Precise, Dynamic Information Flow for Database-Backed Applications

Jean Yang
Carnegie Mellon University

Travis Hance
Dropbox

Thomas H. Austin
San Jose State University

Armando Solar-Lezama
Massachusetts Institute of Technology

Cormac Flanagan
University of California, Santa Cruz

Stephen Chong
Harvard University

Abstract

We present an approach for dynamic information flow control across the application and database. Our approach reduces the amount of policy code required, yields formal guarantees across the application and database, works with existing relational database implementations, and scales for realistic applications. In this paper, we present a programming model that factors out information flow policies from application code and database queries, a dynamic semantics for the underlying λ^{JDB} core language, and proofs of termination-insensitive non-interference and policy compliance for the semantics. We implement these ideas in Jacqueline, a Python web framework, and demonstrate feasibility through three application case studies: a course manager, a health record system, and a conference management system used to run an academic workshop. We show that in comparison to traditional applications with hand-coded policy checks, Jacqueline applications have 1) a smaller trusted computing base, 2) fewer lines of policy code, and 2) reasonable, often negligible, additional overheads.

1. Introduction

From social networks to electronic health record systems, programs increasingly process sensitive data. As information leaks often arise from programmer error, a promising way to reduce leaks is to reduce opportunities for programmer error.

A major challenge in securing web applications involves reasoning about the flow of sensitive data across the application and database. According to the OWASP report [42], errors frequently occur at component boundaries. Indeed, the difficulty of reasoning about how sensitive data flows through both application code and database queries has led to leaks in systems from the HotCRP conference management system [5] to the social networking site Facebook [46]. The patch for the recent HotCRP bug involves policy checks across application code and database queries.

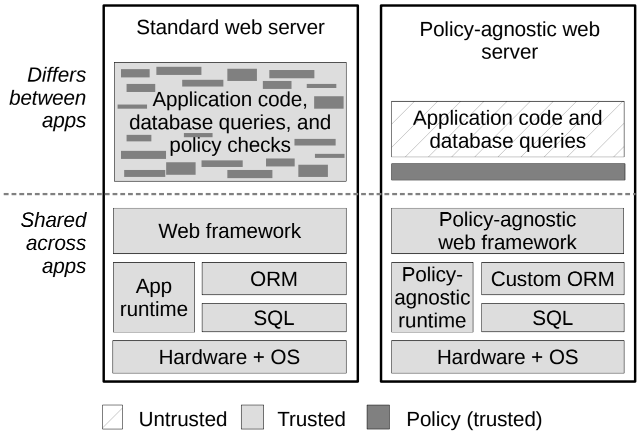


Figure 1. Application architecture in a standard web server compared to a policy-agnostic web server.

Information flow control is important to securing the application-database boundary [16, 19, 30, 42]. This is because leaks often involve the results of computations on sensitive values, rather than sensitive values themselves. To reduce the opportunity for inadvertent leaks, we present a *policy-agnostic* approach [8, 47]. Using this approach, the programmer factors out the implementation of information flow policies from application code and database queries. The system manages the policies, removing the need to trust the remaining code. The program thus specifies each policy once, rather than as repeated intertwined checks across the program. Because of this, policy-agnostic programs require less policy code. We illustrate these differences in Figure 1.

Supporting policy-agnostic programming for web applications requires the framework to enforce information flow policies across the application and database. As we also show in Figure 1, a standard web program runs using an application runtime and a database. An object-relational mapping (ORM) to mediate interactions between the two. Our web framework uses a policy-agnostic application runtime and a specialized ORM that mediates interactions between policy-agnostic application code and policy-agnostic database queries.

There are three main parts to our solution: 1) supporting policy-agnostic database queries, 2) providing formal guarantees across the application and database, and 3) addressing issues of practical feasibility. We extend prior work on the Jeeves programming language [8, 47] that defines a policy-agnostic semantics for a simple imperative language. As is common with language-based approaches, Jeeves's guarantees extend only within the Jeeves runtime. Interoperation with external databases is important as web applications rely on commodity databases for performance reasons. The challenge is, then, to support policy-agnostic programming for database queries in a way that leverages existing database implementations while providing strong guarantees.

We present *faceted databases* for supporting policy-agnostic database queries. The Jeeves runtime performs different computations based on the permissions of the user viewing the output. Because the viewer may not be known in advance, the runtime uses *faceted execution* to simulate simultaneous executions. A *faceted value* is the runtime representation of a value that may differ across executions. Semantically, a faceted database stores faceted values and performs faceted query execution. We show how to use a *faceted object-relational mapping* (FORM) to embed faceted values using relational databases and, surprisingly, to support faceted query execution simply by manipulating meta-data. The FORM manages complex dependencies, allowing a policy to query the data it protects.

Next we show that interoperation with faceted databases yields strong guarantees. We extend Jeeves's core language with relational operators to create the λ^{JDB} core language. We present a dynamic faceted execution semantics for λ^{JDB} and prove termination-insensitive non-interference and policy compliance. The formalization corresponds closely to an implementation strategy using existing database implementations while yielding concise proofs.

Towards supporting realistic applications, we formulate an "Early Pruning" optimization. While simulating multiple executions is desirable for reasoning, exploring multiple executions can be expensive in practice. The Early Pruning optimization allows the program to use program assumptions to safely explore fewer executions. This optimization is particularly useful for web applications, where it is often possible to use the session user to predict the viewer. With Early Pruning, performance may even be better than with hand-coded checks, as the runtime may now check policies once rather than repeatedly throughout execution.

Finally, we demonstrate practically feasibility. We present Jacqueline, a web framework based on Python's Django [3] framework. We use Jacqueline to build several application case studies, including a conference management system that we have deployed to run an academic workshop. The case studies show that using Jacqueline, policies are localized and the size of the policy code is smaller. Consequently, security audits can focus on the localized policy specifications rather than having to review the entire code base. We also demonstrate that Jacqueline has reasonable, often negligible, overheads. For one case, the Jacqueline implementation performs better than an implementation with hand-coded policies.

In summary, we make the following contributions:

- **Policy-agnostic web programming.** We present an approach that allows programmers to factor out information flow policies from the rest of web programs and rely on a web framework to dynamically enforce the policies.
- **Faceted databases.** We present faceted databases to support policy-agnostic relational database queries. We present a faceted object-relational mapping (FORM) strategy for implementing faceted databases using existing relational database implementations.
- **Faceted execution for database-backed applications.** We show interoperation of faceted databases with faceted application runtimes by presenting a dynamic semantics for the λ^{JDB} core language and proving termination-insensitive non-interference and policy compliance.
- **Early Pruning optimization.** We address performance issues with exploring multiple executions by formalizing the Early Pruning optimization, proving that it preserves policy compliance, and demonstrating that it significantly decreases overheads.
- **Demonstration of practical feasibility.** We present the Jacqueline web framework and demonstrate expressiveness and performance through several application case studies. We compare against hand-implemented policies, showing that not only does Jacqueline reduce lines of policy code, but also that policy enforcement has reasonable, often negligible, overheads.

Our approach decreases the opportunity for programmer error, provides strong formal guarantees, and is practically feasible.

2. Introductory Example

Consider a social calendar application. Suppose Alice and Bob want to plan a surprise party for Carol, 7pm next Tuesday at Schloss Dagstuhl. They should be able to create an event such that information is visible only to guests. Carol should see that she has an event 7pm next Tuesday, but not that it is a party. Everyone else may see that there is a private event at Schloss Dagstuhl, but not event details.

