A Core Calculus for Provenance

Umut A. Acar[1], Amal Ahmed[2], James Cheney[3], and Roly Perera[1]

[1] Max Planck Institute for Software Systems
{umut,rolyp}@mpi-sws.org
[2] Indiana University
amal@cs.indiana.edu
[3] University of Edinburgh
jcheney@inf.ed.ac.uk

Abstract. Provenance is an increasing concern due to the revolution in sharing and processing scientific data on the Web and in other computer systems. It is proposed that many computer systems will need to become provenance-aware in order to provide satisfactory accountability, reproducibility, and trust for scientific or other high-value data. To date, there is not a consensus concerning appropriate formal models or security properties for provenance. In previous work, we introduced a formal framework for provenance security and proposed formal definitions of properties called disclosure and obfuscation.

This paper develops a core calculus for provenance in programming languages. Whereas previous models of provenance have focused on special-purpose languages such as workflows and database queries, we consider a higher-order, functional language with sums, products, and recursive types and functions. We explore the ramifications of using traces based on operational derivations for the purpose of comparing other forms of provenance. We design a rich class of provenance views over traces. Finally, we prove relationships among provenance views and develop some solutions to the disclosure and obfuscation problems.

1 Introduction

Provenance, or meta-information about the origin, history, or derivation of an object, is now recognized as a central challenge in establishing trust and providing security in computer systems, particularly on the Web. Essentially, provenance management involves instrumenting a system with detailed monitoring or logging of auditable records that help explain how results depend on inputs or other (sometimes untrustworthy) sources. The security and privacy ramifications of provenance must be understood in order to safely meet the needs of users that desire provenance without introducing new security vulnerabilities or compromising the confidentiality of other users.

The lack of adequate provenance information can cause (and has caused) major problems, which we call *provenance failures* [11]. Essentially, a provenance failure can arise either from failure to *disclose* some key provenance information to users (for example, if a years-out-of-date story causes investors to panic about a company's financial stability [7]), or from failure to *obfuscate* some sensitive provenance information (for example, if a Word document is published with supposedly secret contributors' identities logged in its change history [27]). To address these problems, a number of forms

P. Degano and J.D. Guttman (Eds.): POST 2012, LNCS 7215, pp. 410–429, 2012.

of provenance have been proposed for different computational models, including *why* and *where* provenance [6], *dependency* provenance [9], and a variety of other ad hoc techniques [3,24].

Prior work on provenance and security. Our previous work [9] appears to have been the first to explicitly relate information-flow security to a form of provenance, called *dependency provenance*. Provenance has been studied in language-based security by Cirillo et al. [13] and by Swamy et al. [26,25]. Both focus on specifying and enforcing security policies involving provenance tracking alongside many other concerns, and not on defining provenance semantics or extraction techniques. Work on secure auditing [21,19] and expressive programming languages for security is also related, but this work focuses on explicitly manipulating proofs of authorization or evidence about protocol or program runs rather than automatically deriving or securing provenance information.

There is also some work directly addressing security for provenance [12,8,15]. Chong [12] gave candidate definitions of *data security* and *provenance security* using a trace semantics, based in part on an earlier version of our trace model. Davidson et al. [15] studied privacy for provenance in scientific workflows, focusing on complexity lower bounds. Cheney [8] gave an abstract framework for provenance, proposed definitions of properties called *obfuscation* and *disclosure*, and discussed algorithms and complexity results for instances of this framework including finite automata, workflows, and the semiring model of database provenance [18].

In this paper, we build on prior work on provenance security by studying the disclosure and obfuscation properties of different forms of provenance in the context of a higher-order, pure, functional language. To illustrate what we mean by provenance, we present examples of programming with three different forms of provenance in Transparent ML (TML), a prototype implementation of the ideas of this paper.

1.1 Examples

Where-provenance. Where-provenance [6,5] identifies at most one source location from which a part of the output was copied. For example, consider the following TML session:

```
- f [(1,2), (4,3), (5,6)];
val it = [(5,6), (3,4), (1,2)]
```

Without access to the source code, one can guess that f is doing something like

$$\mathit{reverse} \circ (\mathit{map}\ (\lambda(x,y).\texttt{if}\ x < y\ \texttt{then}\ (x,y)\ \texttt{else}\ (y,x)))$$

However, by providing where-provenance information, the system can explain whether the numbers in the result were copied from the input or constructed in some other way:

```
- trace (f [(1@L1,2@L2),(4@L3,3@L4),(5@L5,6@L6)]);
it = <trace> : ({L1:int,...}, (int*int) list) trace
- where it;
val it = [(5@L5,6), (3@L4,4), (1@L1,2)]
```

This shows that f contrives to copy the first elements of the returned pairs but construct the second components.

Dependency provenance. Dependency provenance [9] is an approach that tracks a set of all source locations on which a result depends. For example, if we have:

```
- g [(1,2,3), (4,5,6)];
val it = [6,6] : int list
```

we again cannot tell much about what g does. By tracing and asking for dependency provenance, we can see:

```
- trace (g [(1@L1,2@L2,3@L3),(4@L4,5@L5,6@L6)]);
val it = <trace> : ({L1:int,...}, int list) trace
- dependency it;
val it = [6@{L1,L2,L3}, 6@{L1,L2,L3}]
```

This suggests that g is simply summing the first triple and returning the result twice, without examining the rest of the list. We can confirm this as follows:

```
- trace (g ((1@L1,2@L2,3@L3)::[]@L));
val it = <trace> : ({L1:int,...}, int list) trace
- dependency it;
val it = [6@{L1,L2,L3}]
```

The fact that L does not appear in the output confirms that g does not look further into the list.

Expression provenance. A third common form of provenance is an expression graph or tree that shows how a value was computed by primitive operations. For example, consider:

```
- (h 3, h 4, h 5)
val it = (6,24,120);
```

We might conjecture that h is actually the factorial function. By tracing h and extracting expression provenance, we can confirm this guess (at least for the given inputs):

```
- trace (h (4@L));
val it = <trace> : ({L:int}, int) trace
- expression it;
val it = 24@{L * (L-1) * (L-2) * (L-3)  * 1}
```

In this case where-provenance and dependency provenance would be uninformative since the result is not copied from, and obviously depends on, the input.

This kind of provenance is used extensively in *workflow* systems often used in e-science [20], where the main program is a high-level process coordinating a number of external (and often concurrent) program or RPC calls, for example, image-processing steps or bulk data transformations, which we could model by adding primitive image-processing operations and types to our language. Thus, even though the above examples use fine-grained primitive operations, this model is also useful for coarse-grained provenance-tracking.

Provenance security. The three models of provenance above represent useful forms of provenance that might increase users' trust or confidence that they understand the results of a program. However, if the underlying data, or the structure of the computation, is sensitive, then making this information available may lead to inadvertent vulnerabilities, by making it possible for users to infer information they cannot observe directly. This is a particular problem if we wish to disclose part of the result of a program, and provenance that justifies part of the result, while keeping other parts of the program's execution, input, or output confidential. As a simple example, consider a program $\texttt{if}\ x \neq 1\ \texttt{then}\ (y, y)\ \texttt{else}\ (z, w)$. If x is sensitive, but z and w happen to both equal 42, then it is safe to reveal the result $(42, 42)$. However, any of the above forms of provenance make it possible to distinguish which branch was taken because the two different copies of 42 in z and w will have different provenance. Thus, if the provenance information is released then a principal can infer that the second branch was taken, and hence, $x = 1$. In technical terms, we cannot *disclose* any of the above forms of provenance for the result while *obfuscating* the fact that $x = 1$.

1.2 Summary

Contributions. In this paper, we build on, and refine, the provenance security framework previously introduced by Cheney [8]. We introduce a core language with replayable execution traces for a call-by-value, higher-order functional language, and make the following technical contributions:

- Refined definitions of obfuscation and disclosure (Sec. 2).
- A core calculus defining traced execution for a pure functional programming language (Sec. 3).
- A generic provenance extraction framework that includes several previously-studied forms of provenance as instances (Sec. 4).
- An analysis of disclosure and obfuscation guarantees provided by different forms of provenance, including techniques based on slicing execution traces (Sec. 5).

Outline. Section 2 briefly recapitulates the framework introduced by Cheney [8] and refines some definitions. We present the (standard) syntax and tracing semantics of TML in Section 3. In Section 4 we introduce a framework for querying and extracting provenance views from traces, including the three models discussed above. Section 5 presents our main results about disclosure, obfuscation, and trace slicing. Section 6 presents related work and Section 7 concludes.

2 Background

We recapitulate the main components of the provenance security framework of Cheney [8]. The framework assumes a given set of *traces* $\mathcal{T}$, together with a collection $\mathcal{Q}$ of possible *trace queries* $Q : \mathcal{T} \to \mathbb{B}$. These represent properties of traces that the system designer may want to protect or that legitimate users or attackers of the system may want to learn. In the previous paper, we considered refinements to take into account

the knowledge of the principals about the possible system behaviors. In this paper, we assume that all traces $\mathcal{T}$ are considered possible by all principals, for simplicity.

