A Platform for Expressive and Secure Data Sharing with Untrusted Third Parties

Eric Griffis Jeffrey A. Vaughan Todd Millstein
University of California, Los Angeles

Novemember 2011

Abstract—**Today, third-party applications provide a variety of rich services to smartphone users. There are compelling reasons to share personal data with third parties, such as social networking applications, but the benefits of sharing data must be balanced against the corresponding risks to personal privacy. Prior work has proposed *personal data vaults* to separate the capturing and sharing of data. We argue that to express realistic access control policies, personal data vaults must support the ability to flexibly transform data before release (e.g., converting location data to the zip code level). We propose a simple, practical personal data vault design that supports full-fledged computation by both trusted and untrusted entities. Our design includes a small scripting language called TRANSMUTE which pipes data streams through sandboxed *filters* written in the Lua programming language. A straightforward analysis of TRANSMUTE scripts provides a strong guarantee of noninterference, and our approach admits a flexible mechanism for audit that provides finer-grained information about data leakage. We formalize our approach and prove a noninterference theorem. We also describe our implementation and evaluate its expressiveness and performance through several case studies.**

I. INTRODUCTION

The advent of cloud-enabled mobile devices has made possible an incredible range of new services. Third-party smartphone applications today can do everything from locating your nearby friends to tracking your environmental impact to monitoring your health status, and the pace of innovation is rapid. While the benefits of these applications are very compelling, these services also pose significant privacy risks to users by providing untrusted third parties with raw data arising from the wealth of repositories and sensors on a modern smartphone, including information about location, calendar, contacts, audio, and video.

Recently researchers have proposed to address this problem in part by decoupling the capture and archival of raw data streams from the sharing of that information with third parties. The basic idea is for user data to be stored in a single secure repository under the user's control. Separately, the user, or a trusted party acting on behalf of the user, can leverage this single point of control to mediate access to data by third parties. Examples of approaches that employ this basic architecture are personal data vaults (PDV) [11, 16], personal cloud butlers [14], and virtual individual servers [4, 15]. In this paper we refer generically to a user's secure data repository as a PDV.

The above approaches each expose a special-purpose access-control mechanism to users. However, we believe that many real-world access-control policies will require the ability to perform general computation within the PDV. For instance, Mun et al. [11] describe the following policy for an application called *Ambulation*, which employs a user's location to track mobility patterns over time:

> "If the application named Ambulation queries, share the exact location when the user's in Westwood (within 1.5 km of the GPS coordinates of (34.073307, -118.448894)), otherwise, share location at a zip code level."

Personal data vaults support this policy through a set of built-in transformations, for example the ability to abstract raw GPS coordinates to the zip code level. However, any fixed set of transformations is unlikely to be expressive enough to enforce fine-grained privacy requirements for the increasing variety of mobile applications, many of which will provide capabilities that are unforeseeable today.

Furthermore, it is desirable to allow *untrusted third parties* to perform computations within a user's PDV. In general, only the application provider has both the technical means and the economic incentive to define and enforce privacy policies desired by users. In addition, some policies or computations may be implementable only using application-specific algorithms, which are unlikely to be found in a general-purpose privacy toolkit. For instance, consider the personal environmental impact record (PEIR) application [10], which employs a user's location data to determine things like the amount of time spent driving per day. PEIR currently requires users to release their raw location data, which is then analyzed on the server side. By enabling the PEIR developers to move some or all of the server-side processing to the PDV itself, it becomes possible for users to gain the benefits of PEIR while ensuring that raw location data never leaves the PDV.

Of course third-party application providers also have incentives to cheat by accessing more data than promised. In the following we will see how to manage access to private data via a combination of proactive dataflow analysis, online sandboxing, and ex post facto audit. Sandboxing and analysis can prevent information leakage, and audit increases the likelihood of detection when undesired information disclosure occurs. This increased risk of detection may change third party incentives such that they are unwilling to violate privacy

policies, even when it is technically feasible.

This paper proposes a simple, practical approach to augmenting PDVs with support for full-fledged computation, by both trusted and untrusted entities, while retaining strong security guarantees as well as transparency for non-technical users. We provide a small scripting language called TRANSMUTE, which pipes streams of data through various *filters* that are written in a general-purpose programming language and outputs new streams of data. Simple access-control lists (ACLs) grant read/write privileges to these data streams. For example, to implement the policy for Ambulation described above, one would create filters to calculate the distance of a given location to (34.073307, -118.448894), to transform a location to the zip code level, etc. A TRANSMUTE script would employ these filters to produce a new stream that contains the abstracted location data, which would be made readable by the Ambulation application.

We provide an execution environment that sandboxes filters. For example, access to the network and the file system is forbidden. As usual, sandboxing prevents malicious filters from mounting a host of possible security attacks and from leaking information via (many) covert channels. In addition, sandboxing provides a basic level of safety that we build upon to allow users to easily understand their access-control policies:

Manifest Data Flow Sandboxing ensures that a filter can only read the data streams that are passed as input to it (and transitively any streams that those streams depend upon), and a filter can only write to streams to which its outputs are piped (possibly transitively). Therefore, it is possible to get strong guarantees about an access-control policy by a simple reachability analysis of the associated TRANSMUTE script, while treating the script's filters as black boxes. For example, a script for our example Ambulation policy would manifestly require access to a user's location stream (since that stream is an explicit input to at least one filter) but would also manifestly never access a user's calendar information.

Audit Via Replay Because filters, and therefore TRANSMUTE scripts, are free of external dependencies, they can easily be replayed, providing a simple and flexible approach to audit. Users can replay TRANSMUTE scripts on both current and old snapshots of the PDV data streams and inspect the outputs. They can augment these scripts with filters that check desired properties of output streams, thereby automating the auditing process. They can also perform *counterfactual* auditing, whereby either data, policy, or both, are modified before replay. This allows users to easily ask questions such as "What data is provided to Ambulation when I am in location L?" and "How does the output change if I modify my Ambulation policy from P_1 to P_2?"

We have built a PDV implementation based on the above ideas. TRANSMUTE scripts running in the PDV employ filters written in the Lua programming language and executed in a sandboxed version of the Lua interpreter.[1] The PDV includes a simple API for reading and writing streams. It also includes tools to compute and visualize a policy's dependencies and to performing auditing tasks. We have implemented several example applications in our PDV and use them to evaluate the expressiveness, security, understandability, and performance of our approach. We have also formalized a core subset of TRANSMUTE in order to precisely specify its behavior. A simple security type system captures the data streams that a particular policy depends upon, and we ensure the correctness of this type system by proving a noninterference theorem for well-typed policies.

Section II gives an informal overview of our PDV design, and Section III provides further examples of its usage. Section IV formalizes TRANSMUTE and proves a noninterference theorem. Section V describes the implementation of our PDV and presents the results of several case studies using our implementation. Finally, Section VI compares our approach with related work and Section VII concludes.

[1] http://www.lua.org/

II. OVERVIEW

This section overviews our approach to supporting expressive policies for sharing data with untrusted third parties, using the Ambulation policy from the previous section as a running example. We first describe the format of data streams and the associated ACLs that restrict read/write access to these streams. Then we present the TRANSMUTE language by example. Finally we describe our support for validating TRANSMUTE scripts via dependency analysis and audit.

A. Data Streams

TRANSMUTE scripts manipulate *streams* of data. A stream is a sequence of values, each optionally annotated with one or more *tags*. Examples of values include integers, binary sequences representing photographs, and GPS coordinates. Tags are represented as strings. This simple approach allows us to represent a wide variety of kinds of data, with tags used to associate application-specific meaning to that data. For instance, the following stream contains GPS coordinates, some of which are tagged as being in the Westwood neighborhood.