We demonstrate how to implement this example using Jacqueline, our new web framework based on Django [3], a *model-view-controller* framework. In a standard MVC framework, the *model* describes the data, the *view* describes frontend page rendering, and the *controller* implements other functionality. An object-relational mapping (ORM) supports a uni-

```
1  class Event(JModel):
2    name = CharField(max_length=256)
3    location = CharField(max_length=512)
4    time = DateTimeField()
5    description = CharField(max_length=1024)
6
7    # Public value for name field.
8    @staticmethod
9    def jacqueline_get_public_name(event):
10     return "Private event"
11
12   # Public value for location field.
13   @staticmethod
14   def jacqueline_get_public_location(event):
15     return "Undisclosed location"
16
17   # Policies for name and location fields.
18   @staticmethod
19   @label_for('name', 'location')
20   @jacqueline
21   def jacqueline_restrict_event(event, ctxt):
22     return (EventGuest.objects.get(
23       event=self, guest=ctxt) != None)
24
25 class EventGuest(JModel):
26   event = ForeignKey(Event)
27   guest = ForeignKey(UserProfile)
```

Figure 2. Jacqueline schema fragment for calendar events.

form object representation. In Jacqueline, the model additionally specifies information flow policies. The faceted object-relational mapping (FORM) additionally supports a uniform representation of sensitive values and policies. Jacqueline is policy-agnostic: other than the policies, a Jacqueline program looks like a policy-free Django program.

The division of labor between the programmer and the framework is as follows. The programmer associates information flow policies with fields in the data schema, codes within the subset of Python supported by our Jeeves library, and accesses the database only through the Jacqueline API. The framework tracks sensitive values and policies between the application and database to produce outputs that adhere to the policies. In our attack model, the user is untrusted and we assume the programmer is not malicious.

We intend for this example to explain the semantics of policy-agnostic web programming. We discuss issues of implementation and optimization issues in later sections.

2.1 Schemas and Policies in Jacqueline

In Jacqueline, programmers specify each information flow policy once, associated with the *data schema* in the model. We show a sample schema for the Event and EventGuest data objects in Figure 2. A Jacqueline schema defines field names, field types, and optional policies. We define the Event class with fields name, location, description, and visibility, where visibility is a user-specified setting corresponding to whether the event is visible to everyone or only to guests. Up to line 5, this looks like a standard Django schema definition.

2.1.1 Secret Values and Public Values

A sensitive value in Jacqueline encapsulates a secret (high-confidentiality) view available only to viewers with sufficient permissions and a public (low-confidentiality) view available to other viewers. Jacqueline allows sensitive values to behave as either the secret value or public value, depending on viewing context (*i.e.* the user viewing a page).

The actual field value is the secret view and the programmer must additionally define a method computing the public view. On line 9 we define the jacqueline_get_public_name method computing the public view of the name field. If the permissions prohibit a viewer from seeing the sensitive name field, then the name field will behave as "Private event" throughout all computations, including database queries. This function takes the current row object (event) as an argument, allowing public values to be computed using row fields. The Jacqueline ORM uses naming conventions (*i.e.* the jacqueline_get_public prefix) to find the appropriate methods to compute public views.

2.1.2 Specifying Policies

Information flow policies guard the flow of sensitive values. On line 21 we implement the policy for the fields name and location, as indicated by the label_for decorator. The policy is a method that takes two arguments, the current row object (event) and the viewer (ctxt) corresponding to the user looking at a page. Our policy queries the EventGuest table (line 25) to determine whether the viewer is associated with the event. Jacqueline enforces this policy with respect to the value of event at the time a value is created and the state of the system at the time of output.

2.2 Faceted Execution

Once the programmer associates policies with sensitive data fields, the rest of the program may be *policy-agnostic*. We call create in Jacqueline the same way as in Django:

```
carolParty = Event.objects.create(
    name = "Carol's surprise party"
  , location = "Schloss Dagstuhl", ...)
```

To manage the policies, the Jacqueline FORM creates faceted values for the sensitive fields. For the name fields, the framework creates the faceted value ⟨k ? "Carol's surprise party" : "Private event"⟩, where k is a fresh Boolean *label* guarding the secret actual field value and the public facet computed from the get_public_name method. The runtime eventually assigns label values based on policies and the viewer. We describe in Section 3 how the FORM stores faceted values in a relational database.

The runtime evaluates faceted values by evaluating each of the facets. Evaluating "Alice's events: " + str(alice.events) yields the resulting faceted value guarded by the same label:

```
⟨k ? "Alice's events: Carol's surprise party"
   : "Alice's events: Private event"⟩
```

Guests of the event will see "Carol's surprise party" as part of the list of Alice's events, while others will see only "Private event". Faceted execution propagates labels through all derived values, conditionals, and variable assignments to prevent indirect and implicit flows.

Jacqueline performs faceted execution for database queries, preventing indirect flows through queries like the following:

```
Event.objects.filter(
  location="Schloss Dagstuhl")
```

If carolParty is the only event in the database, faceted execution of the **filter** query yields a faceted list ⟨m ? [carolParty] : []⟩. Viewers who should not be able to see the location field will not be able to see values derived from the sensitive field.

Jacqueline also prevents implicit leaks through writes to the database. For instance, consider this code that replaces the description field of Event rows with "Dagstuhl event!" when the location field is "Schloss Dagstuhl":

```
for loc in Event.objects.all():
  if loc.location == "Schloss Dagstuhl":
    loc.description = "Dagstuhl event!"
    save(loc)
```

For carolParty the condition evaluates to ⟨k ? True : False⟩. The runtime records the influence of k when evaluating the conditional so that the call to save writes ⟨k ? carolPartyNew : carolParty⟩, where carolPartyNew is the updated value.

2.3 Computing Concrete Views

Computation sinks such as **print** take an additional argument corresponding to the viewer and resolves policies according to the viewer and policies. For instance, **print** carolParty.name displays "Carol's surprise party" to some viewers and "Private event" to others. The programmer does not need to designate the viewer: it can be an implicit parameter set from authorization information.

The policies and viewer define a system of constraints for determining label values. Printing carolParty.name to alice corresponding to the following constraint:

```
k ⇒
  (EventGuest.objects.get(
    event=self, guest=ctxt) != None)
```

To account for dependencies on mutable state, the runtime evaluates this constraint in terms of the guest list at the time of output. Labels are the only free variables in the fully evaluated constraints. There is always a consistent assignment to the labels: assigning all labels to False is always valid.

The constraint semantics allows Jacqueline to handle mutual dependencies between policies and sensitive values. Suppose that the guest list policy depended on the list itself:

```
@label_for('guest')
def jacqueline_restrict_guest(eventguest, ctxt):
  return (EventGuest.objects.get(
    event=eventguest.e, guest=ctxt) != None)
```

The policy requires that there must be an entry in the EventGuest table where the guest field is the viewer ctxt, so the policy for the guest field depends on the value of the field itself. There are two valid outcomes for a viewer who has access: either the system shows empty fields or the system shows the actual fields. Jacqueline always attempts to show values unless policies require otherwise. Note that unless there are mutual dependencies, Jacqueline may determine label values by evaluating policies directly.

Such circular dependencies are increasingly common in real-world applications. Consider, for instance, the following policies: a viewer must be within some radius of a secret location to see the location; a viewer must be a member of a secret list to see the list. Unfortunately, it is common practice to execute such policies in a trusted "omniscient" context that risks leaking information.

3. The Faceted Object-Relational Mapping

Our faceted object-relational mapping (FORM) 1) uses meta-data to represent faceted values and 2) manages queries by manipulating meta-data and marshalling to and from the database representation. Surprisingly, our solution allows us to use existing relational database implementations for creating, updating, selecting, joining, and sorting records. In this section, we introduce the faceted object-relational mapping (FORM) using SQL syntax and present the Early Pruning optimization.

3.1 Executing Relational Queries with Facets

A *faceted row* is a faceted value containing leaves that are non-faceted relational records. Any record containing faceted values may be rewritten to be of this form. We map each faceted row to multiple database rows by augmenting records with meta-data columns corresponding to 1) a unique identifier jid and 2) an identifier jvars describing which facet the row corresponds to, for instance "k1=True,k2=True".