For each principal A, fix a set Ω_A of the possible *provenance views* offered to A, and a function $P_A : \mathcal{T} \to \Omega_A$ mapping each trace to A's provenance view of the trace. For the purposes of this paper, we do not consider interactions among multiple principals, so we typically consider only one principal and omit the A subscripts. We may write $(\Omega, P : \mathcal{T} \to \Omega)$ or just (Ω, P) for a provenance view. Also, we sometimes write $Q : \Omega \to \mathbb{B}$ for a provenance query, that is, a query on a provenance view.

Given this framework, we proposed the following definitions:

Definition 1 (Disclosure). *A query Q is* disclosed *by a provenance view (Ω, P) if for every $t, t' \in \mathcal{T}$, if $P(t) = P(t')$ then $Q(t) = Q(t')$.*

In other words, disclosure means that there can be no traces t, t' that have the same provenance view but where one satisfies the query and the other does not.

Definition 2 (Obfuscation). *A query Q is* obfuscated *by a provenance view (Ω, P) if for every t in $\mathcal{T}$, there exists $t' \in \mathcal{T}$ such that $P(t) = P(t')$ and $Q(t) \neq Q(t')$.*

Thus, obfuscation is not exactly the opposite of disclosure; instead, it means that for every trace there is another trace with the same provenance view but different Q-value. This means that a principal that has access to the provenance view but not the trace cannot be certain that Q is satisfied or not satisfied by the underlying trace.

The definitions above turn out to be too strong; in this paper we will also consider some weaker versions of disclosure and obfuscation.

Definition 3. *We say that $P : \mathcal{T} \to \Omega$* positively discloses *$Q : \mathcal{T} \to \mathbb{B}$ via query $Q' : \Omega \to \mathbb{B}$ if for every t, if $Q'(P(t))$ then $Q(t)$.*

In other words, positive disclosure means that there is a query Q' on the provenance that safely overapproximates Q on the underlying trace. If $Q'(P(t))$ holds then we know $Q(t)$ holds but otherwise we may not learn anything about t.

Definition 4. *We say that P* positively obfuscates *$Q : \mathcal{T} \to \mathbb{B}$ if for every t satisfying Q there exists a trace t' falsifying Q such that $P(t) = P(t')$.*

In other words, positive obfuscation means that the provenance never reveals that Q holds of the trace, but it may reveal that Q fails. This weaker notion is useful for asserting that sensitive data is protected: if the sensitive data is not present in the trace then it is harmless to reveal this, but if the sensitive data is present then the provenance should hide enough information to make its presence uncertain.

Dual notions of negative disclosure and obfuscation can be defined as positive disclossure or obfuscation of $\neg Q$ respectively.

Proposition 1. *If P both positively discloses and negatively discloses Q via Q', then P discloses Q. Similarly, if P both positively and negatively obfuscates Q then P obfuscates Q.*

In the previous paper, we gave several examples of instances of this framework. Here, for illustration, we just review one such instance, given by finite automata.

$$
\begin{array}{lrl}
\text{Types} & \tau & ::= b \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \to \tau_2 \mid \mu\alpha.\tau \mid \alpha \\
\text{Type contexts} & \Gamma & ::= [x_1 : \tau_1, \ldots, x_n : \tau_n] \\
\text{Code pointers} & \kappa & ::= f(x).e \\
\text{Matches} & m & ::= \{\mathtt{inl}(x_1).e_1; \mathtt{inr}(x_2).e_2\} \\
\text{Values} & v & ::= c \mid (v_1, v_2) \mid \mathtt{inl}(v) \mid \mathtt{inr}(v) \mid \mathtt{roll}(v) \mid \langle \kappa, \gamma \rangle \\
\text{Expressions} & e & ::= c \mid x \mid \oplus(\overline{e}) \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \\
 & & \mid\ (e_1, e_2) \mid \mathtt{fst}(e) \mid \mathtt{snd}(e) \mid \mathtt{inl}(e) \mid \mathtt{inr}(e) \\
 & & \mid\ \mathtt{roll}(e) \mid \mathtt{unroll}(e) \mid \mathtt{fun}\ \kappa \mid \mathtt{case}\ e\ \mathtt{of}\ m \mid (e\ e') \\
\text{Traces} & T & ::= c \mid x \mid \oplus(\overline{T}) \mid \mathtt{let}\ x = T_1\ \mathtt{in}\ T_2 \\
 & & \mid\ (T_1, T_2) \mid \mathtt{fst}(T) \mid \mathtt{snd}(T) \mid \mathtt{inl}(T) \mid \mathtt{inr}(T) \\
 & & \mid\ \mathtt{roll}(T) \mid \mathtt{unroll}(T) \mid \mathtt{fun}\ \kappa \\
 & & \mid\ \mathtt{case}\ T \triangleright_{\mathtt{inl}} x.T_1 \mid \mathtt{case}\ T \triangleright_{\mathtt{inr}} x.T_2 \mid (T_1\ T_2) \triangleright_\kappa f(x).T \\
\text{Environments} & \gamma & ::= [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]
\end{array}
$$

Fig. 1. Abstract syntax of Core TML

Example 1 (Automata provenance framework). The set of traces $\mathcal{T}_M$ of an automaton $M = (\Sigma, Q, q_0, \delta, F)$ is the set $Q(\Sigma Q)^*$ of alternating sequences of states and alphabet letters. The queries are simply regular subsets of $\mathcal{T}_M$. The provenance views are given by finite-state transducers. We showed that disclosure is decidable for all queries and views and that obfuscation is decidable for all queries and views whose range is finite. It is unknown whether obfuscation is decidable in the general case.

We now proceed to instantiate the framework with traces generated by a much richer language, with corresponding notions of trace query and provenance view.

3 Core Language

We will develop a core language for provenance based on a standard, typed, call-by-value, pure language, called Transparent ML, or TML. The syntax of TML types, expressions, traces, and other syntactic classes is shown in Figure 1. We include standard constructs for dealing with binary pairs, binary sums, recursive types, and recursive functions; more general constructs such as records, datatypes, or simultaneous recursive functions can of course be handled without difficulty. In $f(x).e$, both f and x are variable names; f is the name of the recursively defined function while x is the name of the argument.

We abbreviate functional terms of the form $f(x).e$ using the letter κ, when convenient; similarly, we often abbreviate the expression $\{\mathtt{inl}(x_1).e_1; \mathtt{inr}(x_2).e_2\}$ as m. We sometimes refer to κ or m as a *code pointer* or *match pointer* respectively; in a fixed program, there are a fixed finite number of such terms and so we can share them instead of explicitly copying them when used in traces.

$\boxed{\gamma, e \Downarrow v, T}$

$$\frac{}{\gamma, x \Downarrow \gamma(x), x} \qquad \frac{}{\gamma, c \Downarrow c, c} \qquad \frac{}{\gamma, \mathtt{fun}\ \kappa \Downarrow \langle \kappa, \gamma \rangle, \mathtt{fun}\ \kappa} \qquad \frac{\gamma, \overline{e} \Downarrow \overline{v}, \overline{T}}{\gamma, \oplus(\overline{e}) \Downarrow \hat{\oplus}(\overline{v}), \oplus(\overline{T})}$$

$$\frac{\gamma, e_1 \Downarrow v_1, T_1 \qquad \gamma, e_2 \Downarrow v_2, T_2}{\gamma, (e_1, e_2) \Downarrow (v_1, v_2), (T_1, T_2)} \qquad \frac{\gamma, e \Downarrow (v_1, v_2), T}{\gamma, \mathtt{fst}(e) \Downarrow v_1, \mathtt{fst}(T)}$$

$$\frac{\gamma, e \Downarrow v, T}{\gamma, \mathtt{inl}(e) \Downarrow \mathtt{inl}(v), \mathtt{inl}(T)} \qquad \frac{\gamma, e_1 \Downarrow v_1, T_1 \qquad \gamma[x \mapsto v_1], e_2 \Downarrow v_2, T_2}{\gamma, \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Downarrow v_2, \mathtt{let}\ x = T_1\ \mathtt{in}\ T_2}$$

$$\frac{\gamma, e \Downarrow v, T}{\gamma, \mathtt{roll}(e) \Downarrow \mathtt{roll}(v), \mathtt{roll}(T)} \qquad \frac{\gamma, e \Downarrow \mathtt{roll}(v), T}{\gamma, \mathtt{unroll}(e) \Downarrow v, \mathtt{unroll}(T)}$$

$$\frac{(\mathtt{inl}(x_1).e_1 \in m) \qquad \gamma, e \Downarrow \mathtt{inl}(v), T \qquad \gamma[x_1 \mapsto v], e_1 \Downarrow v_1, T_1}{\gamma, \mathtt{case}\ e\ \mathtt{of}\ m \Downarrow v_1, \mathtt{case}\ T \triangleright_{\mathtt{inl}} x_1.T_1}$$

$$\frac{(\mathtt{inr}(x_2).e_2 \in m) \qquad \gamma, e \Downarrow \mathtt{inr}(v), T \qquad \gamma[x_2 \mapsto v], e_2 \Downarrow v_2, T_2}{\gamma, \mathtt{case}\ e\ \mathtt{of}\ m \Downarrow v_2, \mathtt{case}\ T \triangleright_{\mathtt{inr}} x_2.T_2}$$

$$\frac{\begin{array}{c}\gamma, e_1 \Downarrow \langle \kappa, \gamma' \rangle, T_1 \\ (\kappa = f(x).e) \qquad \gamma, e_2 \Downarrow v_2, T_2 \qquad \gamma'[f \mapsto \langle \kappa, \gamma' \rangle, x \mapsto v_2], e \Downarrow v, T\end{array}}{\gamma, (e_1\ e_2) \Downarrow v, (T_1\ T_2) \triangleright_\kappa f(x).T}$$