Data	Tags
$(34.063, -118.436)$	
$(34.064, -118.445)$	westwood
$(34.065, -118.445)$	westwood

We distinguish between two kinds of streams. *Concrete streams* contain "raw" data that is uploaded to a user's PDV (either by the user or by another authorized entity), for example the GPS coordinates resulting from a location trace provided by the user's smartphone. *View streams* are computed from other (both concrete and view) streams by TRANSMUTE scripts, for example the transformed location information that will be provided to the Ambulation application.

Data streams have a simple access control model. The PDV maintains an access control matrix describing which principals are allowed to read from a (concrete or view) stream and which are allowed to write to a stream. For example the

```
_in_westwood // _not_in_westwood =
    location
    → tag−circle westwood 34.073307 –118.448894 1.5km
    → split −tags westwood;

_coarsened =
    zips // _not_in_westwood
    → coarsen−location;

for_Ambulation =
    _in_westwood // _coarsened
    → merge;
```

Fig. 1. TRANSMUTE implementation of the example Ambulation policy.

following policy allows the Ambulation service to read the for_Ambulation view stream but not the user's precise location.

	location	for_Ambulation
Ambulation	{}	{**read**}

A PDV has a unique *owner*, who is implicitly assumed to have permission set {**read**, **write**} for all streams and who is the only principal allowed to modify the PDV's access control matrix. We assume there is some mechanism for authenticating principals, but that is orthogonal to our contributions in this paper.

The PDV provides a simple API for reading and writing streams as well as for performing audit operations and PDV maintenance. For instance, the pdv−read command sends a stream to standard out, while the command pdv−write replaces the contents of a concrete stream with standard input. If pdv−read accesses a view stream, it is automatically generated and cached. Both commands verify that the requested operation is permitted by a user's access control lists. Basic commands may be called by a network-facing service, such as a web server, to provide remote PDV access.

B. TRANSMUTE

We illustrate our scripting language TRANSMUTE through the Ambulation policy described earlier. The complete TRANSMUTE implementation of this policy is shown in Figure 1. The script contains three *statements*, each of which uses one or more filters to create new streams from existing ones. For example, the first statement in the figure uses the concrete location stream, which contains raw GPS coordinates, and produces the two streams _in_westwood and _not_in_westwood. The // syntax is used as a separator between streams in the case that multiple streams are input to or output from a filter or command.

Each filter is piped as input (via the → operator) one or more data streams, optionally takes additional command-line arguments, and outputs one or more data streams. In our example, the location stream is piped to the tag−circle filter, whose additional arguments direct it to produce a stream identical to location but with any coordinates within 1.5km of (34.073307, -118.448894) tagged with westwood. The split −tags filter splits the resulting stream into two streams, the first containing all data values tagged with westwood (_in_westwood) and the second containing all other data values (_not_in_westwood). The coarsen−location filter converts each data value in the latter stream to the nearest value from the stream zips, which contains one coordinate per zip code. Finally, the resulting stream is combined with _in_westwood into a single stream via the merge filter and given the name for_Ambulation.

By convention streams whose names begin with an underscore (for example _in_westwood and _coarsened) are considered *ephemeral*. Accordingly, no principals have read or write access to these streams. The for_Ambulation stream, in contrast, is the visible end product of the above script, and by the access control matrix shown earlier the Ambulation application is given read access to it.

All view streams, including both outputs like for_Ambulation and ephemeral streams like _in_westwood may be cached for fast access or deleted by the system at any time to conserve space.

The TRANSMUTE approach to sharing data with untrusted third parties is very expressive, since it incorporates general computation within filters. In our implementation, filters are written as arbitrary (sandboxed) Lua code. At the same time, policies remain accessible even to non-technical users. As our above example illustrates, TRANSMUTE encourages the creation of policies by composing several small filters, each with a relatively simple behavior. This style makes it easy for users to understand the behavior of a TRANSMUTE script as a whole as well as to create and evolve such scripts.

We envision an ecosystem in which filters are created by many sources and widely shared (possibly at a cost). Such sources could range from application providers to advocacy groups and other trusted parties (e.g., the Electronic Freedom Foundation) to individual users. Many filters are quite general (e.g., split −tags and merge) and can easily be reused across many scripts, thereby decreasing the cost of creating, maintaining, and validating scripts. Furthermore, multiple sources may create filters that share the same intended functional specification, and users can select the implementations to employ in their PDV based on their level of trust in that source as well as other considerations.

C. Manifest Data Flow

Filters are side-effect-free programs. That is, they must behave like mathematical functions—repeatedly applying a filter to a fixed sequence of input streams and other arguments always yields the same output streams and does not observably alter system state. This property is guaranteed by running filters in a sandbox, which prevents access to the network, the file system, environment variables, sources of non-determinism, etc. (Our sandbox implementation does not handle possible side effects due to resource exhaustion or external timing channels.)

Sandboxing has the effect of making all of a filter's dependencies explicit in the TRANSMUTE scripts that invoke it, thereby providing a simple and strong guarantee to users

about security and privacy. For example, because the only stream input to tag−circle is location in our example script for Ambulation, a user is guaranteed that this filter cannot read any other data streams on the PDV, such as the user's calendar entries or photos. Similarly, the tag−circle filter outputs a new stream, but it is guaranteed to have no other effects on the state of the PDV or external effects via the network.

These guarantees on individual filters extend naturally to entire TRANSMUTE scripts. For example, since the only data streams mentioned in the computation of the for_Ambulation are location and zips, the user can be sure that no other data can flow to the Ambulation application. To aid users in attaining these guarantees, our PDV implementation includes a command that produces the dataflow graph of a TRANSMUTE script. Figure 2 shows the graph automatically produced for the Ambulation script in Figure 1. The graph elides the names of ephemeral streams. The graph also incorporates access-control information to indicate which principals may read the resulting for_Ambulation stream, in this case just the Ambulation application.

The requirement of side-effect-freeness can affect the way in which filters are implemented. For example, the coarsen−location stream is forced to explicitly take as an argument a stream containing a GPS coordinate per zip code, rather than for example accessing such information over the internet. Such a stream could be written to the user's PDV by the provider of coarsen−location or by any other principal authorized to do so. We believe the benefits of enforcing side-effect-freeness far outweigh these kinds of inconveniences, which have been relatively minor for the example applications that we have built.

D. Audit

Audit is a natural approach to giving users fine-grained information about the behavior of TRANSMUTE scripts and the associated filters. For example, a user of the Ambulation script would like to gain confidence that it is properly transforming certain GPS coordinates to the zip code level rather than providing Ambulation with the raw location data. Audit makes this easy by allowing users to examine both the current for_Ambulation stream as well as the data it has contained in the past.

Because filters are side-effect-free, we can perform audit simply via *replay*. For example, to audit the information that was released to the Ambulation application last Thursday, it suffices to retain last Thursday's location and zips concrete streams; the corresponding for_Ambulation stream can then be regenerated simply by re-running the above TRANSMUTE script on these inputs. To facilitate audit, our PDV implementation allows the owner to periodically *snapshot* the current state, maintaining a copy of all concrete streams, filters, and TRANSMUTE scripts, along with a timestamp that can be used later to identify the snapshots of interest for audit.