The FORM is responsible for marshalling between the database and runtime representations of faceted values. The FORM stores the faceted value ⟨k ? "Carol's surprise party" : "Private event"⟩ as two rows in the Event table with the same jid of 1. The secret facet has a jvars value of "k=True" and the public facet has a jvars value of "k=False". For nested facets, we store more labels in the jvars column, for instance "k1=True,k2=True". In Table 1 we show how this faceted value would look in an augmented table.

3.1.1 Queries That Track Sensitive Values

A key advantage of our representation is that the FORM can issue standard relational queries not only for selections and projections, but also joins and sorts. Storing each facet in a different row allows the FORM to rely on the correct marshalling of query results for preventing indirect flows through queries. Note that the FORM would not be able to issue relational queries in such a straightforward way, for instance, if it stored each faceted value in the same row, or if it stored different facets in different databases.

id	name	location	jid	jvars
1	"Carol's ... party"	"Schloss Dagstuhl"	1	"x=True"
2	"Private event"	"Undisclosed location"	1	"x=False"

Table 1. Example table.

Django Query	Jacqueline Query
EventGuest.objects.**filter**(guest__name="Alice")	

SELECT EventGuest.event, EventGuest.guest
FROM EventGuest
JOIN UserProfile
ON EventGuest.guest_id = UserProfile.id
WHERE UserProfile.name='Alice';
	SELECT EventGuest.event, EventGuest.guest,
	EventGuest.jid, EventGuest.jvars,
	UserProfile.jvars
	FROM EventGuest
	JOIN UserProfile
	ON EventGuest.guest_id = UserProfile.jid
	WHERE UserProfile.name='Alice';

Table 2. Translated ORM queries in Django vs. Jacqueline.

Consider the query **SELECT** $*$ **from** Event **WHERE** location = "Schloss Dagstuhl" on the rows from Table 1. Issuing the query directly on the augmented database will return the one matching row with jid=1 and jvars="k=True". Reconstructing the facet structure yields a faceted value guarded by label k with a collection containing the record in the secret facet and an empty collection in the other facet. Relying on unmarshalling is sufficient for faceted execution.

Surprisingly, rows from joins that occur based on sensitive values will also be appropriately guarded by the appropriate path conditions. The only additional considerations the FORM needs to make for joins are to 1) take into account the jvars fields from both tables and 2) ensure that foreign keys (references into another table) use jid rather than the primary key. In Table 2, we show an example where the **WHERE** clause filters on the results of a **JOIN**. In the **ON** clause, we use the jid rather than id. In the **SELECT** clause, we include the User.jvars as well as the EventGuest.jvars field.

A particularly nice consequence of storing each facet in different rows is that the FORM can take advantage of SQL's **ORDER BY** functionality for sorting. Suppose we had faceted records, each with a single field f, with values ⟨a ? "Charlie" : "$***$"⟩, ⟨b ? "Bob" : "$***$"⟩, and ⟨c ? "Alice" : "$***$"⟩. The FORM can use the standard sorting procedure without leaking information because the secret values are stored in different rows from the public values. Correct unmarshalling will enforce the policies so that, for instance, an output context with the permitted labels $\{a, \neg b, c\}$ would see ["$***$", "Alice", "Charlie"].

A limitation is that the FORM cannot use existing relational implementations for aggregation, for instance counting or summing. Using aggregate queries directly could leak information because without looking at the path conditions, these aggregates would combine values across facets. This does not suggest a fundamental limitation. Applications often prematerialize aggregates, making it reasonable to use the faceted runtime to precompute aggregates. Otherwise, supporting faceted aggregation at scale is a matter of optimizing the procedures, perhaps as database user-defined functions.

3.1.2 Creating and Updating Data and Policies

The FORM creates tables and rows with the appropriate meta-data to keep track of facets. The FORM prevents implicit leaks through updates by updating meta-data appropriately and potentially deleting rows. Invoking save in branches that depend on faceted values creates facets that incorporate the path conditions. To add policies, the programmer needs to manipulate only the meta-data columns (jvars and jid). Adding policies to legacy data involves adding meta-data columns. Updating policies using existing labels simply involves updating policy code.

3.2 Early Pruning Optimization

An important correctness-preserving optimization is to prune facets once the runtime knows the viewer. This involves being able to determine 1) the viewing context and 2) that policy-relevant state relevant will not change before output. Two properties of web programs make this analysis simple. First, the session user is often the viewing context. Second, computation sinks are easy to identify in model-view-controller frameworks: most functions either read from the database or write to the database, but not both. This makes it advantageous for the framework to speculate on the viewer for "get" requests. We formalize Early Pruning in Section 4.4.

3.3 Data Representation Considerations

It is also important to discuss whether storing faceted values in the database may be prohibitively expensive. There are many ways to avoid storing too much data in practice. Work on multi-level databases [22, 31] suggests it is both useful and practically feasible to store multiple versions of data corresponding to different access levels. The question becomes, then, how to avoid storing too much data due to too many possible path conditions. An important optimization

$e ::=$		*Term*
	x	variable
	c	constant
	$\lambda x.e$	abstraction
	$e_1\ e_2$	application
	$\text{ref } e$	reference allocation
	$!e$	dereference
	$e_1{:=}e_2$	assignment
	$\langle k\ ?\ e_H : e_L\rangle$	faceted expression
	$\text{label } k \text{ in } e$	label declaration
	$\text{restrict}(k,e)$	policy specification
	$\text{row } \overline{e}$	create a table
	$\sigma_{i=j}\ e$	select rows where $i = j$
	$\pi_{\overline{i}}\ e$	project columns
	$e_1 \bowtie e_2$	join or cross-product of tables
	$e_1 \cup e_2$	union of tables
	$\text{fold } e_f\ e_p\ e_t$	table fold
$S ::=$		*Statement*
	$\text{let } x = e \text{ in } S$	let statement
	$\text{print } \{e_v\}\ e_r$	print statement
$c ::=$		*Constant*
	f	file handle
	b	boolean
	i	integer
	s	string
x,y,z		*Variable*
k,l		*Label*

Figure 3. λ^{JDB} syntax.

involves combining values that are the same to a single view. In Section 4, we define an optimization to allow sharing rows that different facets have in common.

4. Formal Semantics and Policy Compliance

We model the faceted object relational mapping with the idealized core language called λ^{JDB}. We prove that λ^{JDB} satisfies termination-insensitive non-interference and policy compliance across the application and database.

4.1 Syntax and Formal Semantics

The language λ^{JDB} extends the language λ^{jeeves} [8] with support for databases, which we model as relational tables. Figure 3 summarizes the λ^{JDB} syntax, with the constructs from λ^{jeeves} marked in gray. The λ^{jeeves} language, in turn, extends the standard imperative λ-calculus with constructs for declaring new labels (label k in e), for imperatively attaching policies to labels (restrict(k,e)), and for creating faceted values ($\langle k\ ?\ e_H : e_L\rangle$). This last expression behaves like e_H from the perspective of any principal authorized to see data with label k and e_L for all other principals.

The language λ^{JDB} extends λ^{jeeves} with support for databases, where each table is a (possibly empty) sequence of rows and each row is a sequence of strings. We require that all rows in a table have the same size. To manipulate tables, λ^{JDB} includes the usual operators of the relational calculus: *selection* ($\sigma_{i=j}\ e$), which selects the rows in a table where fields i and j are identical, *projection* ($\pi_{\overline{i}}\ e$), which returns a new table containing columns $\overline{i}$ from the table e, *cross-product* ($e_1 \bowtie e_2$), which returns all possible combinations of rows from e_1 and e_2, and *union* ($e_1 \cup e_2$), which appends two tables. The construct row $\overline{e}$ creates a new single-row table. The fold operation fold $e_f\ e_p\ e_t$ supports iterating, or folding, over tables. Fold has the "type" $\forall\overline{A},B.(B \to \overline{A} \to B) \to B \to \text{table } \overline{A} \to B$.