Fig. 2. Dynamic semantics of Core TML: selected rules for expression evaluation

3.1 Dynamic Semantics

We augment a standard large-step operational semantics for TML by adding a parameter T, which records a trace of the evaluation of the expression. The judgment $\gamma, e \Downarrow v, T$ says that in environment γ, expression e evaluates to value v with trace T. (Traces were defined in Figure 1.) Many of the trace forms are isomorphic to the corresponding expression forms. The exceptions are the case and application evaluation rules. In either case, the first argument is evaluated to determine what expression to evaluate to obtain the final result. For case traces, we record the trace of the case scrutinee and the taken branch. Traces can contain free variables and so we re-bind the variable in the trace of the taken branch. Similarly for an application we record the traces of the function subexpression and the argument subexpression, and also record the trace of the evaluation of the body of the function. Again, since the body trace can mention the function and argument names as free variables, we re-bind these variables.

We want to emphasize at this point that we do not necessarily expect that implementations routinely construct fully detailed traces along the above lines. Rather, the trace semantics is proposed here as a candidate for the most detailed form of provenance we will consider. Recording and compressing or filtering relevant information from traces in an efficient way is beyond the scope of this paper.

Trace Replay. We equip traces with a semantics that relates them to expressions. We write $\gamma, T \curvearrowright v$ for the *replay* relation that reruns a trace on an environment (possibly different from the one originally used to construct T). The rules for most trace forms are the same as the standard rules for evaluating the corresponding expression forms.

$\boxed{\gamma, T \curvearrowright v}$

$$\frac{\gamma, T \curvearrowright \texttt{inl}(v) \qquad \gamma[x_1 \mapsto v], T_1 \curvearrowright v_1}{\gamma, \texttt{case}\ T \triangleright_{\texttt{inl}} x_1.T_1 \curvearrowright v_1} \qquad \frac{\gamma, T \curvearrowright \texttt{inr}(v) \qquad \gamma[x_2 \mapsto v], T_2 \curvearrowright v_2}{\gamma, \texttt{case}\ T \triangleright_{\texttt{inr}} x_2.T_2 \curvearrowright v_2}$$

$$\frac{\gamma, T_1 \curvearrowright \langle \kappa, \gamma' \rangle \qquad \gamma, T_2 \curvearrowright v_2, T_2' \qquad \gamma'[f \mapsto \langle \kappa, \gamma' \rangle, x \mapsto v_2], T \curvearrowright v}{\gamma, (T_1\ T_2) \triangleright_\kappa f(x).T \curvearrowright v}$$

Fig. 3. Dynamic semantics of Core TML: selected rules for trace replay

Figure 3 shows the rules for replaying case and application traces. Essentially, these rules require that the same control flow branches are taken as in the original run. If the input environment is different enough that the same branches cannot be taken, then replay fails.

3.2 Basic Properties of Traces

In this section, we identify key properties of traces, including type safety, and the consistency and fidelity properties that characterize how traces record the evaluation of an expression.

Determinacy and Type Safety. We employ a standard type system for expressions, with straightforward extensions to handle traces. Determinacy of typechecking and type-safety can be established for expression evaluation and trace replay. Types do not play a significant role in the main technical results, however, so we elide the details.

Consistency and Fidelity. We say that a trace T is *consistent* with an environment γ if there exists v such that $\gamma, T \curvearrowright v$. Evaluation produces consistent traces, and replaying a trace on the same input yields the same value:

Theorem 1 (Consistency). *If* $\gamma, e \Downarrow v, T$ *then* $\gamma, T \curvearrowright v$.

Furthermore, the trace produced by evaluation is faithful to the original expression, in the sense that whenever the trace can be successfully replayed on a different input, the result (and its trace) is the same as what we would obtain by rerunning e from scratch, and the resulting trace is the same as well. We call this property *fidelity*.

Theorem 2 (Fidelity). *If* $\gamma, e \Downarrow v, T$ *and* $\gamma', T \curvearrowright v'$ *then* $\gamma', e \Downarrow v', T$.

4 Provenance Views and Trace Queries

4.1 Provenance Extraction

Many previous approaches to provenance can be viewed as performing a form of *annotation propagation*. The idea is to decorate the input with annotations (often, initially, unique identifiers) and propagate the annotations through the evaluation. For example, in where-provenance, annotations are optional tags that can be thought of as pointers showing where output data was copied from in the source [6,5]. Other techniques, such

$$
\begin{array}{rcl}
\mathsf{F}(x,\widehat{\gamma}) & = & \widehat{\gamma}(x) \\
\mathsf{F}(\mathtt{let}\ x = T_1\ \mathtt{in}\ T_2,\widehat{\gamma}) & = & \mathsf{F}(T_2,\widehat{\gamma}[x{\mapsto}\mathsf{F}(T_1,\widehat{\gamma})]) \\
\mathsf{F}(c,\widehat{\gamma}) & = & c^{\mathsf{F}_c} \\
\mathsf{F}(\oplus(T_1,\ldots,T_n),\widehat{\gamma}) & = & (\hat{\oplus}(v_1,\ldots,v_n))^{\mathsf{F}_\oplus(a_1,\ldots,a_n)} \\
 & \text{where} & v_i^{a_i} = \mathsf{F}(T_i,\widehat{\gamma}) \\
\mathsf{F}((T_1,T_2),\widehat{\gamma}) & = & (\mathsf{F}(T_1,\widehat{\gamma}),\mathsf{F}(T_2,\widehat{\gamma}))^{\perp} \\
\mathsf{F}(\mathtt{fst}(T),\widehat{\gamma}) & = & v_1^{\mathsf{F}_1(a,b)} \\
 & \text{where} & (v_1^b,\widehat{v}_2)^a = \mathsf{F}(T,\widehat{\gamma}) \\
\mathsf{F}(\mathtt{inl}(T),\widehat{\gamma}) & = & \mathtt{inl}(\mathsf{F}(T,\widehat{\gamma}))^{\perp} \\
\mathsf{F}((\mathtt{case}\ T) \triangleright_{\mathtt{inl}} x.T_1,\widehat{\gamma}) & = & v^{\mathsf{F}_L(a,b)} \\
 & \text{where} & \mathtt{inl}(\widehat{v})^a = \mathsf{F}(T,\widehat{\gamma}) \\
 & \text{and} & v^b = \mathsf{F}(T_1,\widehat{\gamma}[y{\mapsto}\widehat{v}]) \\
\mathsf{F}(\mathtt{fun}\ \kappa,\widehat{\gamma}) & = & \langle\kappa,\widehat{\gamma}\rangle^{\mathsf{F}_\kappa} \\
\mathsf{F}((T_1\ T_2) \triangleright_\kappa f(x).T,\widehat{\gamma}) & = & v^{\mathsf{F}_{\mathsf{app}}(a,b)} \\
 & \text{where} & \langle\kappa,\widehat{\gamma}'\rangle^a = \mathsf{F}(T_1,\widehat{\gamma}) \\
 & \text{and} & \widehat{v}_2 = \mathsf{F}(T_2,\widehat{\gamma}) \\
 & \text{and} & v^b = \mathsf{F}(T,\widehat{\gamma}'[f{\mapsto}\langle\kappa,\widehat{\gamma}'\rangle^a, x{\mapsto}\widehat{v}_2])
\end{array}
$$

Fig. 4. Generic extraction (selected rules)

as why-, how-, dependency, and workflow provenance, can also be defined in terms of annotation propagation [18,17,4,10].

Based on this observation, we define a provenance extraction framework in which values are decorated with annotations and extraction functions take traces and return annotated values that can be interpreted as useful provenance information. We apply this framework to specify several concrete annotation schemes and extraction functions.