To begin an audit procedure, the user copies a specified snapshot into the audit environment with the pdv−audit command. The user may then manipulate or inspect the audit copy of the snapshot data arbitrarily and run standard PDV commands on it. For example, the user can run the TRANSMUTE script for Ambulation and manually inspect the for_Ambulation results for compliance to the intended policy. Compliance checking can also be automated by updating the TRANSMUTE script with additional statements that check desired properties of the resulting for_Ambulation stream, for example that GPS coordinates outside of Westwood are properly abstracted to the zip code level based on the data in the zips stream. Finally, *counterfactual* audit can be performed by modifying concrete streams and/or the TRANSMUTE script to ask "what if" questions, for example to learn what information is revealed to Ambulation when the zips stream is empty.

III. EXAMPLES

The Ambulation example from the previous section illustrates how TRANSMUTE scripts that employ general computation within filters supports the creation of fine-grained privacy policies. In this section we complement that example with two additional examples inspired by real applications, which highlight additional benefits of our approach.

A. Wakefulness

One exciting application domain for mobile services is the collection of medical data about a user for sharing with researchers or clinicians. In this example we consider an application that we call *Wakefulness*, which is inspired by an existing Android application that monitors a user's sleep quality [11]. The application has a user fill out a small survey each morning about her sleep quality from the previous night. The survey includes specific details such as the time the user went to bed and woke up, as well as a qualitative rating of the night's sleep ("excellent," "very good," "good," "fair," "poor").

We consider a Wakefulness user who wishes to enforce the following privacy policy:

> Share the user's sleep summary with the Wakefulness application. Wakefulness may also request more specific details about a particular date. In that case, release the information, but notify the user if details are requested for more than one date in any given month.

We express this example using the following TRANSMUTE script:

```
sleep_summary = survey
    → project date summary;

requested_details = clinician_requests // survey
    → join date;

anomalous_requests = clinician_requests
    → group−by date "^(%d+−%d+)−"
    → count 2+;
```

The script produces three streams as output, the first two of which are made readable to the Wakefulness application. The first statement provides the user's sleep summary to the clinician. The concrete stream survey contains the user's

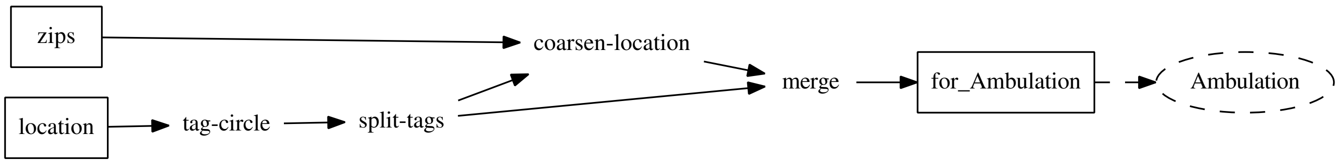

Fig. 2. Dataflow graph produced for the TRANSMUTE script in Figure 1

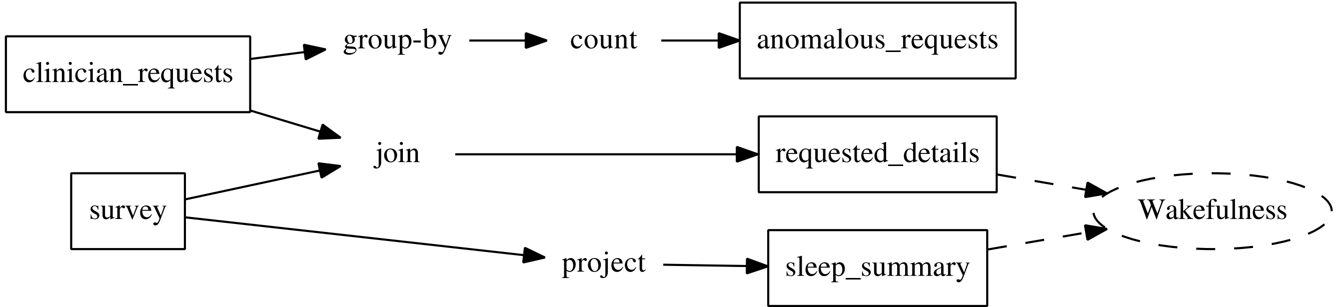

Fig. 3. Dataflow graph produced for the Wakefulness TRANSMUTE script

sleep survey data, which is only readable and writable by the user. Analogous with a traditional query language like SQL, the project filter is used to pull out specific "columns" of information from that stream, in this case the user's summary for each date.

The second statement provides the clinician with more specific data. To support this, we leverage the ability for third parties to write their own concrete streams in a user's PDV. In this case, clinician_requests contains a list of dates for which the clinician requests more detail. A filter that mimics a standard relational join is used to produce the data for those dates.

The last statement in the above script notifies the user about frequent requests for specific sleep data. This is done by using a filter to group sleep data by year and month (as expressed in the specified regular expression over the date string) and another filter to find groups with at least two entries. The results are written to a new stream, which the user can periodically examine. If a user finds clinician requests intrusive, she may prevent further data access by reconfiguring the policy.

B. Data-driven Advertising

Many mobile applications employ a third-party advertisements service to generate revenue. For example, free applications for Android often use Google's AdMob[2] service to display ads. To provided targeted ads, such services typically require some personal information about users, for example their location. This poses a dilemma for users who want to obtain the benefits of free applications but are not comfortable releasing private information.

[2]www.admob.com

In this example we illustrate how our approach naturally supports data-driven advertising while preserving user privacy. The key idea is for the advertisement service to compute targeted ads for a user directly on the user's PDV. Therefore, the ads service can determine the best ads to display and the user's private information never leaves his PDV.

The workflow for this example is very simple and is shown in the following TRANSMUTE script:

```
for_Advertiser = location // from_Advertiser
    → nearby−advertisers;
```

The nearby−advertisers filter is provided by the (untrusted) ads library. The filter takes as input the user's concrete location stream, along with a concrete stream written by the ads service itself which contains a list of possible ads to display. The filter produces the list of targeted ads in a stream for_Advertiser which is readable by the ads service. In this way, the ads service can determine which ads to send to the user, and the user's location information remains private. This approach relies critically on the fact that filters are sandboxed, which guarantees that the untrusted nearby−advertisers filter cannot have any side effects.

The script shown above can easily be modified to obtain other desired policies. For example, the user could employ the coarsen−location filter shown earlier to abstract location information to the zip code or city level before passing it to the nearby−advertisers filter. As another example, the user could (manually or automatically) modify the output of nearby−advertisers to remove ads for businesses from particular categories before making the result visible to the ads service.

This general approach can easily be used for other application domains where third parties are partially untrusted. For instance, consider an application for preparation of income

taxes. The user may safely enter her financial data into the PDV and allow untrusted-but-sandboxed filters to produce the necessary tax forms. Neither the financial data nor the resulting tax forms need be made readable to the tax preparation service.

IV. Formal model of the PDV

This section presents a simplified formal model, corePDV, capturing the essence of the PDV architecture, and demonstrates that corePDV programs satisfy noninterference, type soundness, and other interesting properties. Some details and all proofs are deferred to the appendix.

A. Data model and PDV structure

The corePDV data model is defined in terms of streams of tagged data values. As shown in Figure 4 $\mathbf{strm}$ is the set of all streams. Maps from stream names to stream data are established by environments, $E : \mathbf{strmName} \rightharpoonup \mathbf{strm}$. Here the symbol $\rightharpoonup$ indicates a partial function space. Unlike the full PDV implementation, where streams may be anonymous and identified by position alone, all corePDV streams are named.

The main computational components of corePDVs are filters that map zero or more streams into zero or more streams. Filters can be given parameters as key-value pairs, and a map from keys to values is called a *configuration* I. Filters are named by a signature, Σ, such that when

$$\Sigma(T, I) = (n, m, f)$$

function $f : \mathbf{strm}^n \to \mathbf{strm}^m$ is the implementation of the filter named T. Different configurations I may yield filter implementations with different arities.