4.2 Formal Semantics

We formalize the big-step semantics as the relation $\Sigma, e \Downarrow_{pc} \Sigma', V$, denoting that expression e and store Σ evaluate to V, producing a new store Σ'. The program counter pc is a set of *branches*. Each branch is either a label k or a negated label $\neg k$. Association with k means the computation is visible only to principals authorized to see k and association with $\neg k$ visibility only to principals *not* authorized to see k.

We chose our representation of faceted databases to be faithful to realistic implementation strategies. We could represent faceted tables as $\langle k\ ?\ \text{table } T_1 : \text{table } T_2\rangle$, but this approach would incur significant space overhead, as it requires storing two copies of possibly large database tables, possibly with only small differences between the two tables. Instead, we use the more efficient approach of *faceted rows*, where each row $(B,\overline{s})$ in the database includes a set of branches B describing who can see that row. For example, the expression $\langle k\ ?\ \text{row "Alice" "Smith"} : \text{row "Bob" "Jones"}\rangle$ evaluates to the following table [1]:

$$(\{k\}, (\text{"Alice"}, \text{"Smith"}))$$
$$(\{\neg k\}, (\text{"Bob"}, \text{"Jones"}))$$

Note that we do not model the facet identifier row jid, as it is not necessary for the formal semantics or proof.

To accommodate both faceted values and faceted tables, we define the partial operation $\langle\!\langle \cdot\ ?\ \cdot : \cdot \rangle\!\rangle$ to create either a new faceted value or a table with internal branches on rows:

$$\begin{array}{lcl}
\langle\!\langle \cdot\ ?\ \cdot : \cdot \rangle\!\rangle & : & \mathit{Label} \times \mathit{Val} \times \mathit{Val} \to \mathit{Val} \\
\langle\!\langle k\ ?\ F_H : F_L \rangle\!\rangle & \stackrel{\text{def}}{=} & \langle k\ ?\ F_H : F_L \rangle \\
\langle\!\langle k\ ?\ \text{table } T_H : \text{table } T_L \rangle\!\rangle & \stackrel{\text{def}}{=} & \text{table } T \\
\multicolumn{3}{l}{\text{where } T = \{(B,\overline{s}) \mid (B,\overline{s}) \in T_H \cap T_L\} \cup} \\
\multicolumn{3}{l}{\qquad\qquad \{(B \cup \{k\},\overline{s}) \mid (B,\overline{s}) \in T_H \setminus T_L, \neg k \notin B\} \cup} \\
\multicolumn{3}{l}{\qquad\qquad \{(B \cup \{\neg k\},\overline{s}) | (B,\overline{s}) \in T_L \setminus T_H, k \notin B\}}
\end{array}$$

[1] Note that this value representation does not support mixed expressions such as $\langle k\ ?\ 3 : \text{row "Alice"}\rangle$, which mix integers and tables in the same faceted values. Programs that try to unnaturally mix values will get stuck.

Figure 4. Faceted evaluation of λ^{JDB} without relational operators.

Runtime Syntax

$$
\begin{array}{rclcl}
e & \in & \mathit{Expr} & ::= & \ldots \mid a \mid \text{table } T \\
\Sigma & \in & \mathit{Store} & = & (\mathit{Addr} \rightarrow_p \mathit{Val}) \cup (\mathit{Label} \rightarrow \mathit{Val}) \\
R & \in & \mathit{RawValue} & ::= & c \mid a \mid (\lambda x.e) \\
a & \in & \mathit{Address} & & \\
F & \in & \mathit{FacetedValue} & ::= & R \mid \langle k \; ? \; F_1 : F_2 \rangle \\
T & \in & \mathit{Table} & = & (\mathit{Branches} \times \mathit{String}^n)^* \\
V & \in & \mathit{Val} & ::= & F \mid \text{table } T \\
b & \in & \mathit{Branch} & ::= & k \mid \neg k \\
pc, B & \in & \mathit{Branches} & ::= & b^*
\end{array}
$$

Evaluation Contexts

$$
\begin{array}{rcl}
E & ::= & \langle k \; ? \; E : e \rangle \mid \langle k \; ? \; V : E \rangle \\
 & \mid & \bullet \; e \mid v \; \bullet \mid \text{ref } \bullet \mid ! \bullet \mid \bullet := e \\
 & \mid & V := \bullet \mid \text{row } V \ldots \bullet \; e \ldots \mid \sigma_{i=j} \; \bullet \mid \pi_{\bar{i}} \; \bullet \\
 & \mid & \bullet \bowtie e \mid V \bowtie \bullet \mid \bullet \cup e \mid V \cup \bullet \\
 & \mid & \text{fold } \bullet \; e \; e \mid \text{fold } V \; \bullet \; e \mid \text{fold } V \; V \; \bullet
\end{array}
$$

Strict Contexts

$$
\begin{array}{rcl}
S & ::= & \bullet \; e \mid ! \bullet \mid \bullet := V \mid \sigma_{i=j} \; \bullet \mid \pi_i \; \bullet \\
 & \mid & \bullet \bowtie V \mid \text{table } T \bowtie \bullet \mid \bullet \cup V \\
 & \mid & \text{table } T \cup \bullet \mid \text{row } V \ldots \bullet \; e \ldots \\
 & \mid & \text{fold } V \; V \; \bullet
\end{array}
$$

Expression Evaluation Rules for λ^{jeeves} Subset $\boxed{\Sigma, e \Downarrow_{pc} \Sigma', V}$

$$\frac{}{\Sigma, V \Downarrow_{pc} \Sigma, V} \quad [\textsc{f-val}]$$

$$\frac{a \notin dom(\Sigma) \qquad \Sigma' = \Sigma[a := \langle\langle \, pc \; ? \; V : 0 \, \rangle\rangle]}{\Sigma, \text{ref } V \Downarrow_{pc} \Sigma', a} \quad [\textsc{f-ref}]$$

$$\frac{a \notin dom(\Sigma)}{\Sigma, !a \Downarrow_{pc} \Sigma, 0} \quad [\textsc{f-deref-null}]$$

$$\frac{}{\Sigma, !a \Downarrow_{pc} \Sigma, \Sigma(a)} \quad [\textsc{f-deref}]$$

$$\frac{\Sigma' = \Sigma[a := \langle\langle \, pc \; ? \; V : \Sigma(a) \, \rangle\rangle]}{\Sigma, a := V \Downarrow_{pc} \Sigma', V} \quad [\textsc{f-assign}]$$

$$\frac{\begin{array}{c} E \neq [\,] \qquad e \text{ not a value} \\ \Sigma, e \Downarrow_{pc} \Sigma', V' \qquad \Sigma', E[V'] \Downarrow_{pc} \Sigma'', V'' \end{array}}{\Sigma, E[e] \Downarrow_{pc} \Sigma'', V''} \quad [\textsc{f-ctxt}]$$

$$\frac{\Sigma, e[x := V] \Downarrow_{pc} \Sigma', V'}{\Sigma, (\lambda x.e) \; V \Downarrow_{pc} \Sigma', V'} \quad [\textsc{f-app}]$$

$$\frac{\begin{array}{c} k \notin pc \qquad \neg k \notin pc \\ \Sigma, e_1 \Downarrow_{pc \cup \{k\}} \Sigma_1, V_1 \\ \Sigma_1, e_2 \Downarrow_{pc \cup \{\neg k\}} \Sigma', V_2 \\ V' = \langle\langle \, k \; ? \; V_1 : V_2 \, \rangle\rangle \end{array}}{\Sigma, \langle k \; ? \; e_1 : e_2 \rangle \Downarrow_{pc} \Sigma', V'} \quad [\textsc{f-split}]$$

$$\frac{k \in pc \qquad \Sigma, e_1 \Downarrow_{pc} \Sigma', V}{\Sigma, \langle k \; ? \; e_1 : e_2 \rangle \Downarrow_{pc} \Sigma', V} \quad [\textsc{f-left}]$$

$$\frac{\neg k \in pc \qquad \Sigma, e_2 \Downarrow_{pc} \Sigma', V}{\Sigma, \langle k \; ? \; e_1 : e_2 \rangle \Downarrow_{pc} \Sigma', V} \quad [\textsc{f-right}]$$

$$\frac{\Sigma, \langle k \; ? \; S[V_H] : S[V_L] \rangle \Downarrow_{pc} \Sigma', V'}{\Sigma, S[\langle k \; ? \; V_H : V_L \rangle] \Downarrow_{pc} \Sigma', V'} \quad [\textsc{f-strict}]$$