Extraction framework. Let A be an arbitrary set of *annotations* a, which we usually assume includes a blank annotation $\perp$ and a countably infinite set of identifiers $\ell \in \mathit{Loc}$, called *locations*. We define A*-annotated values* $\widehat{v}$ (or just *annotated values*, when A is clear) using the following grammar:

$$
\begin{array}{l}
\widehat{v} ::= v^a \qquad \widehat{\gamma} ::= [x_1 \mapsto \widehat{v}_1, \ldots, x_n \mapsto \widehat{v}_n] \\
v ::= c \mid (\widehat{v}_1,\widehat{v}_2) \mid \mathtt{inl}(\widehat{v}) \mid \mathtt{inr}(\widehat{v}) \mid \langle\kappa,\widehat{\gamma}\rangle \mid \mathtt{roll}(\widehat{v})
\end{array}
$$

We write $\widehat{\gamma}$ for *annotated environments* mapping variables to annotated values. We define an erasure function $|\widehat{v}|$ that maps each annotated value to an ordinary value by erasing the annotations. Similarly, $|\widehat{\gamma}|$ is the ordinary environment obtained by erasing the annotations from the values of $\widehat{\gamma}$.

We will define a family of provenance extraction functions $\mathsf{F}(T,\widehat{\gamma})$ that take a trace T and an environment $\widehat{\gamma}$ and return an annotated value. Each such F can be specified by giving the following annotation-propagation functions:

$$
\begin{array}{c}
\mathsf{F}_c, \mathsf{F}_\kappa : A \\
\mathsf{F}_1, \mathsf{F}_2, \mathsf{F}_L, \mathsf{F}_R, \mathsf{F}_{\mathsf{app}}, \mathsf{F}_{\mathtt{unroll}} : A \times A \to A \\
\mathsf{F}_\oplus : A \times \cdots \times A \to A
\end{array}
$$

Each function shows how the annotations involved in the corresponding computational step propagate to the result. For example, $\mathsf{F}_1(a, b)$ gives the annotation on the result of a `fst`-projection, where a is the annotation on the pair and b is the annotation of the first element. Figure 4 shows how to propagate annotations through a trace given basic annotation-propagation functions.

Remark 1. The extraction framework hard-wires the behavior of certain operations such as `let`, `inl`(), `inr`(), and application. It would also be possible to extend the framework to provide to customize their behavior; however, this functionality is not needed by any of the forms of provenance in this paper, and it is not clear whether there are natural provenance models that require them.

Theorem 3. *Every generic provenance extraction function is compatible with replay: that is, for any $\widehat{\gamma}, T, v$, if $|\widehat{\gamma}|, T \curvearrowright v$ then $|\mathsf{F}(T, \widehat{\gamma})| = v$.*

Where-provenance. Where-provenance can be defined via an annotation-propagating semantics where annotations are either labels ℓ or the blank annotation $\bot$. We define the where-provenance semantics $\mathsf{W}(T, \widehat{\gamma})$ using the following annotation-propagation functions:

$$\begin{aligned} \mathsf{W}_c, \mathsf{W}_\kappa &= \bot \\ \mathsf{W}_1, \mathsf{W}_2, \mathsf{W}_L, \mathsf{W}_R, \mathsf{W}_{\mathtt{app}}, \mathsf{W}_{\mathtt{unroll}} &= \lambda(x, y).y \\ \mathsf{W}_\oplus &= \lambda(a_1, \ldots, a_n).\bot \end{aligned}$$

Essentially, these functions preserve the annotations of data that are copied, and annotate computed or constructed data with $\bot$. This semantics is similar to that in Buneman et al. [5] and previous treatments of where-provenance in databases, adapted to TML.

Theorem 4. *Suppose $|\widehat{\gamma}|, e \Downarrow v', T$. If an annotated value v^a appears in $\mathsf{W}(T, \widehat{\gamma})$ with annotation $a \neq \bot$, then v^a is an exact copy (including any nested annotations) of a part of $\widehat{\gamma}$.*

Expression provenance. To model expression provenance, we consider *expression annotations* t consisting of labels ℓ, blanks $\bot$, constants c, or primitive function applications $\oplus(t_1, \ldots, t_n)$. We define expression-provenance extraction $\mathsf{E}(T, \widehat{\gamma})$ in much the same way as W, with the following differences:

$$\mathsf{E}_c = c \qquad \mathsf{E}_\oplus(t_1, \ldots, t_n) = \oplus(t_1, \ldots, t_n)$$

It would also be straightforward to define a translation from traces to provenance graphs (for example, Open Provenance Model graphs [23]).

The correctness property for expression provenance states that the expression annotation correctly recomputes the value it annotates.

Theorem 5. *Suppose $|\widehat{\gamma}|, e \Downarrow v', T$, where each subvalue in $\widehat{\gamma}$ is annotated with a copy of itself. If an annotated value v^e appears in $\mathsf{E}(T, \widehat{\gamma})$ with $e \neq \bot$, then e is a closed expression evaluating to $|v|$.*

Dependency provenance. To extract dependency provenance, we will use annotations ϕ that are sets of source locations $\{\ell_1, \ldots, \ell_n\}$. Initial annotations consist of distinct singleton sets $\{\ell\}$. We define $\mathsf{D}(T, \widehat{\gamma})$ using the following propagation functions:

$$\begin{aligned} \mathsf{D}_c, \mathsf{D}_\kappa &= \emptyset \\ \mathsf{D}_1, \mathsf{D}_2, \mathsf{D}_L, \mathsf{D}_R, \mathsf{D}_{\mathtt{app}}, \mathsf{D}_{\mathtt{unroll}} &= \lambda(x, y).x \cup y \\ \mathsf{D}_\oplus &= \lambda(a_1, \ldots, a_n).a_1 \cup \cdots \cup a_n \end{aligned}$$

This semantics is based on the dynamic provenance tracking semantics given by Cheney et al. [9], generalized to TML.

This definition satisfies the *dependency-correctness* property introduced in [9]. This property requires an auxiliary relation $\approx_\ell$ that says that two annotated values are equal except possibly at parts labeled by ℓ, whose straightforward definition we omit. Then we can show:

Theorem 6. *Suppose* $|\widehat{\gamma}|, e \Downarrow v, T$ *and* $\widehat{\gamma}' \approx_\ell \widehat{\gamma}$ *and* $|\widehat{\gamma}'|, e \Downarrow v', T'$. *Then we have* $\mathsf{D}(T, \widehat{\gamma}) \approx_\ell \mathsf{D}(T', \widehat{\gamma}')$.

This says that the label of a value in the input propagates to all parts of the output where changing the value can have an impact on the result.

Path annotations. For annotations to be useful when the full input is unavailable, we consider annotations where the locations ℓ are *paths* that uniquely address parts of the input environment. We write $\mathsf{path}(\gamma)$ for the environment γ with each component annotated with the path to that component. More generally, we define $\mathsf{path}_p(\gamma)$ as $[x_1 := \mathsf{path}_{p.x}(\gamma(x_1)), \ldots, x_n := \mathsf{path}_{p.x_n}(\gamma(x_n))]$ where $\mathsf{path}_p(v)$ is defined as follows:

$$\begin{aligned} \mathsf{path}_p(c) &= c^p \\ \mathsf{path}_p((v_1, v_2)) &= (\mathsf{path}_{p.1}(v_1), \mathsf{path}_{p.2}(v_2))^p \\ \mathsf{path}_p(\mathtt{inl}(v)) &= \mathtt{inl}(\mathsf{path}_{p.1}(v))^p \\ \mathsf{path}_p(\mathtt{inr}(v)) &= \mathtt{inr}(\mathsf{path}_{p.1}(v))^p \\ \mathsf{path}_p(\langle \kappa, \gamma \rangle) &= \langle \kappa, \mathsf{path}_p(\gamma) \rangle^p \end{aligned}$$

For example, $\mathsf{path}([x = (1, 2), y = \mathtt{inl}(4)]) = [x = (1^{x.1}, 2^{x.2})^x, y = \mathtt{inl}(4^{y.\mathtt{inl}})^y]$.

4.2 Patterns, Partial Traces, and Trace Queries

We introduce patterns for values, environments and traces. The syntax of patterns (pattern environments) is similar to that of values (respectively environments), extended with special *holes*:

$$\begin{aligned} p &::= c \mid (p_1, p_2) \mid \mathtt{inl}(p) \mid \mathtt{inr}(p) \mid \mathtt{roll}(p) \mid \langle \kappa, \rho \rangle \mid \Diamond \mid \Box \\ \rho &::= [x_1 \mapsto p_1, \ldots, x_n \mapsto p_n] \end{aligned}$$

Patterns actually denote binary relations on values. The hole symbol $\Box$ denotes the total relation, while the exact-match symbol $\Diamond$ denotes the identity relation. (The $\Diamond$ pattern is used in backward disclosure slicing.)