A PDV $P \in \mathbf{P}$ is modeled as a pair of a *snapshot history* Q and log L. The snapshot history Q is a map that for a given time identifies that *snapshot* q was active at that time. That is, Q allows the recovery of old snapshot q.

A snapshot q is a tuple (Σ, E, p, ϕ). Σ is a filter signature. Environment E represents the current state of all concrete streams. Script p and access control policy ϕ describe how view streams are computed from concrete streams and which principals may access or update streams. A coreTransmute script p consists of two parts. First, a statement **concrete** $\overrightarrow{x}$ enumerates the names of all concrete streams. Second a list of declarations defines new streams in terms of filters and previously declared streams. An access control policy ϕ is simply a function that implements an access control matrix and defines when a principal should be permitted to read or modify a stream. At the top level principals interact with the PDV by issuing **get** and **put** requests (see Appendix A) for their semantics.

Appendix A defines the semantics of script evaluation as a function $[\![\mathrm{p}]\!]$. The function executes script p in the context of a given filter signature Σ and environment E, returning either a new environment containing the output streams or the result $\perp$ to indicate a run-time error (e.g., applying arguments of the wrong arity to a filter). The appendix also formalizes the notion of a *query*, which executes a script on behalf of a given principal, possibly returning **err** to indicate an access-control failure.

B. Formal properties

This section states formal properties of script evaluation.

Type soundness A simple static type system (Definition 1 below) can prevent runtime failures (results of $\perp$) when evaluating PDV scripts. Type soundness (Theorem 2 below) is independently important to ensuring PDV reliability and also supports the proof of noninterference described later.

Definition 1 ($\Sigma \vdash p \triangleright \overrightarrow{z}$): A coreTransmute script p is well-formed with respect to filter signature Σ and supports stream names $\overrightarrow{z}$, when $\Sigma \vdash p \triangleright \overrightarrow{z}$ can be derived in the following system:

$$\frac{}{\Sigma \vdash \textbf{concrete}\ \overrightarrow{z} \triangleright \overrightarrow{z}}\ \textsc{wf-Conc}$$

$$\frac{\Sigma \vdash p \triangleright \overrightarrow{z} \qquad \Sigma(T, I) = (|\overrightarrow{x}|, |\overrightarrow{y}|, _) \qquad \overrightarrow{x} \subseteq \overrightarrow{z} \qquad \overrightarrow{y} \cap \overrightarrow{z} = \emptyset}{\Sigma \vdash p; \overrightarrow{y} = T\ I\ \overrightarrow{x} \triangleright \overrightarrow{y} \cup \overrightarrow{z}}\ \textsc{wf-Let}$$

Theorem 2 (Type soundness): Suppose $\Sigma \vdash p \triangleright \overrightarrow{z}$ and $dom(E)$ is the set of concrete streams in p. Then $[\![p]\!]_\Sigma(E) = E'$ where $\mathrm{dom}(E') = \overrightarrow{z}$.

Proof sketch: By induction on the $\cdot \vdash \cdot \triangleright \cdot$ derivation.

Functional query evaluation When filters are implemented correctly, i.e. are total functions, all corePDV queries terminate and are deterministic.

Property 3 (Functionality): Partially-applied evaluation function $query(q, \cdot, \cdot) \equiv \lambda(x, a).query(q, x, a)$. is, for any q, a (total, single-valued) function.

This property says that a set of corePDV snapshots assigns a unique meaning to every query, depending only on the time about which the request asks and the identity of the requester; it is this property which justifies our interest in time travel audit. Additionally, it shows that complete time travel audit is compatible with garbage collecting old view streams and appears necessary to interpret meaningfully the results of counterfactual audit operations.

Noninterference This section demonstrates that a simple static analysis is able to provide a sound approximation of the information flows within a PDV.

The $p \blacktriangleright \overrightarrow{x} \rightsquigarrow y$ relation specifies the static analysis. This relation is intended to hold when script p only uses streams named in $\overrightarrow{x}$ when computing the stream named y. Definitions 4 and 5 establish the $\cdot \blacktriangleright \cdot \rightsquigarrow \cdot$ relation.

Definition 4 ($d \blacktriangleright \overrightarrow{x} \rightsquigarrow y$ and $d \blacktriangleright \not\rightsquigarrow \overrightarrow{y}$): The *influence* and *neglect* relations, defined as follows, give a static approximation of information flow for a particular filter.

$$\frac{y \in \overrightarrow{y}}{\overrightarrow{y} = T\ I\ \overrightarrow{x} \blacktriangleright \overrightarrow{x} \rightsquigarrow y} \qquad \frac{z \notin \overrightarrow{y}}{\overrightarrow{y} = T\ I\ \overrightarrow{x} \blacktriangleright \not\rightsquigarrow z}$$

Definition 5 ($p \blacktriangleright \overrightarrow{x} \rightsquigarrow y$): The influence relation is lifted

Set of tags	$\mathbf{tag}$		a countable set
Data values	$\mathbf{data}$		a countable set
Streams	$\mathbf{strm}$		finite sequences elements drawn from $\mathbf{data} \times \mathcal{P}(\mathbf{tag})$
Stream tuples	$\mathbf{strm}^n$		tuples of $n \in \mathbb{N}$ streams
Stream names	$\mathbf{strmName}$		a countable set
Environments	$\mathbf{Env}$	$=$	$\mathbf{strmName} \rightharpoonup \mathbf{strm}$
Set of filter names	$\mathbf{T}$		a countable set
Set of initializer keys	$\mathbf{K}$		a countable set
Set of initializer values	$\mathbf{V}$	$=$	$\mathbb{R} \cup \mathbb{B} \cup \mathbf{tag} \cup \ldots$
Filter signatures	$\mathbf{\Sigma}$	$=$	$\mathbf{T} \times (\mathbf{K} \rightharpoonup \mathbf{V}) \to n : \mathbb{N} \times m : \mathbb{N} \times (\mathbf{strm}^n \to \mathbf{strmLst}^m)$
Set of tPDVs	$\mathbf{P}$	$=$	$\mathbf{Q} \times \mathbf{L}$
Set of Logs	$\mathbf{L}$	$=$	$\mathbf{time} \times \mathbf{prin} \times \mathbf{strmName}$
Set of Snapshot Histories	$\mathbf{Q}$	$=$	$\mathbf{time} \to \mathbf{q}$
Set of Snapshots	$\mathbf{q}$	$=$	$\mathbf{\Sigma} \times \mathbf{Env} \times \{\!\|p\|\!\} \times \Phi$
Stream Name Metavariables	x, y, z	$\in$	$\mathbf{strmName}$
Filter Name Metavariable	T	$\in$	$\mathbf{T}$
Initialization Key Metavariable	k	$\in$	$\mathbf{K}$
Initialization Value Metavariable	v	$\in$	$\mathbf{V}$
Filter Initializer	I	$::=$	$\{\overrightarrow{k \texttt{ = } v}\}$
Concrete Stream Declaration	c	$::=$	$\textsf{concrete}\ \overrightarrow{x}$
Stream Definition	d	$::=$	$\overrightarrow{y} \texttt{ = } T\, I\, \overrightarrow{x}$
Transmute script (program)	p	$::=$	$c \mid p; \overrightarrow{d}$
Principal Identifiers	$\mathbf{prin}$		a countable set
Access Control Policies	Φ	$=$	$\mathbf{prin} \times \mathbf{strmName} \to \mathcal{P}(\{\textsf{read}, \textsf{write}\})$
Metavariable for principals	a	$\in$	$\mathbf{prin}$
Metavariable for streams	S	$\in$	$\mathbf{strm}$
Request	r	$::=$	$\textsf{get}\, a\, x \mid \textsf{put}\, a\, x\, S$