Wrapping a facet with label k around non-table values F_H and F_L simply creates a faceted value containing k, F_H, and F_L. Wrapping a facet with label k around tables T_H and T_L creates a new table T containing the rows from T_H and T_L, annotated with k and $\neg k$ respectively, with an optimization to share the rows that T_H and T_L have in common. We extend this operator to sets of branches:

$$
\begin{array}{lcl}
\langle\langle \cdot \; ? \; \cdot : \cdot \rangle\rangle & : & \mathit{Branches} \times \mathit{Val} \times \mathit{Val} \rightarrow \mathit{Val} \\
\langle\langle \, \emptyset \; ? \; V_H : V_L \, \rangle\rangle & \overset{\text{def}}{=} & V_H \\
\langle\langle \, \{k\} \cup B \; ? \; V_H : V_L \, \rangle\rangle & \overset{\text{def}}{=} & \langle\langle \, k \; ? \; \langle\langle \, B \; ? \; V_H : V_L \, \rangle\rangle : V_L \, \rangle\rangle \\
\langle\langle \, \{\neg k\} \cup B \; ? \; V_H : V_L \, \rangle\rangle & \overset{\text{def}}{=} & \langle\langle \, k \; ? \; V_L : \langle\langle \, B \; ? \; V_H : V_L \, \rangle\rangle \, \rangle\rangle
\end{array}
$$

We show the faceted evaluation rules in Figures 4 and 5. The key rule is [F-SPLIT], describing how evaluation of a faceted expression $\langle k \; ? \; e_1 : e_2 \rangle$ involves evaluating the subexpressions in sequence. Evaluation adds k to the program counter to evaluate e_1 and $\neg k$ to evaluate e_2 and then joins the results in the operation $\langle\langle \, k \; ? \; V_1 : V_2 \, \rangle\rangle$. The rules [F-LEFT] and [F-RIGHT] show that only one expression is evaluated if the program counter already contains either k or $\neg k$.

Our rules use contexts to describe faceted execution. The rule [F-CTXT] for $E[e]$ enables evaluation of a subexpression inside an evaluation context. We use S to range over strict operator contexts, operations that require a non-faceted value. If an expression in a strict context yields a faceted value $\langle k \; ? \; V_H : V_L \rangle$, then the rule [F-STRICT] applies the strict operator to each of V_H and V_L. For example, the evaluation of $\langle k \; ? \; f : g \rangle(4)$ reduces to the evaluation of $\langle k \; ? \; f(4) : g(4) \rangle$, where S in this case is $\bullet \; (4)$. The rules [F-SELECT],

Figure 5. Faceted evaluation with relational operators.

$$\frac{}{\Sigma, \text{row } \bar{s} \Downarrow_{pc} \Sigma, (\text{table } (\epsilon, \bar{s}))} \quad [\text{F-ROW}] \qquad \frac{}{\Sigma, (\text{table } T_1) \cup (\text{table } T_2) \Downarrow_{pc} \Sigma, (\text{table } T_1.T_2)} \quad [\text{F-UNION}]$$

$$\frac{T' = \{(B, s_1 \ldots s_n) \in T \mid s_i = s_j\}}{\Sigma, \sigma_{i=j}\ (\text{table } T) \Downarrow_{pc} \Sigma, (\text{table } T')} \quad [\text{F-SELECT}] \qquad \frac{\bar{i} = i_1 \ldots i_n \qquad T' = \{(B, s_{i_1} \ldots s_{i_n}) \mid (B, s_1 \ldots s_m) \in T\}}{\Sigma, \pi_{\bar{i}}\ (\text{table } T) \Downarrow_{pc} \Sigma, (\text{table } T')} \quad [\text{F-PROJECT}]$$

$$\frac{T_3 = \{(B_1 \cup B_2, s_1 \ldots s_m s'_1 \ldots s'_n) \mid (B_1, s_1 \ldots s_m) \in T_1, (B_2, s'_1 \ldots s'_n) \in T_2\}}{\Sigma, (\text{table } T_1) \bowtie (\text{table } T_2) \Downarrow_{pc} \Sigma, (\text{table } T_3)} \quad [\text{F-JOIN}]$$

$$\frac{}{\Sigma, \text{fold } V_f\ V_p\ (\text{table } \epsilon) \Downarrow_{pc} \Sigma, V_p} \quad [\text{F-FOLD-EMPTY}]$$

$$\frac{\Sigma, \text{fold } V_f\ V_p\ (\text{table } T) \Downarrow_{pc} \Sigma', V' \qquad B \text{ inconsistent with } pc}{\Sigma, \text{fold } V_f\ V_p\ (\text{table } (B, \bar{s}).T) \Downarrow_{pc} \Sigma', V'} \quad [\text{F-FOLD-INCONSISTENT}]$$

$$\frac{\Sigma, \text{fold } V_f\ V_p\ (\text{table } T) \Downarrow_{pc} \Sigma', V' \qquad B \text{ consistent with } pc \qquad \Sigma', V_f\ \bar{s}\ V' \Downarrow_{pc \cup B} \Sigma'', V''}{\Sigma, \text{fold } V_f\ V_p\ (\text{table } (B, \bar{s}).T) \Downarrow_{pc} \Sigma'', \langle\langle B\ ?\ V'' : V' \rangle\rangle} \quad [\text{F-FOLD-CONSISTENT}]$$

[F-SELECT], [F-PROJ], [F-JOIN], and [F-UNION] formalize the relational calculus operators on tables of faceted rows.

The rules for folding are more interesting. If a row $(B, \bar{s})$ is inconsistent (*i.e.*, not visible to) the current program counter label pc, then rule [F-FOLD-INCONSISTENT] ignores that row. If the row is consistent, then rule [F-FOLD-CONSISTENT] applies the fold operator V_f to the row contents $\bar{s}$ and the accumulator V', producing a new accumulator V''. The result of that fold step is $\langle\langle B\ ?\ V'' : V' \rangle\rangle$, a faceted expression that appears like V'' to principals that can see the B-labeled row and like V' to other principals.

The faceted execution semantics describe the propagation of labels and facets for the purpose of complying with policies at computation sinks. λ^{JDB} expressions do not perform I/O, while λ^{JDB} statements include the effectful construct print $\{e_v\}$ e_r that prints expression e_r under the policies and viewing context e_v. We provide the λ^{jeeves} rules for declaring labels, attaching policies, and assigning labels for printing in Appendix A.

4.3 Application-Database Policy Compliance

λ^{jeeves} [8] has the properties that 1) a single faceted execution is equivalent to multiple different executions without faceted values and 2) the system cannot leak sensitive information through the output or the choice of output channel. We prove that the properties extend to λ^{JDB}.