$\boxed{v \approx_p v}$

$$\frac{}{v \approx_\Box v'} \qquad \frac{}{v \approx_\Diamond v} \qquad \frac{}{c \approx_c c} \qquad \frac{v_1 \approx_{p_1} v_1' \qquad v_2 \approx_{p_2} v_2'}{(v_1, v_2) \approx_{(p_1,p_2)} (v_1', v_2')}$$

$$\frac{v \approx_p v' \qquad C \in \{\texttt{inl}, \texttt{inr}, \texttt{roll}\}}{C(v) \approx_{C(p)} C(v')} \qquad \frac{\gamma \approx_\rho \gamma'}{\langle \kappa, \gamma \rangle \approx_{\langle \kappa, \rho \rangle} \langle \kappa, \gamma' \rangle}$$

$$\gamma \approx_\rho \gamma' \iff \forall x \in \mathrm{dom}(\rho).\, \gamma(x) \approx_{\rho(x)} \gamma'(x)$$

Fig. 5. Equality modulo patterns

$$\begin{aligned}
\Box \sqcup p = p \sqcup \Box &= p \qquad \Diamond \sqcup p = p \sqcup \Diamond = p[\Diamond/\Box] \\
(p_1, p_2) \sqcup (p_1', p_2') &= (p_1 \sqcup p_1', p_2 \sqcup p_2') \qquad c \sqcup c = c \\
C(p) \sqcup C(p') &= C(p \sqcup p') \qquad C \in \{\texttt{inl}, \texttt{inr}, \texttt{roll}\} \\
\langle \kappa, \rho \rangle \sqcup \langle \kappa, \rho' \rangle &= \langle \kappa, \rho \sqcup \rho' \rangle \\
(\rho \sqcup \rho')(x) &= \begin{cases} \rho(x) \sqcup \rho'(x) & x \in \mathrm{dom}(\rho) \cup \mathrm{dom}(\rho') \\ \rho(x) & x \in \mathrm{dom}(\rho) \backslash \mathrm{dom}(\rho') \\ \rho'(x) & x \in \mathrm{dom}(\rho') \backslash \mathrm{dom}(\rho) \end{cases}
\end{aligned}$$

Fig. 6. Least upper bounds of patterns and environments

We say that v matches v' modulo p (written $v \approx_p v'$) if v and v' match the structure of p, and are equal at corresponding positions denoted by $\Diamond$. Moreover, we say $p \sqsubseteq v$ if $v \approx_p v$, and we write $p \sqcup p'$ for the least upper bound (join) of two patterns. Rules defining $\approx_p$ and $\sqcup$ are given in Figures 5 and 6.

When $p \sqsubseteq v$, we write $v|_p$ for the pattern obtained by replacing all of the $\Diamond$-holes in p with the corresponding values in v. For example, $(1,2)|_{(\Diamond,\Box)} = (1, \Box)$. Similarly, we write $\gamma|_\rho$ for $[x_1 = \gamma(x_1)|_{\rho(x_1)}, \ldots, x_n = \gamma(x_n)|_{\rho(x_n)}]$.

We also consider partial traces, usually written S, which are trace expressions where some subexpressions have been replaced with $\Box$:

$$S ::= \cdots \mid \Box$$

As with patterns, we write $S \sqsubseteq T$ to indicate that T matches S, that is, S can be made equal to T by filling in some holes.

For the purpose of disclosure and obfuscation analysis, we will consider the "traces" to be triples (γ, T, v) where T is consistent with γ and v, that is, $\gamma, T \curvearrowright v$. We refer to such a triple as a consistent triple. Furthermore, to analyze forms of provenance based on annotation we will consider consistent annotated triples $(\hat{\gamma}, T, \hat{v})$ where $\hat{v} = \mathsf{F}(T, \hat{\gamma})$. We consider trace or provenance queries built out of partial values and partial traces. We will later also consider queries derived from different forms of provenance, based on annotated triples.

Definition 5. *1. Let $\phi(\gamma)$ be a predicate on input environments. An* input query $\mathsf{IN}\gamma.\phi(\gamma)$ *is defined as $\{(\gamma, T, v) \mid \gamma, T \curvearrowright v$ and $\phi(\gamma)\}$. As a special case, we write IN_ρ for $\mathsf{IN}\gamma.(\rho \sqsubseteq \gamma)$.*

2. *Let $\phi(v)$ be a predicate on output values. An* output query $\mathsf{OUT}v.\phi(v)$ *is defined as* $\{(\gamma, T, v) \mid \gamma, T \curvearrowright v \text{ and } \phi(v)\}$. *As a special case, we write* $\mathsf{OUT}_p = \mathsf{OUT}v.(p \sqsubseteq v)$.

5 Disclosure and Obfuscation Analysis

5.1 Disclosure

We first consider properties disclosed by various forms of provenance considered above. Both where-provenance and expression provenance disclose useful information about the input. Dependency provenance does not disclose input information in an easy-to-analyze way, but is useful for obfuscation, as discussed later.

For where-provenance, we consider input queries $Q_{v_0,\ell} = \mathsf{IN}\gamma.\ (\gamma.\ell = v_0)$ and output queries $Q'_{v_0,\ell} = \mathsf{OUT}v.\ (v.\ell = v_0)$, where ℓ is a path and v_0 is a value. Such a query tests whether the value at a given path in γ or v matches the provided value.

Theorem 7. *The provenance view* $(\gamma, T, v) \mapsto \mathsf{W}(T, \mathsf{path}(\gamma))$ *positively discloses* $Q_{v_0,\ell}$ *via* $Q'_{v_0,\ell}$.

For expression-provenance, let $\gamma(t)$ be the result of evaluating t in γ with all paths ℓ replaced by their values $\gamma.\ell$ in γ. We consider queries $Q_{t,v_0} = \mathsf{IN}\gamma.\ (\gamma(t) = v_0)$, where t is an expression provenance annotation and v_0 is a value. Such a query tests whether evaluating an expression e over γ yields the specified value. For example, $\mathsf{IN}\gamma.\ x.1 + y.2 = 4$ holds for $\gamma = [x = (1, 2), y = (2, 3)]$, because $\gamma(x.1) + \gamma(y.2) = 1 + 3 = 4$. We also consider output queries $Q'_{t,v_0} = \mathsf{OUT}v.\ (\hat{v_1}^{\,t}$ appears in v and $|\hat{v_1}| = v_0)$, that simply test whether an annotated copy of v_0 appears in the output with annotation t.

Theorem 8. *The provenance view* $(\gamma, T, v) \mapsto \mathsf{E}(T, \mathsf{path}(\gamma))$ *positively discloses* Q_{t,v_0} *via* Q'_{t,v_0}.

Expression provenance and where-provenance are also related in the following sense:

Theorem 9. *Where-provenance is computable from expression-provenance.*

Proof. Where-provenance annotations can be extracted from expression-provenance annotations by mapping locations ℓ to themselves and all other expressions to $\perp$.

Note that this implies that for a function like "factorial", the where-provenance of the output is always $\perp$. Hence, any query disclosed by where-provenance is disclosed by expression-provenance, and any query obfuscated by expression-provenance is also obfuscated by where-provenance.

We now consider a form of *trace slicing* that takes a partial output value and removes information from the input and trace that is not needed to disclose the output. We show that such *disclosure slices* also disclose generic provenance views (Theorem 11). Thus, disclosure slices form a quite general form of provenance in their own right.

Definition 6. *Let* $\gamma, T \Downarrow v$, *and suppose* $S \sqsubseteq T$ *and* $\rho \sqsubseteq \gamma$. *We say* (ρ, S) *is a* disclosure slice *with respect to partial value* p *if for all* $\gamma' \sqsupseteq \rho$ *and* $T' \sqsupseteq S$ *such that if* $\gamma', T' \curvearrowright v'$, *we have* $p \sqsubseteq v$ *iff* $p \sqsubseteq v'$.

$$\boxed{p, T \xrightarrow{\mathsf{disc}} S, \rho}$$

$$\frac{}{\Box, T \xrightarrow{\mathsf{disc}} \Box, []} \qquad \frac{}{p, x \xrightarrow{\mathsf{disc}} x, [x \mapsto p]} \qquad \frac{}{p, c \xrightarrow{\mathsf{disc}} c, []} \qquad \frac{}{\langle \kappa, \rho \rangle, \mathtt{fun}\ \kappa \xrightarrow{\mathsf{disc}} \mathtt{fun}\ \kappa, \rho}$$

$$\frac{}{\langle \kappa', \rho \rangle, \mathtt{fun}\ \kappa \xrightarrow{\mathsf{disc}} \mathtt{fun}\ \kappa', \Box} \qquad \frac{p_2, T_2 \xrightarrow{\mathsf{disc}} S_2, \rho_2[x \mapsto p_1] \qquad p_1, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1}{p_2, \mathtt{let}\ x = T_1\ \mathtt{in}\ T_2 \xrightarrow{\mathsf{disc}} \mathtt{let}\ x = S_1\ \mathtt{in}\ S_2, \rho_1 \sqcup \rho_2}$$

$$\frac{\Diamond, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1 \quad \cdots \quad \Diamond, T_n \xrightarrow{\mathsf{disc}} S_n, \rho_n}{p, \oplus(T_1, \ldots, T_n) \xrightarrow{\mathsf{disc}} \oplus(S_1, \ldots, S_n), \rho_1 \sqcup \cdots \sqcup \rho_n}$$

