ_x labels the instruction pointer also by definition of μ, and π is also logged to the trace given Alg. 1.
   (b) If $o_{c_0} \prec o'_{d_1}$, then we can find $c' < c_0$ such that $s'_{\mu(c')}$ produces an output while $s_{c'}$ produces none. This is only possible if at least one `then`, `else` or `while` branch containing a `return` or `external_call` instruction is skipped in $s_{c''}$ for some $c'' \le c'$. Then ℓ_x labels the instruction point from $s_{c''}$ onwards until the end of the current entry point run (ℓ_x is pushed at the bottom of the label stack by Eq. 33–36 when a branch containing `return` or `external_call` is skipped). Consequently, when Alg. 1 is run at $c > c''$, ℓ_x labels the instruction pointer, and π is logged to the trace.
(2) If $d_1 < \min(d_0, m')$, then by our choice of (c_0, d_0), $o_{c_0} \not\preceq o'_{d_1}$, but since $o_{c_0} \preceq o'_{d_0}$, we get $o_{c_0} \succ o'_{d_1}$. Hence, we can find $c' < c_0$ such that $s_{c'}$ produces an output while $s'_{\mu(c')}$ produces no output or a different one. As in the first case, π is logged to the trace.
(3) If $d_1 = m'$, since no output takes place in a final state, ℓ_x labels the instruction pointer in s', and we conclude as above.

□

Lemma A.7 (Correct monitoring). $\tilde{T} = \mathcal{M}_\Phi^{-1}(\top)$.

Proof. This corresponds to the correctness of MonPoly's monitoring algorithm [8]. □

Lemma A.8. *If Φ is enforceable, then for every $q \in \mathsf{Seq}(\mathbb{Q})$, we have $\mathcal{M}_\Phi(\mathsf{trace} \circ \mathsf{last\_state} \circ \mathsf{run}(q)) = \top$.*

Proof. First, recall that our monitoring algorithm $\tilde{\mathcal{M}}_\Phi$ is a labeled version of MonPoly's $\mathcal{M}_\Phi$, returning the same verdicts modulo labeling.

Assume that Φ is enforceable. Then the empty trace does not violate Φ, and a violation can only occur after logging a controllable event, i.e., `Learn`. As `Learn` events are always logged as in Alg. 1, it suffices to inspect this procedure. But Alg. 1 only alters the trace if no violation is detected in the new trace. Hence, for all $q \in \mathsf{Seq}(\mathbb{Q})$, $\tilde{\mathcal{M}}_\Phi(\mathsf{trace} \circ \mathsf{last\_state} \circ \mathsf{run}(q)) = \langle \top, \overline{\ell} \rangle$ for some $\overline{\ell} \subseteq \mathcal{L}$, and hence $\mathcal{M}_\Phi(\mathsf{trace} \circ \mathsf{last\_state} \circ \mathsf{run}(q)) = \top$ by the above remark. □

There follows:

Theorem 5.7. *If Φ is enforceable and* `Learn`*-decreasing, Dmol's semantics enforce $\tilde{P}$.*

Proof. By Lemmas A.6–A.8, using Lemma A.5. □

B EXAMPLE OF COMPILED DMOL CODE

```
from apps import test
from databank.imports import *
__stack__ = None

def hello(uid):
    global __stack__
    if ('name' not in locals()):
        name = Cell(None)
    if ('msg' not in locals()):
        msg = Cell(None)
    msg = Cell('Hi')
    msg.add_inputs(__stack__.all())
    ___0 = test.sql(
        Cell('SELECT␣name␣FROM␣users␣WHERE␣uid␣=␣?0'),
        Cell([uid]))
    name = ___0[Cell(0)][Cell(0)]
    name.add_inputs(__stack__.all())
    ___1 = Cell((name != Cell('')))
    if ___1:
        __stack__.push()
        __stack__.add(___1.inputs)
        msg = ((msg + Cell(',␣')) + name)
        msg.add_inputs(__stack__.all())
        __stack__.pop()
    else:
        logsanitize(
          test.id_,
          Cell(___1, adopt=__stack__.all()).inputs,
          [], [], auto=True)
        msg.add_inputs(__stack__.all())
        msg.add_inputs(___1.inputs)
    __r__ = Cell(msg, adopt=__stack__.all())
    __s__ = __stack__.all()
    return (__r__, __s__, [], [])

@test.route('hello/<int:uid>')
def _hello(uid):
    global __stack__
    __stack__ = Stack()
    uid = test.register('hello', 'uid', uid)
    (__r__, __s__, __a__, __u__) = hello(uid)
    logreturn(test.id_, test.me(),
        __r__, __s__, __a__, __u__, auto=True)
    return __r__
```