The proof involves extending the *projection* property of λ^{jeeves}: a single execution with faceted values *projects* to multiple different executions without faceted values. To prove this property, we first define what it means to be a *view* and to be *visible*. A *view* L is a set of principals. B is visible to view L (written $B \sim L$) if $\forall k \in B.k \in L$ and $\forall \neg k \in B.k \notin L$. We extend views to values:

$$\begin{aligned} L : Val(\text{with facets}) &\to Val(\text{without facets}) \\ L(R) &= R \\ L(\langle k\ ?\ F_1 : F_2 \rangle) &= \begin{cases} L(F_1) & k \in L \\ L(F_2) & k \notin L \end{cases} \\ L(\text{table } T) &= \{(\emptyset, \bar{s}) \mid (B, \bar{s}) \in T, B \text{ visible to } L\} \end{aligned}$$

We extend views to expressions:

$$L(\langle k\ ?\ e_1 : e_2 \rangle) = \begin{cases} L(e_1) & k \in L \\ L(e_2) & k \notin L \end{cases}$$

For all other expression types we recursively apply the view to subexpressions.

We then prove the Projection Theorem. The full proof is in Appendix E. Proofs of the key lemmas are in Appendices B and C.

Theorem 1 (Projection)**.** *Suppose* $\Sigma, e \Downarrow_{pc} \Sigma', V$. *Then for any view L for which pc is visible,*

$$L(\Sigma), L(e) \Downarrow_{\emptyset} L(\Sigma'), L(V)$$

The Projection Theorem allows us to extend λ^{jeeves}'s property of termination-insensitive non-interference. To state the theorem we first define two faceted values to be *L-equivalent* if they have identical values for the view L. This notion of L-equivalence naturally extends to stores $(\Sigma_1 \sim_{pc} \Sigma_2)$ and expressions $(e_1 \sim_{pc} e_2)$. The theorem is as follows:

Theorem 2 (Termination-Insensitive Non-Interference)**.** *Let L be any view. Suppose* $\Sigma_1 \sim_L \Sigma_2$ *and* $e_1 \sim_L e_2$*, and that:*

$$\Sigma_1, e_1 \Downarrow_{\emptyset} \Sigma_1', V_1 \qquad \Sigma_2, e_2 \Downarrow_{\emptyset} \Sigma_2', V_2$$

then $\Sigma_1' \sim_L \Sigma_2'$ *and* $V_1 \sim_L V_2$.

The Termination-Insensitive Non-Interference Theorem allows us to extend the termination-insensitive *policy compliance* theorem of λ^{jeeves} [8]: data is revealed to an external observer only if it is allowed by the policies specified.

4.4 Early Pruning

The Early Pruning optimization involves shrinking a table T by keeping each row $(B, \bar{s})$ only when B is consistent with the viewer constraint described by pc. We show the rule below:

$$\frac{\begin{array}{c}\Sigma, e \Downarrow_{pc} \Sigma', (\text{table } T) \\ T' = \{(B, \bar{s}) \in T \mid B \text{ consistent with } pc\}\end{array}}{\Sigma, e \Downarrow_{pc} \Sigma', (\text{table } T')} \quad [\text{F-PRUNE}]$$

We prove the Projection Theorem holds with this extension.

5. Implementation

While previous implementations of Jeeves [8, 47] use Scala, we implement Jacqueline in Python, as an extension of Django [3], because of the popularity of both for web programming. Our code is available at `https://github.com/jeanqasaur/jeeves`.

5.1 Python Embedding of the Jeeves Runtime

We implemented Jeeves as a library that dynamically rewrites code to behave according to the λ^{jeeves} semantics. The library exports functions for creating labels, creating sensitive values, attaching policies, and using policies to show values. Our implementation supports a subset of Python's syntax that includes if-statements, for-loops, and return statements.

5.1.1 Faceted Execution

To support faceted execution, the implementation defines a Facet data type for primitives and objects where the facets may themselves be faceted. A value may exist only in some execution paths, in which case we use a special object Unassigned() for other paths. To perform faceted execution, the implementation uses operator overloading and dynamic source transformation via the macro library MacroPy [6]. The source transformation intercepts evaluation of conditionals, loops, assignments, and function calls. The implementation handles local assignment by replacing a function's local scope with a Namespace object determining scope. To prevent implicit flows, the runtime keeps track of path conditions to index state updates, database writes, and policy declarations.

5.1.2 Evaluating Policies at Computation Sinks

The runtime maps labels to policies. If there are no mutual dependencies between policies and sensitive values, the runtime evaluates policies to determine label values. Otherwise, the runtime produces an ordering over Boolean label assignments and uses the SAT subset of the Z3 SMT solver [33] to find a satisfying assignment.

5.2 Jacqueline Implementation

We extend Django's functionality by "monkey-patching," inheriting from Django's classes and overloading the methods of the FORM. The FORM is responsible for 1) marshalling between faceted representations in the application and database and 2) managing the meta-data to track facets in the database. To represent faceted values, the FORM creates schemas with additional meta-data columns. The FORM reconstructs facets from the meta-data by looking up policies from object schemas and adding them to the runtime environment. We implement the Early Pruning optimization by reconstructing only the relevant facets when the runtime knows the viewer. FORM queries manipulate the meta-data columns in addition to the actual columns. Programmers may access the database only through the supported API.

6. Jacqueline in Practice

We built 1) a conference management system, 2) a health record manager, and 3) a course management system to evaluate Jacqueline along the following dimensions:

- **Code architecture.** We compare the implementation of the Jacqueline conference management system to an implementation with hand-coded policies in Django. We demonstrate that Jacqueline helps with both centralizing policies and with size of policy code.
- **Performance.** We show that for representative actions, Jacqueline has comparable—and, in one case, better—performance compared to Django. For the stress tests, the Jacqueline programs often have close to zero overhead and at most a 1.75x slowdown compared to vanilla Django. We also demonstrate the effectiveness of and necessity of the Early Pruning optimization.

We deployed our conference management system to run an academic workshop [1].

6.1 Case Study Applications

Conference management system. We support user registration, update of profile information, designation of roles (*i.e.* PC member), paper and review submission, and assignment of reviews. Permissions depend on the current stage of the conference: submission, review, or decision.

Health record manager. We implemented a simple health record system based on a representative fragment of the privacy standards described in the Health Insurance Portability and Accountability Act (HIPAA) [9, 36]. HIPAA describes

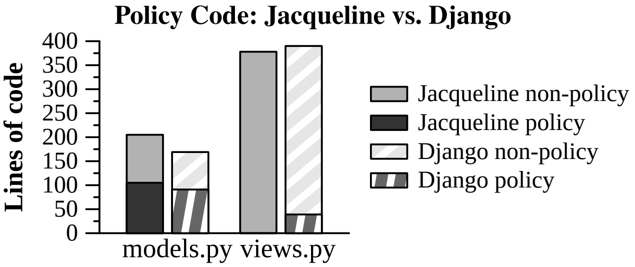


Figure 6. Distribution of policy code with Jacqueline and Django conference management systems.

how individuals, hospitals, and insurance companies may view a medical history depending on roles and stateful information such as whether there exists a permission waiver.

Course manager. Our tool allows instructors and students to organize assignments and submissions.

6.2 Code Comparisons

We compare our Jacqueline implementation of a conference management system against a Django implementation of the same system. We demonstrate that 1) Jacqueline reduces the trusted computing base and 2) separating policies and other functionality decreases policy code size.

6.2.1 Django Conference Management System

We compare the lines of code in the Jacqueline and Django conference management systems in Figure 6. (Note that Jacqueline counts are bloated from the additional imports and function decorators required.) Jacqueline demonstrates advantages in both the distribution and size of policy code. In the Jacqueline implementation, policy code is confined to the models.py file describing the data schemas, while in the Django implementation, there are also policies throughout the controller file views.py. The Jacqueline implementation has 106 total lines of policy code, whereas the Django implementation has 130 lines manifesting as repeated checks and filters across views.py. While the Django code requires auditing the 575 lines of models.py and views.py, the Jacqueline code requires auditing only the 200 lines of models.py (~200 lines of code), reducing the size of the application-specific trusted code base by 65%.