Fig. 4. Significant sets, metavariables and grammars in the corePDV model. We write $\{\!|p|\!\}$ for the set of all terms freely generated by grammar p.

to entire Transmute scripts as follows:

$$\frac{y \in \overrightarrow{y}}{\textsf{concrete}\ \overrightarrow{y} \blacktriangleright y \leadsto y}\ \textsc{inf-Conc}$$

$$\frac{d \blacktriangleright \overrightarrow{z} \leadsto y \qquad \overrightarrow{x} = \bigcup_{z \in \overrightarrow{z}} \{\, \overrightarrow{w} \mid p \blacktriangleright \overrightarrow{w} \leadsto z \,\}}{p; d \blacktriangleright \overrightarrow{x} \leadsto y}\ \textsc{inf-Assign}$$

$$\frac{d \blacktriangleright \not\leadsto y \qquad p \blacktriangleright \overrightarrow{x} \leadsto y}{p; d \blacktriangleright \overrightarrow{x} \leadsto y}\ \textsc{inf-Trans}$$

Experts may observe that the flow analysis does not account for implicit flows between filters. This is sound because inter-filter control flow is completely determined by static view stream definitions declarations, not by dynamic conditionals. Furthermore, sandboxing rules out covert channels.

We define noninterference in the usual manner [20]. In the following we write $E_1 \sim_{\overrightarrow{x}} E_1$ when $\overrightarrow{x} \subseteq \mathrm{dom}(E_1) \cap \mathrm{dom}(E_2)$ and $E_1(x) = E_2(x)$ for all $x \in \overrightarrow{x}$.

Definition 6 (Noninterference): Transmute scripts p satisfies $(\overrightarrow{x}, y)$–noninterference when for any E_1 and E_2 with $E_1 \sim_{\overrightarrow{x}} E_2$ and $\mathrm{dom}(E_1) = \mathrm{dom}(E_2) = \mathit{concrete}(p)$,

$$[\![p]\!]_\Sigma(E_1)(y) = [\![p]\!]_\Sigma(E_2)(y).$$

The following theorem shows that the behavior of well-formed Transmute scripts is accurately modeled by our static flow analysis and the associated non-interference properties.

Theorem 7 (Privacy): Suppose that $\Sigma \vdash p \triangleright \overrightarrow{w}$ and $p \blacktriangleright \overrightarrow{x} \leadsto y$ where $y \in \overrightarrow{w}$. Then p satisfies $(\overrightarrow{x}, y)$–noninterference.

Proof sketch: By induction on the structure of p.

V. Implementation and Evaluation

A. Implementation

Our PDV implementation maintains a directory structure for the user, containing a list of time-stamped *snapshot directories*, and a distinguished *working directory*. The snapshot directories collectively represent the fixed, immutable historical and present state of the PDV, and the working directory

is scratch space in which concrete streams, TRANSMUTE scripts, and metadata are assembled before being committed to snapshot directory. Accordingly pdv−read operations always examine the state of the unique, *current* snapshot, and pdv−write operations always target the working directory. When an authorized (according to the access control matrix) principal writes to a user's PDV using pdv−write, files in the working directory are updated in place. After a logical group of updates, the user can employ pdv−snapshot to create a new current snapshot directory based on the contents of the working directory. Thus a sequence of pdv−snapshot operations yield a time-ordered sequence of snapshot directories, containing a comprehensive history of PDV states. Separating the pdv−write and pdv−snapshot operations allows users to select a variety of policies for grouping writes into logical transactions or rejecting undesired writes.

View streams are generated lazily by pdv−write. If the stream being read is a view stream that does not yet exist, the snapshot's TRANSMUTE script is compiled and executed (see below for details) in order to generate the stream. Side-effect-freedom ensures that laziness does not effect computed view streams. For the same reason the PDV could (but does not currently) aggressively garbage collect view streams without effecting audit.

Audit of historical states proceeds as expected, by running a variant of the pdv−read command on old snapshots. And couterfactual audit is implemented by constructing a directory structure roughly parallel to the main PDV. The pdv−audit command copies snapshots directories into the audit space where their streams, filters, and TRANSMUTE scripts can be updated and read.

The TRANSMUTE compiler is a Python script that converts a single TRANSMUTE script into a single shell script. The translation is quite straightforward. For instance, the following TRANSMUTE snippet:

```
_in_westwood // _not_in_westwood =
    location
    → tag−circle westwood 34.073307 –118.448894 1.5km
    → split −tags westwood;
```

is translated to the following shell script:

```
cat location \
    | redirect westwood 34.073307 –118.448894 1.5km \
    | tag−circle \
    > __temp1

cat __temp1 \
    | redirect westwood \
    | split −tags \
    > __temp2

cat __temp2 \
    | redirect 1 \
    | substream \
    > _in_westwood

cat __temp2 \
    | redirect 2 \
    | substream \
    > _not_in_westwood
```

The translation splits each statement so that it invokes at most one filter, introducing temporary streams as necessary. The redirect command simply converts command-line arguments into arguments that are passed via standard input. Some temporary streams represent multiple logical streams and are implemented as a stream containing a sequence of substreams (e.g., __temp2, which is the ouptut from split −tags). The substream command is used to access each of these substreams by position.

Our run-time environment for filters is a modified version of the Lua-5.1.4 interpreter. Our modifications implement sandboxing by removing several Lua library functions that can lead to external dependencies. We modified the io module to eliminate all functionality except for access to stdin, stdout, and stderr. We modified the os module to remove date and time functions, and the the ability to fork new processes and to read/write environment variables. We modified the package module and a few global functions to remove the ability to dynamically load libraries. Finally, we removed the debug module altogether.

B. Audit

In this section, we demonstrate the audit capabilities of our approach by example. First we use the data-driven marketing example to illustrate how a user can manually audit untrusted code. Specifically, the user would like to verify that the information readable by the advertiser is just a set of business names. The user first copies the working directory into an internal audit snapshot with the pdv−audit command. If past data leakage is suspected, pdv−audit can also take the timestamp of a previous snapshot. The from_Advertiser stream must contain only plausible business names with corresponding GPS coordinates, and the for_Advertiser stream must be a subset of those business names. Since the stream format is space-insensitive in places, the advertiser might attempt to encode private location data within the space between records. Manual inspection exposes this behavior along with other subversive encoding tricks like record duplication.