$$\frac{p_1, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1 \qquad p_2, T_2 \xrightarrow{\mathsf{disc}} S_2, \rho_2}{(p_1, p_2), (T_1, T_2) \xrightarrow{\mathsf{disc}} (S_1, S_2), \rho_1 \sqcup \rho_2} \qquad \frac{(p, \Box), T \xrightarrow{\mathsf{disc}} S, \rho}{p, \mathtt{fst}(T) \xrightarrow{\mathsf{disc}} \mathtt{fst}(S), \rho}$$

$$\frac{p, T \xrightarrow{\mathsf{disc}} S, \rho}{\mathtt{inl}(p), \mathtt{inl}(T) \xrightarrow{\mathsf{disc}} \mathtt{inl}(S), \rho} \qquad \frac{p, T \xrightarrow{\mathsf{disc}} S, \rho}{\mathtt{inl}(p), \mathtt{inr}(T) \xrightarrow{\mathsf{disc}} \mathtt{inr}(\Box), []}$$

$$\frac{p, T \xrightarrow{\mathsf{disc}} S, \rho}{\mathtt{roll}(p), \mathtt{roll}(T) \xrightarrow{\mathsf{disc}} \mathtt{roll}(S), \rho} \qquad \frac{\mathtt{roll}(p), T \xrightarrow{\mathsf{disc}} S, \rho}{p, \mathtt{unroll}(T) \xrightarrow{\mathsf{disc}} \mathtt{unroll}(S), \rho}$$

$$\frac{p_1, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1[x_1 \mapsto p] \qquad \mathtt{inl}(p), T \xrightarrow{\mathsf{disc}} S, \rho}{p_1, \mathtt{case}\ T \triangleright_{\mathtt{inl}} x_1.T_1 \xrightarrow{\mathsf{disc}} \mathtt{case}\ S \triangleright_{\mathtt{inl}} x_1.S_1, \rho \sqcup \rho_1}$$

$$\frac{p, T \xrightarrow{\mathsf{disc}} S, \rho[f \mapsto p_1, x \mapsto p_2] \qquad p_1 \sqcup \langle \kappa, \rho \rangle, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1 \qquad p_2, T_2 \xrightarrow{\mathsf{disc}} S_2, \rho_2}{p, (T_1\ T_2) \triangleright_\kappa f(x).T \xrightarrow{\mathsf{disc}} (S_1\ S_2) \triangleright_\kappa f(x).S, \rho_1 \sqcup \rho_2}$$

$$\frac{fvs(\kappa) = \{x_1, \ldots, x_n\}}{\Diamond, \mathtt{fun}\ \kappa \xrightarrow{\mathsf{disc}} \mathtt{fun}\ \kappa, [x_1 \mapsto \Diamond, \ldots, x_n \mapsto \Diamond]} \qquad \frac{\Diamond, T_1 \xrightarrow{\mathsf{disc}} S_1, \rho_1 \qquad \Diamond, T_2 \xrightarrow{\mathsf{disc}} S_2, \rho_2}{\Diamond, (T_1, T_2) \xrightarrow{\mathsf{disc}} (S_1, S_2), \rho_1 \sqcup \rho_2}$$

$$\frac{\Diamond, T \xrightarrow{\mathsf{disc}} S, \rho}{\Diamond, \mathtt{inl}(T) \xrightarrow{\mathsf{disc}} \mathtt{inl}(S), \rho} \qquad \frac{\Diamond, T \xrightarrow{\mathsf{disc}} S, \rho}{\Diamond, \mathtt{inr}(T) \xrightarrow{\mathsf{disc}} \mathtt{inr}(S), \rho} \qquad \frac{\Diamond, T \xrightarrow{\mathsf{disc}} S, \rho}{\Diamond, \mathtt{roll}(T) \xrightarrow{\mathsf{disc}} \mathtt{roll}(S), \rho}$$

Fig. 7. Disclosure slicing (selected rules)

Note that by this definition, minimal disclosure slices exist (since there are finitely many slices) but need not be unique. For example, both $\Box \vee \mathtt{true}$ and $\mathtt{true} \vee \Box$ are disclosure slices showing that $\mathtt{true} \vee \mathtt{true}$ evaluates to $\mathtt{true}$, but $\Box \vee \Box$ is not a disclosure slice.

Figure 7 shows selected rules defining a disclosure slicing judgment $p, T \xrightarrow{\mathsf{disc}} \rho, S$. Basically, the idea is to push a partial value backwards through a trace to obtain a partial input environment and trace slice. The partial input environment is needed to handle local variables in traces; for example, in the rule for `let`, we first slice through the body of the let, then identify the partial value showing the needed parts of the let-bound value, and use that to slice backwards through the first subtrace. Slicing for application traces is similar, but more complex due to the closure environments. Note also that the special $\Diamond$ patterns are used to slice backwards through primitive operations even when

we do not know the values of the inputs or results. (Another possibility is to annotate the traces of primitive operations with these values.)

Lemma 1. *If $\gamma, T \curvearrowright v$ then for any $p \sqsubseteq v$ there exists $S \sqsubseteq T$ and $\rho \sqsubseteq \gamma$ such that $p, T \xrightarrow{\mathsf{disc}} S, \rho$. Moreover, there is a unique least such S and ρ.*

We choose a function $\mathsf{Disc}_p(\gamma, T, v)$ on consistent triples (γ, T, v) whose value is $(\gamma|_\rho, S)$ where $p, T \xrightarrow{\mathsf{disc}} S, \rho$ and S, ρ are the least slices (obtained by determinizing the slicing algorithm by applying the first, hole-propagating rule greedily before any other rule). The idea is that we slice using the rules in Figure 7 and then transform ρ by filling in all $\Diamond$-holes with the corresponding values in γ. Note that the v parameter is irrelevant and is included only so that Disc_p is a uniform function from consistent triples to slices.

To prove the correctness of this disclosure slicing algorithm, we need a stronger notion of equivalence. Recall the definition of $v \approx_p v'$ as shown in Figure 5. Using this relation, we can prove the correctness of the slicing relation as follows:

Lemma 2. *Assume $\gamma, T \curvearrowright v$ and $p, T \xrightarrow{\mathsf{disc}} S, \rho$.*

1. *If $p \sqsubseteq v$ then for all $\gamma' \approx_\rho \gamma$ and $T' \sqsupseteq S$, if $\gamma', T' \Downarrow v'$ then $v' \approx_p v$.*
2. *If $p \not\sqsubseteq v$ then for all $\gamma' \approx_\rho \gamma$ and $T' \sqsupseteq S$, if $\gamma', T' \Downarrow v'$ then $p \not\sqsubseteq v'$.*

Proof. Both parts follow by induction on the structure of slicing derivations.

Correctness follows as a consequence of the above two properties.

Theorem 10. Disc_p *discloses* OUT_p.

Proof. Suppose $\mathsf{Disc}_p(\gamma, T, v) = \mathsf{Disc}_p(\gamma', T', v')$. Then $(\gamma|_\rho, S) = (\gamma'|_{\rho'}, S')$ where $p, T \xrightarrow{\mathsf{disc}} S, \rho$ and $p, T' \xrightarrow{\mathsf{disc}} S', \rho'$. Hence, $S = S'$ and $\gamma|_\rho = \gamma'|_{\rho'}$, which in turn implies $\gamma \approx_\rho \gamma'$. Suppose that $\mathsf{OUT}_p(\gamma, T, v)$ holds; that is, $p \sqsubseteq v$. Then by Lemma 2(1), $v' \approx_p v$ so $p \sqsubseteq v'$. Conversely, suppose $p \not\sqsubseteq v$. Then by Lemma 2(2), we have $p \not\sqsubseteq v'$.

Disclosure from slices. Finally, we link disclosure for value patterns to disclosure for generic provenance views. Essentially, we show that for any F, the disclosure slice for p positively discloses the F-provenance annotations of values matching p. Informally, this means that disclosure slices provide a highly general form of provenance specialized to a part of the output: one can compute and reveal the disclosure slice and others can then compute any generic provenance view from the slice, without rerunning the original computation or consulting input data or subtraces that are dropped in the slice.