6.3 Performance Measurements

We measured times using an Amazon EC2 m3.2xlarge instance running Ubuntu 14.04 with 30GB of memory, two 80GB SSD drives, and eight virtual 64-bit Intel(R) Xeon(R) CPU E5-2670 v2 2.50Ghz processors. We use the FunkLoad testing framework [4] for HTTP requests across the network, excluding CSS and images. We average over 10 rapid sequential requests. We test with sequential users because how well Jacqueline handles concurrent users compared to Django simply depends on the amount of available memory.

CFM Representative Actions

View single paper			View single user		
Papers	Jacq.	Django	Users	Jacq.	Django
8	0.160s	0.177s	8	0.164s	0.158s
16	0.165s	0.175s	16	0.164s	0.159s
32	0.160s	0.177s	32	0.164s	0.159s
64	0.159s	0.173s	64	0.164s	0.159s
128	0.160s	0.173s	128	0.167s	0.158s
256	0.159s	0.173s	256	0.163s	0.159s
512	0.159s	0.178s	512	0.169s	0.162s
1024	0.161s	0.173s	1024	0.163s	0.159s

Figure 7. Times to view profiles for a single paper and single user, in Jacqueline and Django.

We show 1) policy enforcement in Jacqueline has reasonable overheads, especially compared to Django and 2) Early Pruning is effective and often necessary.

6.3.1 Representative Actions

We increased the number of relevant database entries and measured the time it takes to view the profiles for single papers and users. We show these numbers, as well as comparisons to Django, in Figure 7. The time it takes to load these profiles is under 2ms and roughly equivalent to the time it takes to do the equivalent action in Django. For viewing a single paper, Jacqueline actually performs better than the Django implementation. This is because in the Django code, the implementation needs iterate over collections of data rows again in order to apply policy checks. In the Jacqueline implementation, the framework applies the policies and resolves each one once. Times for submitting a single paper scale similarly.

6.3.2 Stress Tests

In Figure 8 we show results for displaying an increasing number of papers and users in both implementations. In these tests, the system resolves different policies for each paper and user field. The graphs demonstrate that with both Jacqueline and Django, the time to load data scales linearly with respect to the underlying algorithms and Jacqueline has a 1.75x overhead for showing all papers. The overhead comes from fetching both versions of data before resolving the policies. Integrating policies more deeply with the database could reduce this overhead. There is no solver overhead, as there are no mutual dependencies between sensitive values and policies.

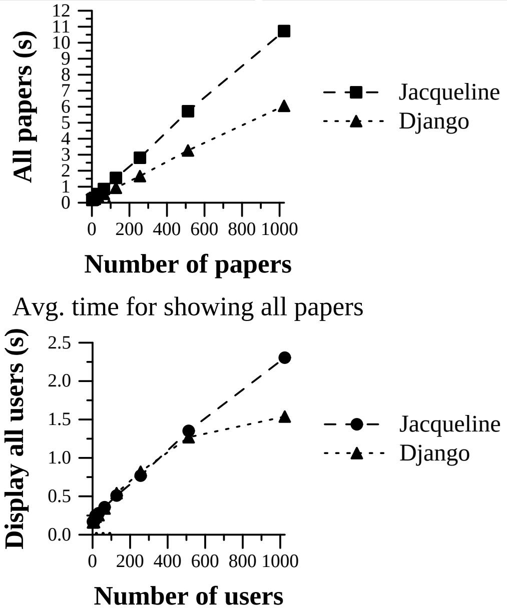
In Figure 9 we show the stress tests from our health record manager and course manager demonstrate similar scaling. These are truly stress tests: most systems will not load a thousand data rows at once, especially when each value has its own policy involving database queries.

6.3.3 Early Pruning Optimization

We found the Early Pruning optimization to be necessary for nontrivial computations over sensitive values. In the course

CFM Stress Tests

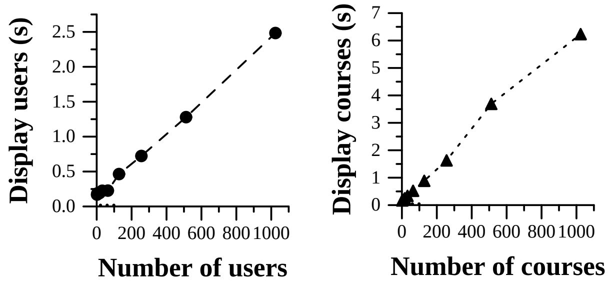
View all papers			View all users		
# P	Jacq.	Django	# U	Jacq.	Django
8	0.241s	0.201s	8	0.172s	0.163s
16	0.299s	0.241s	16	0.249s	0.234s
32	0.542s	0.388s	32	0.279s	0.254s
64	0.855s	0.554s	64	0.358s	0.341s
128	1.551s	0.931s	128	0.510s	0.541s
256	2.810s	1.633s	256	0.769s	0.820s
512	5.717s	3.265s	512	1.352s	1.269s
1024	10.729s	6.055s	1024	2.305s	1.538s



Avg. time for showing all papers

Figure 8. Times to view list of summary information for all papers and all users, in Jacqueline and Django.

Other Stress Tests



Health records

(a)

Course manager

(b)

Figure 9. Jacqueline stress tests for other case studies.

Courses	Without pruning	With pruning
4	0.377s	0.185s
8	64.024s	0.192s
16	–	0.248s
32	–	0.337s
64	–	0.522s
128	–	0.886s
256	–	1.630s
512	–	3.691s
1024	–	6.233s

Figure 10. Showing all courses, with and without Early Pruning.

manager stress test, the page that shows all courses also looks up the instructors for each course, leading to blowup. We show in Figure 10 how for just eight courses and instructors, the system begins to hit memory limits. Because Early Pruning can simplify other computations after the viewer is known, these computations are only problematic when they are used to compute the viewer. We do not expect such computations to be common.

7. Related Work

Our approach builds on a long history of work in information flow control [7, 12, 14, 18, 21, 28, 29, 34, 37, 44, 48]. The policy-agnostic approach differs from prior work in the following key way. Using prior approaches, the programmer needs to implement the policy checks and filters correctly across the program. Our solution mitigates programmer burden by leveraging the language runtime to produce outputs adhering to policies. This is similar in philosophy to angelic nondeterminism [11], program repair [40, 41], and acceptability-oriented computing [38, 39].

Prior work on information flow across the application-database boundary focuses on rejecting queries that leak information, rather than on modifying queries to enforce policies. SeLINQ [42], the work of Lourenço and Caires [30], and Ur/Web use static types. DBTaint [19], Passe [10], and Hails [25] perform dynamic analysis. SIF [17] combines static labels and dynamic checks. There are also approaches based on symbolic execution [27], secure multi-execution [13, 20, 23], and analysis of data provenance [2, 15] focused on rejecting programs that violate desired properties.

Policy-agnostic programming differs from other approaches in how data may affect control flow. Variational data structures [45] encapsulate properties related to program customization, but data does not affect control flow. Aspect-oriented programming [26, 43] has similar goals to policy-agnostic programming of separating program concerns, but aspects must be implemented at specific control flow points and cannot alter control flow.

Our approach addresses information flow as opposed to access control [24, 32, 35]. Access control focuses on preventing leaks at application endpoints and does not address leaks through indirect flows or implicit flows. Similarly, work on multi-level databases [22, 31] focuses on the storage and access control issues surrounding data at different levels of access in the database.

8. Conclusions

We demonstrate that it is practically feasible to achieve policy compliance by construction in database-backed applications. We present a technique for precise, dynamic information flow control that tracks sensitive values and policies through database queries and updates as well as application code. The technique supports a policy-agnostic programming model that allows the program to specify each information flow policy once, instead of as repeated intertwined checks across the program. The web framework performs different computations depending on the viewer, according to the policies. The shift of responsibility to the framework reduces the opportunity for programmer error to cause information leaks.

Our solution works with existing implementation of relational databases and yields formal guarantees across the application and database. We implement these ideas in Jacqueline, a Python web framework, and demonstrate that, compared to traditional applications with hand-coded policies, applications written using Jacqueline have less policy code and run with reasonable, often negligible, overheads. This work represents a promising step towards securing database-backed web applications.