Next, we use the the ambulation example to show how a user automatically checks a precisely defined view streams. In this particular example, a for_Ambulation record's GPS coordinate should identically match the GPS coordinate of a record in either the location or zips concrete stream, depending on its proximity to a given point. This property can be automatically checked by adding the following statement to the TRANSMUTE script for Ambulation within the audit environment:

```
audit_for_Ambulation =
    for_Ambulation // location // zips
    → audit−ambulation 34.073307 –118.448894 1.5km;
```

The audit−ambulation filter checks the property described above and writes any records that violate the property to the audit_for_Ambulation stream. The user can then read this stream to identify any suspicious behavior.

	time (s)	input (%)	output (%)	work (%)
Advertiser	0.55	50	6	44
Ambulation	8.63	39	5	53
Wakefulness	1.11	37	11	52

Fig. 5. Total performance by application

	time (s)	input (%)	output (%)	work (%)
nearby-advertisers	0.21	54	6	40
configure	0.06	20	19	61
acl	0.11	75	0	25

Fig. 6. Per-filter performance of Advertiser program

	time (s)	input (%)	output (%)	work (%)
index	0.28	25	10	65
split-tags	0.30	24	15	61
coarsen-location	6.55	49	1	50
tag-circle	0.30	23	15	62
merge	0.20	54	27	19
acl	0.09	75	0	25
configure	0.13	45	31	24

Fig. 7. Per-filter performance of Ambulation program

	time (s)	input (%)	output (%)	work (%)
count	0.13	62	9	29
join	0.14	64	9	27
configure	0.06	21	20	59
group-by	0.13	62	9	29
acl	0.11	75	0	25
project	0.13	12	10	78

Fig. 8. Per-filter performance of Wakefulness program

Finally, we use the wakefulness example to show how a user can check counterfactual scenarios. Specifically, the user would like to verify that if either the clinician_requests or survey concrete stream is empty, then the Wakefulness script will produce an empty requested_details view stream. The user first takes an audit snapshot and then overwrites either the clinician_requests or survey concrete stream to be the empty stream. The user can then execute the TRANSMUTE script and inspect the output to ensure compliance.

C. Performance

We measure performance for each application as the average time to build all view streams. We provide results on a per-application basis (Figure 5) as well as on a per-filter basis within each application (Figures 6-8). Most notably, the Ambulation application—which indexes and repeatedly searches a 2MB zip-code database—exhibits acceptable performance. Input handling is by far the most expensive part of the process. Input and output handling are, however, largely dependent on the implementation and could be optimized. Also note that the nearby−advertisers and coarsen−location filters perform similar operations on data sets of differing orders of magnitude. The nearby−advertisers filter indexes tens of records, and coarsen−location indexes tens of thousands, but both filters conduct searches over the same location input.

D. Security of the PDV model

The PDV provides strong guarantees to users through both its dependency analysis and its support for audit, and the noninterference theorem shows that a correctly implemented program does not leak information against a well-characterized attacker who observes the contents of output streams. However it is important to consider covert channels as well.

The PDV implementation is robust against internal timing channels, in which a malicious program attempts to transmit or infer information by causing or variations in . Because the sandbox forces filters to run without access to an explicit timer, non-blocking IO, or concurrent operations (which can be used to construct timers), filters lack an effective means to evaluate how quickly, or in what order, system event occur.

In contrast, the PDV architecture does not prevent external timing or resource exhaustion attacks. In an external timing attack, a curious observer makes inferences based on the amount of time it takes to service possibly maliciously selected system requests. In a resource exhaustion attack, external observers or malicious programs attempt to harm a system (or potentially convey information) by using up scarce resources. These attacks may be mitigated by higher-level mechanisms for timing [2] and by blacklisting filters that take too long or consume too many resources. Additionally, while a stream reader might surmise that a filter has crashed or diverged, this can only leak one bit of information since the PDV architecture does not allow outputs of running computations to be observed [1]. While sandboxing protects a PDV system executing filters from many resource exhaustion attacks, such as “fork bombs,” detection and blocklisting appear necessary to deal with a filter that produces large streams in an attempt to exhaust disk space.

The PDV maintains an auditable log of all attempted queries, whether or not allowed by the access control policy. This log is maintained separately from the data vault's streams so that unauthorized users cannot affect the results queries by creating “access denied” log entries.

VI. RELATED WORK

As mentioned in the introduction, several researchers have previously described a PDV-like architecture that allows users to retain control over their raw data and how that data is shared with third parties. Our key advance is the introduction of arbitrary computation within the PDV while retaining transparency for users. Closest to our setting is prior work on personal data vaults [11, 16]. That work employs a form of *granular* access control lists (ACLs) that support a fixed set of transformations on particular kinds of data. That work also provides a form of audit similar to ours but lacks support for counterfactual auditing. Finally, personal data vaults provide a *rule recommender* service, which helps users define policies that meet their intended privacy goals, while we lack such a mechanism.

Other work on PDV-like architectures has been explored in the context of social networking systems. Personal cloud butlers [14] allow users to write access-control policies in an extension to Datalog called SociaLite. The authors recognize

that this language is often not expressive enough and so allow auxiliary functions to be implemented in Java and Python, but they do not describe mechanisms to retain security and transparency in the face of such code. Virtual individual servers [4, 15] and Matryoshka-based networks [5] use the social group structure in the network to express access control policies. These systems naturally support data sharing policies that aggregate information across multiple users via the social network structure, while our work currently focuses on single-user policies.

TRANSMUTE is a form of *dataflow programming language*, in which a program is described as a directed graph that expresses the flow of data through the system. Dataflow languages have been used for a variety of purposes, for example high-performance streaming applications [17], extensible software routers [9], and modeling of embedded systems [7]. We propose a novel use of the dataflow style. Since filters in TRANSMUTE can be arbitrarily complex, TRANSMUTE can express sophisticated access control policies. At the same time, the use of the dataflow style (and the guarantees provided by sandboxing filters) ensures that all data dependencies are explicit and easy to identify, even for non-technical users.

Our system's support for audit via replay is inspired by similar mechanisms that have been proposed in the context of virtual machines and operating systems (e.g., [6, 8]), as well as programming languages [18], but is made simplified and made more powerful by the side-effect-free nature of filters. Because filters are side-effect free, counterfactural audit is able to partially alter old PDV states and compute meaningful results. This would not be possible were filters able to connect with external resources, such as web services, directly.

In proof carrying access control systems, such as Aura [19] and Grey [3], requests to security-sensitive APIs must be accompanied by unforgeable proof objects explaining why access should be allowed, and a runtime system ensures these objects are logged appropriately. Thus Aura and Grey allow users or administrators to answer *why* was access granted to a resource. In contrast the PDV audit mechanism is designed to help users understand *how* data flows or flowed from an input to an output stream.

TRANSMUTE obtains strong guarantees about secure information flow by sandboxing filters and making the dependencies among filters and data streams explicit. An alternative approach would be to require filters to be written in a language that employs a security type system (e.g., [12, 13, 20]). That approach would enforce noninterference at compile time, without requiring any run-time overhead, and it would provide finer-grained notions of noninterference than our approach, which treats filters as black boxes. On the other hand, security type systems place a burden on filter implementers due to the need for annotations and can be overly conservative in the face of complex language features.

VII. CONCLUSIONS AND FUTURE WORK

This paper presents an expressive and secure approach to sharing data with untrusted third parties. It is expressive because access-control policies are written in our TRANSMUTE scripting language, which pipes streams of data through *filters* that support general computation. It is secure through a practical combination of several techniques: a sandboxing mechanism for filters; a dependency analysis on TRANSMUTE scripts that ensures a form of noninterference; and a flexible mechanism for auditing current, past, and hypothetical data-sharing scenarios. We have formalized our approach and proven a noninterference theorem. We have also implemented our approach and illustrate its benefits via several case studies. In future work, we plan to further evaluate our PDV design through controlled user studies as well as a small deployment to mediate between smartphone users and real applications.

REFERENCES

[1] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-88312-8. doi: http://dx.doi.org/10.1007/978-3-540-88313-5_22. URL http://dx.doi.org/10.1007/978-3-540-88313-5_22.

[2] Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of multithreaded programs by compilation. *ACM Trans. Inf. Syst. Secur.*, 13:21:1–21:32, July 2010. ISSN 1094-9224. doi: http://doi.acm.org/10.1145/1805974.1805977. URL http://doi.acm.org/10.1145/1805974.1805977.

[3] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Detecting and resolving policy misconfigurations in access-control systems. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, SACMAT '08, pages 185–194, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-129-3.