Theorem 11. *Assume $|\widehat{\gamma}|, T \curvearrowright v$ and $p \sqsubseteq v$. Suppose $p, T \xrightarrow{\mathsf{disc}} S, \rho$. Suppose that F is a generic extraction function. Then the annotations associated with p in $\mathsf{F}(T, \widehat{\gamma})$ can be correctly extracted from S using only input parts needed by ρ. That is, suppose we have $\widehat{\gamma} \approx_\rho \widehat{\gamma}'$ (lifting $\approx_\_$ to annotated values in the obvious way) and $T' \sqsupseteq S$, where $|\widehat{\gamma}'|, T' \curvearrowright v'$. Then we have $\mathsf{F}(T, \widehat{\gamma}) \approx_p \mathsf{F}(T', \widehat{\gamma}')$.*

5.2 Obfuscation

We now consider obfuscating properties of traces. We first consider what can be obfuscated by the standard provenance views. Where-provenance, essentially, obfuscates anything that can never be copied to the output or affect the control flow of something that is copied to the output. Similarly, expression provenance obfuscates any part of the input that never participates in expression annotations. In both cases, we can potentially learn about parts of the input that affected control flow, however. For example, $\texttt{if}\ x = 1\ \texttt{then}\ 1\ \texttt{else}\ y$ does not obfuscate the value of x in either model, provided y comes from the input, since we can inspect the annotation of the result to determine that $x = 1$ or $x \neq 1$.

Given that we want to ensure obfuscation, we consider conservative techniques that accept (or construct) only provenance views that successfully obfuscate, but may reject some views or construct views that are unnecessarily opaque.

There are several ways to erase information from traces (or other provenance views) to ensure obfuscation of input properties. One way is to re-use the static analysis of dependency provenance (in [9], for example) to identify parts of the output that suffice to make it impossible to guess sensitive parts of the input. Alternatively we can use dynamic dependency provenance to increase precision, by propagating dependency tracking information from the input to the output.

This is similar to using static analysis or dynamic labels for information flow security; the difference is one of emphasis. In information flow security, we usually identify high- or low-security *locations* and try to certify that high-security data does not affect the computation of low-security data; here, instead, we identify a high-security *property* of the trace (e.g. that the input satisfies a certain formula) and try to determine what parts of the output do not depend on sensitive inputs, and hence can be safely included in the provenance view. However, these techniques do not provide guidance about what parts of the *trace* can be safely included in the provenance view.

Here, we develop an alternative approach based on traces. Consider a pattern $\rho \sqsubseteq \gamma$ that erases all information that is considered sensitive. We construct an *obfuscation slice* by re-evaluating T on ρ as much as possible, to compute a sliced trace S and partial value p. We erase parts of T and of the original output value that depend on the erased parts of ρ. Thus, any part of the trace or output value that remains in the obfuscation slice is irrelevant to the sensitive part of the input, and cannot be used to guess it.

Figure 8 shows a syntactic algorithm for computing obfuscation slices as described above. Many rules are essentially generalizations of the rules for evaluation to allow for partial inputs, outputs and traces. The rules of interest, near the bottom of the figure, show how to handle attempts to compute that encounter holes in places where a value constructor is expected. When this happens, we essentially propagate the hole result and return a hole trace. This may be unnecessarily aggressive for some cases, but is necessary for the case and application traces where the trace form gives clues about the control flow.

We define $\mathsf{Obf}_\rho(\gamma, T, v)$ as (p, S) where $\rho, T \xrightarrow{\mathsf{obf}} p, S$. We can show that this is total for well-formed, $\Diamond$-free traces and input environments.

Lemma 3. *If* $\gamma, T \curvearrowright v$ *and* $\rho \sqsubseteq \gamma$ *and* $\rho, T \xrightarrow{\mathsf{obf}} p, S$ *then* $p \sqsubseteq v$ *and* $S \sqsubseteq T$.

$\boxed{\rho, T \xrightarrow{\text{obf}} p, S}$

$$\frac{}{\rho, x \xrightarrow{\text{obf}} \rho(x), x} \qquad \frac{}{\rho, c \xrightarrow{\text{obf}} \rho(x), c} \qquad \frac{}{\rho, \texttt{fun}\ \kappa \xrightarrow{\text{obf}} \langle \kappa, \rho \rangle, \texttt{fun}\ \kappa}$$

$$\frac{\rho, T_1 \xrightarrow{\text{obf}} p_1, S_1 \qquad \rho[x \mapsto p_1], T_2 \xrightarrow{\text{obf}} p_2, S_2}{\rho, \texttt{let}\ x = T_1\ \texttt{in}\ T_2 \xrightarrow{\text{obf}} p_2, \texttt{let}\ x = S_1\ \texttt{in}\ S_2}$$

$$\frac{\rho, T_1 \xrightarrow{\text{obf}} v_1, S_1 \quad \cdots \quad \rho, T_n \xrightarrow{\text{obf}} v_n, S_n}{\rho, \oplus(T_1, \ldots, T_n) \xrightarrow{\text{obf}} \oplus(v_1, \ldots, v_n), \oplus(S_1, \ldots, S_n)}$$

$$\frac{\rho, T_1 \xrightarrow{\text{obf}} p_1, S_1 \qquad \rho, T_2 \xrightarrow{\text{obf}} p_2, S_2}{\rho, (T_1, T_2) \xrightarrow{\text{obf}} (p_1, p_2), (S_1, S_2)} \qquad \frac{\rho, T \xrightarrow{\text{obf}} (p_1, p_2), S}{\rho, \texttt{fst}(T) \xrightarrow{\text{obf}} p_1, \texttt{fst}(S)}$$

$$\frac{\rho, T \xrightarrow{\text{obf}} p, S}{\rho, \texttt{inl}(T) \xrightarrow{\text{obf}} \texttt{inl}(p), \texttt{inl}(S)} \qquad \frac{\rho, T \xrightarrow{\text{obf}} \texttt{inl}(p), S \qquad \rho[x_1 \mapsto p], T_1 \xrightarrow{\text{obf}} p_1, S_1}{\rho, \texttt{case}\ T \triangleright_{\texttt{inl}} x_1.T_1 \xrightarrow{\text{obf}} p_1, \texttt{case}\ S \triangleright_{\texttt{inl}} x_1.S_1}$$

$$\frac{\rho, T \xrightarrow{\text{obf}} p, S}{\rho, \texttt{roll}(T) \xrightarrow{\text{obf}} \texttt{roll}(p), \texttt{roll}(S)} \qquad \frac{\rho, T \xrightarrow{\text{obf}} \texttt{roll}(p), S}{\rho, \texttt{unroll}(T) \xrightarrow{\text{obf}} p, \texttt{unroll}(S)}$$

$$\frac{\rho, T_1 \xrightarrow{\text{obf}} \langle \kappa, \rho_0 \rangle, S_1 \qquad \rho, T_2 \xrightarrow{\text{obf}} p_2, S_2 \qquad \rho[f \mapsto \langle \kappa, \rho_0 \rangle, x \mapsto p_2], T \xrightarrow{\text{obf}} p, S}{\rho, (T_1\ T_2) \triangleright_\kappa f(x).T \xrightarrow{\text{obf}} p, (S_1\ S_2) \triangleright_\kappa f(x).S}$$

$$\frac{\rho, T_i \xrightarrow{\text{obf}} \square, S_i}{\rho, \oplus(T_1, \ldots, T_n) \xrightarrow{\text{obf}} \square, \square} \qquad \frac{\rho, T \xrightarrow{\text{obf}} \square, S}{\rho, \texttt{fst}(T) \xrightarrow{\text{obf}} \square, \square} \qquad \frac{\rho, T \xrightarrow{\text{obf}} \square, S}{\rho, \texttt{unroll}(T) \xrightarrow{\text{obf}} \square, \square}$$

$$\frac{\rho, T \xrightarrow{\text{obf}} \square, S}{\rho, \texttt{case}\ T \triangleright_{\texttt{inl}} x_1.T_1 \xrightarrow{\text{obf}} \square, \square} \qquad \frac{\rho, T_1 \xrightarrow{\text{obf}} \square, S_1}{\rho, (T_1\ T_2) \triangleright_\kappa f(x).T \xrightarrow{\text{obf}} \square, \square}$$

Fig. 8. Obfuscation slicing (selected rules)

Lemma 4. *If $\gamma, e \Downarrow v, T$ and $\rho \sqsubseteq \gamma$ and $\rho, T \xrightarrow{\text{obf}} p, S$ then for all $\gamma' \sqsupseteq \rho$, if $\gamma', e \Downarrow v', T'$ then $\rho, T' \xrightarrow{\text{obf}} p, S$.*

Theorem 12. *For traces generated by terminating expressions, and ρ with holes of nonsingular types, and $\rho' \sqsupset \rho$, we have Obf_ρ positively obfuscates $\mathsf{IN}_{\rho'}$.*

Proof. Suppose $\mathsf{IN}_{\rho'}$ holds of (γ, T, v) where $\rho' \sqsupset \rho$. Then $\rho \sqsubset \rho' \sqsubseteq \gamma$. Moreover, since the inclusion is strict, ρ must contain holes that can be replaced with different values, so there exists another $\gamma' \sqsupseteq \rho$ that differs from ρ'. Since T was generated by a terminating expression, we know that $\gamma', e \Downarrow v', T'$ can be derived for some v', T'. By Lemma 4 (and the easily-verified determinacy of $\xrightarrow{\text{obf}}$) we know that $\rho, T' \xrightarrow{\text{obf}} p, S$, hence $\mathsf{Obf}_\rho(\gamma', T', v') = (p, S) = \mathsf{Obf}_\rho(\gamma, T, v)$, as required.