[4] Ramón Cáceres, Landon Cox, Harold Lim, Amre Shakimov, and Alexander Varshavsky. Virtual individual servers as privacy-preserving proxies for mobile devices. In *Proceedings of the 1st ACM workshop on Networking, systems, and applications for mobile handhelds*, MobiHeld '09, pages 37–42, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-444-7.

[5] Leudo Antonio Cutillo, Refik Molva, and Thorsten Strufe. Privacy preserving social networking through decentralization. In *Proceedings of the Sixth international conference on Wireless On-Demand Network Systems and Services*, WONS'09, pages 133–140, Piscataway, NJ, USA, 2009. IEEE Press. ISBN 978-1-4244-3374-2. URL http://portal.acm.org/citation.cfm?id=1688899.1688918.

[6] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02,

pages 211–224, New York, NY, USA, 2002. ACM. ISBN 978-1-4503-0111-4.

[7] J. Eker, J.W. Janneck, E.A. Lee, Jie Liu, Xiaojun Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. Taming heterogeneity - the ptolemy approach. *Proceedings of the IEEE*, 91(1):127 – 144, jan 2003.

[8] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX ATEC '05*, 2005.

[9] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[10] M. Mun, S. Reddy, K. Shilton, N. Yau, J. Burke, D. Estrin, M. Hansen, E. Howard, R. West, and P. Boda. Peir, the personal environmental impact report, as a platform for participatory sensing systems research. In *MobiSys '09*, 2009.

[11] M. Mun, S. Hao, M. Mishra, K. Shilton, J. Burke, D. Estrin, M. Hansen, and R. Govindan. Personal data vaults: A locus of control for personal data streams. In *ACM CoNext*, 2010.

[12] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.

[13] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.*, 25 (1):117–158, 2003. ISSN 0164-0925.

[14] S. Seong, M. Nasielski, J. Seo, D. Sengupta, S. Hangal, S. K. Teh, R. Chu, B. Dodson, and M. S. Lam. Architecture and implementation of a decentralized social networking platform. Draft, 2009. URL http://prpl.stanford.edu/papers/prpl09.pdf.

[15] A. Shakimov, H. Lim, R. Caceres, L.P. Cox, K. Li, Dongtao Liu, and A. Varshavsky. Vis-a-vis: Privacy-preserving online social networking via virtual individual servers. In *Third International Conference on Communication Systems and Networks (COMSNETS 2011)*, pages 1 –10, January 2011.

[16] K. Shilton, J. A. Burke, D. Estrin, R. Govindan, M. Hansen, J. Kang, and M. Mun. Designing the personal data stream: Enabling participatory privacy in mobile personal sensing. In *TPRC*, 2009.

[17] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, 2002.

[18] A. Tolmach and A. Appel. Debugging Standard ML without reverse engineering. In *LFP*, 1990.

[19] Jeffrey A. Vaughan, Limin Jia, Karl Mazurak, and Steve Zdancewic. Evidence-based audit. In *Computer Security Foundations Symposium (CSF '08)*, pages 177–191, 2008.

[20] D. Volpano, Smith G, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

APPENDIX

A. Compution in corePDV

In this section we define how to compute using a PDV. In the sequel we will use the term *query* to refer to the side-effect free operation of looking up data in a PDV. In contrast **get** and **put** *requests* may both query and update a PDV.

The key PDV operation is answering queries of the form "what is the value of the stream named x?" or "what was the value of x as time t?" The answer may depend, due to access-control, on which principal is posing the query, as well as the time at which the query is submitted. Figure 9 shows how such queries are evaluated. A result of **err** indicates a permissions error and a result of $\perp$ indicates an unexpected failure. Permission errors are detected by check a PDV's ϕ field. Failures $\perp$ occur when looking up an undefined stream name, or using a transducer with the wrong arity; we will later show that such failures $\perp$ cannot occur in well-formed code. In a successful query for time t, the snapshot $Q(t) = (\Sigma, E_0, p, \phi)$ is used to evaluate script p on environment E_0. The evaluation process, $[\![p]\!]_\Sigma(E_0)$, repeatedly updates the environment by applying transducers named in view stream declarations.

Note that in the definition of $[\![\cdot]\!]$, application $f(\overrightarrow{E(x)}) = f(E(x_1), \ldots, E(x_n))$ is defined, because the form of Σ, with the constraints on $|x|$ and $|y|$, ensure that the transducer is applied with correct arity.

At the top level, we model PDVs as reactive systems that take and respond to requests. For the purpose of modeling we have a single update request that replaces the old stream value with a new stream; an implementation would include insertion, deletion, etc.

A PDV responds to requests as shown in Figure 10. To handle a get, we query the PDV and log the request. To handle a put, we update the requested stream in the query processor history, provided it's allowed by policy. The time parameter is intended to represent the "current time," or "right now." This could be treated more formally but would complicate the model while adding little value. PDV request handling is defined to yield **err** in the event of any failure; this is intended to prevent unauthorized users from learning about the structure of the PDV (i.e. which streams are defined) via queries.

B. Proofs

Definition Environments E_1 and E_2 are equivalent to observations at $\overrightarrow{x}$, written $E_1 \sim_{\overrightarrow{x}} E_2$, when $\overrightarrow{x} \subseteq \mathrm{dom}(E_1) \cap \mathrm{dom}(E_2)$ and for all $x \in \overrightarrow{x}, E_1(x) = E_2(x)$.

Definition 8 ($concrete(p)$): A script's concrete stream names are given by

$$\begin{aligned} concrete(\textbf{concrete } \vec{x}) &= \overrightarrow{x} \\ concrete(p; d) &= concrete(p) \end{aligned}$$

Theorem 2 Suppose $\Sigma \vdash p \triangleright \overrightarrow{z}$ and $dom(E)$ is the set of concrete streams in p. Then $[\![p]\!]_\Sigma(E) = E'$ where $\mathrm{dom}(E') = \overrightarrow{z}$.

Proof: By structural induction on the $\cdot \vdash \cdot \triangleright \cdot$ derivation. The WF-CONC case is immediate.

$$
\begin{array}{rcl}
\mathit{query} & : & \mathbf{Q} \times \mathbf{time} \times \mathbf{strmName} \times \mathbf{prin} \rightarrow \mathbf{strm} \cup \{\textsf{err}, \bot\} \\
\mathit{query}(Q, t, x, a) & = & \mathit{query}(Q(t), x, a)
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{query} & : & \mathbf{q} \times \mathbf{strmName} \times \mathbf{prin} \rightarrow \mathbf{strm} \cup \{\textsf{err}, \bot\} \\
\mathit{query}((\Sigma, E, p, \phi), x, a) & = & \begin{cases} [\![p]\!]_\Sigma(E)(x) & \textsf{read} \in \phi(a, x) \\ \textsf{err} & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\ ]\!] & : & \{\!|p|\!\} \times \mathbf{\Sigma} \times \mathbf{Env} \rightarrow (\mathbf{Env} \cup \{\bot\}) \\
[\![c]\!]_\Sigma(E) & = & E \\
[\![p; d]\!]_\Sigma(E) & = & [\![d]\!]_\Sigma([\![p]\!]_\Sigma(E))
\end{array}
$$

$$
\begin{array}{rcl}
[\![\ ]\!] & : & \{\!|d|\!\} \times \mathbf{\Sigma} \times (\mathbf{Env} \cup \{\bot\}) \rightarrow (\mathbf{Env} \cup \{\bot\}) \\
[\![_]\!]_\Sigma(\bot) & = & \bot \\
[\![\overrightarrow{y} \textsf{ = } T\, I\, \overrightarrow{x}]\!]_\Sigma(E) & = & \begin{cases} E[\, \overrightarrow{y} \mapsto f(\overrightarrow{E(x)})\,] & |\overrightarrow{x}| = n,\ |\overrightarrow{y}| = m, \\ & \overrightarrow{x} \subseteq \mathrm{dom}(E),\ \text{and } \overrightarrow{y} \cap \mathrm{dom}(E) = \emptyset \\ \bot \end{cases} \\
& & \quad \text{where } \Sigma(T, I) = (n, m, f)
\end{array}
$$

Fig. 9. Query and script evaluation.

$$
\begin{array}{rcl}
\mathit{handle} & : & \mathbf{P} \times \mathbf{time} \times \{\!|r|\!\} \rightarrow \mathbf{P} \times (\mathbf{strm} \cup \{\textsf{err}, \textsf{ok}\}) \\
\mathit{handle}((Q, L), t, \textsf{get}\, a\, x) & = & \begin{cases} ((Q, L'), R) & R \in \mathbf{strm} \cup \{\textsf{err}\} \\ ((Q, L'), \textsf{err}) & \text{otherwise} \end{cases} \\
& & \quad \text{where } L' = (L, (t, a, x)) \\
& & \quad \text{and} \quad \mathrm{R} = \mathit{query}(Q, t, a, x) \\
\mathit{handle}((Q, L), t, \textsf{put}\, a\, x\, S) & = & \begin{cases} ((\mathit{update}(Q, t, x, S), L), \textsf{ok}) & \mathit{writeOkAt}(Q(t'), x, a) = \textsf{ok} \\ ((Q, L), \textsf{err}) & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{update} & : & \mathbf{Q} \times \mathbf{time} \times \mathbf{strmName} \times \mathbf{strm} \rightarrow \mathbf{Q} \\
\mathit{update}(Q, t, x, S)(t') & = & \begin{cases} \mathit{update}(Q(t'), x, S) & t' \geq t \\ Q(t') & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{update} & : & \mathbf{q} \times \mathbf{strmName} \times \mathbf{strm} \rightarrow \mathbf{q} \\
\mathit{update}((\Sigma, E, p, \phi), x, S) & = & (\Sigma, E[\, x \mapsto S\,], p, \phi)
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{writeOkAt} & : & \mathbf{q} \times \mathbf{strmName} \times \mathbf{prin} \rightarrow \{\textsf{err}, \textsf{ok}\} \\
\mathit{writeOkAt}((\Sigma, E, p, \phi), x, a) & = & \begin{cases} \textsf{ok} & \textsf{write} \in \phi(x, a) \text{ and } x \in \mathrm{dom}(E) \\ \textsf{err} & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 10. Semantics of PDV update operations

In the WF-LET case we have $p = p_0; \overrightarrow{y}$ **=** $T\ I\ \overrightarrow{x}$ and $\overrightarrow{z} = \overrightarrow{y} \cup \overrightarrow{w}$. Furthermore, $\Sigma(T, I) = (|x|, |y|, f)$, and $\overrightarrow{x} \subseteq \overrightarrow{w}$, and $\overrightarrow{y} \cap \overrightarrow{w} = \emptyset$. We want to show that there exists some environment $E'' = [\![p]\!]_\Sigma E$ where $\mathrm{dom}(E'') = \overrightarrow{y} \cup \overrightarrow{w}$. Calculate as follows:

$$\begin{array}{ll} [\![p_0; \overrightarrow{y} \texttt{=} T\ I\ \overrightarrow{x}]\!]_\Sigma & \\ \quad = \ [\![\overrightarrow{y} \texttt{=} T\ I\ \overrightarrow{x}]\!]_\Sigma([\![p_0]\!]_\Sigma(E)) & \\ \quad = \ [\![\overrightarrow{y} \texttt{=} T\ I\ \overrightarrow{x}]\!]_\Sigma(E') & \text{by the IH} \\ \quad = \ f(E'[\ \overrightarrow{y} \mapsto f(\overrightarrow{E'(x)})\ ] & \end{array}$$

The last step relies on the assumptions above, and the induction hypothesis which gives $\mathrm{dom}(E') = \overrightarrow{w}$. Conclude by observing that $\mathrm{dom}(E'[\ \overrightarrow{y} \mapsto f(\overrightarrow{E'(x)})\ ]) = \mathrm{dom}(E') \cup \overrightarrow{y} = \overrightarrow{y} \cup \overrightarrow{w}$. ■

Theorem 7 Suppose that $\Sigma \vdash p \rhd \overrightarrow{w}$ and $p \blacktriangleright \overrightarrow{x} \rightsquigarrow y$ where $y \in \overrightarrow{w}$. Then p satisfies $(\overrightarrow{x}, y)$–noninterference.

Proof: By induction on the structure of p. In the case that $p =$ **concrete** $\overrightarrow{y}$ we conclude immediately. Suppose instead that $p = p_0; d$. Inverting the derivation of the influence relation yields two case.

Case INF-TRANS: By inversion $d \blacktriangleright \not\rightsquigarrow y$ and $p_0 \blacktriangleright \overrightarrow{x} \rightsquigarrow y$. In turn, this gives $[\![d]\!]_\Sigma(E)(y) = E(y)$ for any E with $y \in \mathrm{dom}(E)$. Finish using the following equational reasoning:

$$\begin{array}{rcl} [\![p]\!]_\Sigma(E_1)(y) & = & [\![p_0; d]\!]_\Sigma(E_1)(y) \\ & = & [\![d]\!]_\Sigma([\![p_0]\!]_\Sigma(E_1))(y) \\ & = & [\![p_0]\!]_\Sigma(E_1)(y) \\ & & \quad \text{as } y \in \mathrm{dom}(E_1) \text{ by Theorem 2} \\ & = & [\![p_0]\!]_\Sigma(E_2)(y) \\ & & \quad \text{by the I.H.} \\ & = & [\![d]\!]_\Sigma([\![p_0]\!]_\Sigma(E_2))(y) \\ & = & [\![p]\!]_\Sigma(E_2)(y) \end{array}$$

Case INF-ASSIGN: By inversion $d \blacktriangleright \overrightarrow{z} \rightsquigarrow y$ and $d = \overrightarrow{y}$ **=** $T\ I\ \overrightarrow{z}$ where y is the ith element of $\overrightarrow{y}$ for some i. Inversion also yields

$$\overrightarrow{x} = \bigcup_{z \in \overrightarrow{z}} \{\ \overrightarrow{w} \mid p_0 \blacktriangleright \overrightarrow{w} \rightsquigarrow z\ \}$$

so by the induction hypothesis, we have $[\![p_0]\!]_\Sigma(E_1)(z) = [\![p_0]\!]_\Sigma(E_2)(z)$ for any $z \in \overrightarrow{z}$. Let $(n, _, f) = \Sigma(T, I)$ and conclude with the following equational reasoning:

$$\begin{array}{l} [\![p]\!]_\Sigma(E_1)(y) \\ \quad = \ [\![d]\!]([\![p_0]\!]_\Sigma(E_1))(y) \\ \quad = \ \pi_i(f([\![p_0]\!]_\Sigma(E_1)(z_1), \ldots, [\![p_0]\!]_\Sigma(E_1)(z_n))) \\ \quad = \ \pi_i(f([\![p_0]\!]_\Sigma(E_2)(z_1), \ldots, [\![p_0]\!]_\Sigma(E_2)(z_n))) \\ \quad = \ [\![p]\!]_\Sigma(E_2)(y) \end{array}$$

■