5.3 Discussion

The analysis in section 5.1 gives novel characterizations of what information is disclosed by where-provenance and expression provenance. Essentially, where-provenance

discloses information about what parts of the input are copied to the output, while expression provenance additionally discloses information about how parts of the input can be combined to compute parts of the output. The analysis in section 5.1 also shows (in a formal sense) that where-provenance and expression provenance are closely related: one can obtain where-provenance from expressions simply by erasure. Moreover, we can obtain a number of intermediate provenance models based on transductions over expression-provenance annotations.

The disclosure slicing algorithm is based on an interesting insight (which we are exploring in concurrent work on slicing): at a technical level, the information we need for program comprehension via slicing (to understand how a program has evaluated its inputs to produce outputs) is quite similar to what we need for provenance. Although we have identified connections between provenance and slicing before [9], our disclosure and obfuscation slicing algorithms provide further evidence of this close connection.

6 Related Work

There is a huge, and growing, literature on provenance [3,10,24,22], but there is little work on formal models of provenance and none on provenance in a general-purpose higher-order language. Due to space limits, we confine our comparison to closely related work on formal techniques for provenance, and on related ideas in programming languages and language-based security. We refer the interested reader to the aforementioned surveys for more information on provenance in workflows and databases, and to [11,8] for further discussion of prior work on provenance security.

Provenance. This work differs from previous work on provenance in databases in several important ways. First, we consider a general purpose, higher-order language, whereas previous work considers database query languages of limited expressiveness (e.g., monotone query languages), which include unordered collection types with monadic iteration operations but not sum types, recursive types or first-class functions. Second, we aim to record traces adequate to answer a wide range of provenance queries in this general setting, whereas previous work has focused on particular kinds of queries (e.g., where-provenance [6,5], why-provenance [6], how-provenance [18,17]).

Provenance has also been studied extensively for scientific workflow systems [3,24,14]. Most work in this area describes the provenance tracking behavior of a system through examples and does not give a formal semantics that could be used to prove correctness properties. An exception is Hidders et al. [20], which is the closest workflow provenance work to ours. They model workflows using a core database query language extended with nondeterministic, external function calls, and partially formalize a semantics of *runs*, or sets of triples (γ, e, v) labeling an operational derivation tree.

Other related topics. Our trace model is partly inspired by previous work on self-adjusting computation [1], where execution traces are used to efficiently recompute functional programs under arbitrary modifications to their inputs. Provenance-like ideas have also appeared in the context of bidirectional computation [2]. Dimoulas et al. study correctness properties for *blame* in contracts based on semantic properties that, they

suggest, may be related to provenance [16]. However, to our knowledge no formal relationships between provenance and self-adjusting computation, bidirectional computation, or blame have been developed.

7 Conclusions

While the importance of understanding provenance and its security characteristics has been widely documented, to date there has been little work on formal modeling of either provenance or provenance security. In this paper, we elaborate upon the ideas introduced in previous work [8], by instantiating the formal framework proposed there with a general-purpose functional programming language and a natural notion of execution traces. We showed how more conventional forms of provenance can be extracted from such traces via a generic provenance extraction mechanism. Furthermore, we studied the key notions of disclosure and obfuscation in this context. In the process we identified weaker *positive* and *negative* variants of disclosure and obfuscation, based on the observation that the original definitions seem too strong to be satisfied often in practice. Our main results include algorithms for *disclosure slicing*, which traverses a trace backwards to retain information needed to certify how an output was produced, and *obfuscation slicing*, which reruns a trace on partial input (excluding sensitive parts of the input), yielding a partial trace and partial output that excludes all information that could help a principal learn sensitive data.

To summarize, our main contribution is the development of a general model of provenance in the form of a core calculus that instruments runs of programs with detailed execution traces. We validated the design of this calculus by showing that traces generalize other known forms of provenance and by studying their disclosure and obfuscation properties. There are many possible avenues for future work, including:

- identifying richer languages for defining trace queries or provenance views
- developing and implementing effective algorithms for trace slicing, and relating these to program slicing
- extending trace and provenance models to handle references, exceptions, input/output, concurrency, nondeterminism, communication, etc.

References

1. Acar, U.A., Blelloch, G.E., Harper, R.: Adaptive functional programming. ACM Trans. Program. Lang. Syst. 28(6), 990–1034 (2006)
2. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: resourceful lenses for string data. In: POPL, pp. 407–419. ACM, New York (2008)
3. Bose, R., Frew, J.: Lineage retrieval for scientific data processing: a survey. ACM Comput. Surv. 37(1), 1–28 (2005)
4. Buneman, P., Cheney, J., Tan, W.-C., Vansummeren, S.: Curated databases. In: PODS, pp. 1–12 (2008)
5. Buneman, P., Cheney, J., Vansummeren, S.: On the expressiveness of implicit provenance in query and update languages. ACM Transactions on Database Systems 33(4), 28 (2008)

6. Buneman, P., Khanna, S., Tan, W.-C.: Why and Where: A Characterization of Data Provenance. In: Van den Bussche, J., Vianu, V. (eds.) ICDT 2001. LNCS, vol. 1973, pp. 316–330. Springer, Heidelberg (2000)
7. Carey, S., Rogow, G.: UAL shares fall as old story surfaces online. Wall Street Journal (September 2008), http://online.wsj.com/article/SB122088673--738010213.html
8. Cheney, J.: A formal framework for provenance security. In: CSF, pp. 281–293. IEEE (2011)
9. Cheney, J., Ahmed, A., Acar, U.A.: Provenance as dependency analysis. Mathematical Structures in Computer Science 21(6), 1301–1337 (2011)
10. Cheney, J., Chiticariu, L., Tan, W.C.: Provenance in databases: Why, how, and where. Foundations and Trends in Databases 1(4), 379–474 (2009)
11. Cheney, J., Chong, S., Foster, N., Seltzer, M., Vansummeren, S.: Provenance: A future history. In: OOPSLA Companion (Onward! 2009), pp. 957–964 (2009)
12. Chong, S.: Towards semantics for provenance security. In: Workshop on the Theory and Practice of Provenance (2009), Informal online proceedings: http://www.usenix.org/events/tapp09/
13. Cirillo, A., Jagadeesan, R., Pitcher, C., Riely, J.: TAPIDO: Trust and Authorization Via Provenance and Integrity in Distributed Objects (Extended Abstract). In: Gairing, M. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 208–223. Springer, Heidelberg (2008)
14. Davidson, S.B., Freire, J.: Provenance and scientific workflows: challenges and opportunities. In: SIGMOD, New York, NY, USA, pp. 1345–1350 (2008)
15. Davidson, S.B., Khanna, S., Milo, T., Panigrahi, D., Roy, S.: Provenance views for module privacy. In: PODS, pp. 175–186 (2011)
16. Dimoulas, C., Findler, R.B., Flanagan, C., Felleisen, M.: Correct blame for contracts: no more scapegoating. In: POPL, pp. 215–226. ACM, New York (2011)
17. Foster, J.N., Green, T.J., Tannen, V.: Annotated XML: queries and provenance. In: PODS, pp. 271–280 (2008)
18. Green, T.J., Karvounarakis, G., Tannen, V.: Provenance semirings. In: PODS, pp. 31–40 (2007)
19. Guts, N., Fournet, C., Zappa Nardelli, F.: Reliable Evidence: Auditability by Typing. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 168–183. Springer, Heidelberg (2009)
20. Hidders, J., Kwasnikowska, N., Sroka, J., Tyszkiewicz, J., Van den Bussche, J.: A Formal Model of Dataflow Repositories. In: Cohen-Boulakia, S., Tannen, V. (eds.) DILS 2007. LNCS (LNBI), vol. 4544, pp. 105–121. Springer, Heidelberg (2007)
21. Jia, L., Vaughan, J.A., Mazurak, K., Zhao, J., Zarko, L., Schorr, J., Zdancewic, S.: Aura: a programming language for authorization and audit. In: ICFP, New York, NY, USA, pp. 27–38 (2008)
22. Moreau, L.: The foundations for provenance on the web. Foundations and Trends in Web Science 2(2-3) (2010)
23. Moreau, L., et al.: The open provenance model core specification (v1.1). Future Generation Computer Systems 27(6), 743–756 (2010)
24. Simmhan, Y., Plale, B., Gannon, D.: A survey of data provenance in e-science. SIGMOD Record 34(3), 31–36 (2005)
25. Swamy, N., Chen, J., Fournet, C., Strub, P.-Y., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. In: ICFP, pp. 266–278 (2011)
26. Swamy, N., Corcoran, B.J., Hicks, M.: Fable: A language for enforcing user-defined security policies. In: IEEE Symposium on Security and Privacy, pp. 369–383 (2008)
27. Varghese, S.: UK government gets bitten by Microsoft Word. Sydney Morning Herald (July 2003), http://www.smh.com.au/articles/2003/07/02/1056825430340.html