References

[1] Workshop, 2014. scrubbed to protect double-blind.

[2] Toward provenance-based security for configuration languages. In *Presented as part of the 4th USENIX Workshop on the Theory and Practice of Provenance*, Berkeley, CA, 2012. USENIX. URL https://www.usenix.org/conference/tapp12/workshop-program/presentation/Anderson.

[3] Django: The web framework for perfectionists with deadlines. https://www.djangoproject.com, accessed July 3, 2015.

[4] Funkload. http://funkload.nuxeo.org, accessed July 3, 2015.

[5] HotCRP bug report: Download PC review assignments obeys paper administrators. https://github.com/kohler/hotcrp/commit/80ff96606bbe26e242ac7ebca85b440f2dbffebb, accessed July 3, 2015.

[6] MacroPy. https://github.com/lihaoyi/macropy, accessed July 3, 2015.

[7] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. Sharing mobile code securely with information flow control. In *Oakland*, 2012.

[8] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama. Faceted execution of policy-agnostic programs. In *PLAS*, 2013.

[9] A. Barth, A. Datta, J. C. Mitchell, and H. Nissenbaum. Privacy and contextual integrity: Framework and applications. Oakland, 2006.

[10] A. Blankstein and M. J. Freedman. Automating isolation and least privilege in web services. 2014.

[11] R. Bodik, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor. Programming with angelic nondeterminism. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, New York, NY, USA, 2010. ACM.

[12] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *ESOP*, volume 3924 of *LNCS*. Springer Verlag, 2006.

[13] R. Capizzi, A. Longo, V. Venkatakrishnan, and A. Sistla. Preventing information leaks through shadow executions. In *ACSAC*.

[14] J. Chen, R. Chugh, and N. Swamy. Type-preserving compilation of end-to-end verification of security enforcement. In *PLDI*, 2010.

[15] J. Cheney. A formal framework for provenance security. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, pages 281–293, Washington, DC, USA, 2011. IEEE Computer Society. URL http://dx.doi.org/10.1109/CSF.2011.26.

[16] A. Chlipala. Static checking of dynamically-varying security policies in database-backed applications. OSDI'10, 2010.

[17] S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *USENIX*, 2007.

[18] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. *SIGPLAN Not.*, 44(6):50–62, June 2009. ISSN 0362-1340. . URL http://doi.acm.org/10.1145/1543135.1542483.

[19] B. Davis and H. Chen. DBTaint: Cross-application information flow tracking via databases. In J. K. Ousterhout, editor, *USENIX Conference on Web Application Development, WebApps'10, Boston, Massachusetts, USA, June 23-24, 2010*. USENIX Association, 2010. URL https://www.usenix.org/publications/proceedings/?f[0]=im_group_audience%3A5.

[20] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Secure multi-execution of web scripts: Theory and practice. *Journal of Computer Security*, 22(4), 2014.

[21] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[22] D. E. Denning, S. G. Akl, M. Morgenstern, P. G. Neumann, R. R. Schell, and M. Heckman. Views for multilevel database security. In *IEEE Symposium on Security and Privacy*, pages 156–172. IEEE Computer Society, 1986. ISBN 0-8186-0716-5. URL http://dblp.uni-trier.de/db/conf/sp/sp1986.html#DenningAMNSH86.

[23] D. Devriese and F. Piessens. Noninterference through secure multi-execution. *Oakland*, 2010.

[24] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. ICSE '05. ACM, 2005.

[25] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. OSDI'12, 2012.

[26] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, 1997.

[27] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. Technical Report MSR-TR-2011-94, Microsoft Research Technical Report, 2011.

[28] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, New York, NY, USA, 2007. ACM.

[29] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: a platform for secure distributed computation and storage. In *SOSP*, 2009.

[30] L. Lourenço and L. Caires. Information flow analysis for valued-indexed data security compartments. In *Trustworthy Computing*, 2013.

[31] T. Lunt, D. Denning, R. Schell, M. Heckman, and W. Shockley. The seaview security model. *Software Engineering, IEEE Transactions on*, 16(6):593–607, Jun 1990. ISSN 0098-5589. .

[32] A. Milicevic, D. Jackson, M. Gligoric, and D. Marinov. Model-based, event-driven programming paradigm for interactive web applications. In *SPLASH 2013*. ACM, 2013.

[33] L. D. Moura and N. Björner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[34] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, 1999.

[35] J. P. Near and D. Jackson. Rubicon: bounded verification of web applications. In *SIGSOFT FSE*, 2012.

[36] Office for Civil Rights. Summary of the HIPAA privacy rule, 2003.

[37] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, Jan. 2003. URL http://gallium.inria.fr/~fpottier/publis/fpottier-simonet-toplas.ps.gz.

[38] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251254.1251275.

[39] M. Rinard, C. Cadar, and H. H. Nguyen. Exploring the acceptability envelope. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 21–30, New York, NY, USA, 2005. ACM. ISBN 1-59593-193-7. . URL http://doi.acm.org/10.1145/1094855.1094866.

[40] H. Samimi, E. D. Aung, and T. D. Millstein. Falling back on executable specifications. In *ECOOP*, 2010.

[41] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 277–287, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337257.

[42] D. Schoepe, D. Hedin, and A. Sabelfeld. SeLINQ: Tracking information across application-database boundaries. ICFP '14, New York, NY, USA, 2014. ACM.

[43] D. R. Smith. A generative approach to aspect-oriented programming. In *GPCE*, 2004.

[44] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, 2011.

[45] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden. Variational data structures: Exploring tradeoffs in computing with variability. In *Onward!*, 2014.

[46] A. Warwick. Facebook photo leak flaw raises security concerns. http://www.computerweekly.com/news/2240242708/Facebook-photo-leak-flaw-raises-security-concerns, March 2015. [Online; posted 20-March-2015].

[47] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *POPL*, 2012.

[48] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *SOSP*, October 2009.

A. Rules from λ^{jeeves}

We show the most relevant rules from the dynamic semantics for the Jeeves core language λ^{jeeves} [8].

A.1 Managing Labels

These rules describe how to declare labels and attach policies to labels. The rule [F-LABEL] dynamically allocates a label (label k in e), adding a fresh label to the store with the default policy of $\lambda x.true$. Any occurrences of k in e are α-renamed to k' and the expression is evaluated with the updated store. Policies may be further refined (restrict(k,e)) by the rule [F-RESTRICT], which evaluates e to a policy V that should be either a lambda or a faceted value comprised of lambdas. The additional policy check is restricted by pc, so that policy checks cannot themselves leak data. The rule joins the resulting policy check V_p with the existing policy for k, ensuring that policies can only become more restrictive.

[F-LABEL]

$$\frac{\begin{array}{c} k' \, fresh \\ \Sigma[k' := \lambda x.true], e[k := k'] \Downarrow_{pc} \Sigma', V \end{array}}{\Sigma, \text{label } k \text{ in } e \Downarrow_{pc} \Sigma', V'}$$

[F-RESTRICT]

$$\frac{\begin{array}{c} \Sigma, e \Downarrow_{pc} \Sigma_1, V \\ V_p = \langle\langle pc \cup \{k\} \ ? \ V : \lambda x.true \rangle\rangle \\ \Sigma' = \Sigma_1[k := \Sigma_1(k) \wedge_f V_p] \end{array}}{\Sigma, \text{restrict}(k,e) \Downarrow_{pc} \Sigma', V}$$

A.2 Displaying Outputs

The rule [F-PRINT] handles print statements (print $\{e_1\}$ e_2), where the result of evaluating e_2 is printed to the channel resulting from the evaluation of e_1. Both the channel V_f and the value to print V_c may be faceted values. The describes how to select the facets that correspond with our specified policies. The rule determines the set of relevant labels through the transitive closure function *closeK*. The labels are used to construct e_p from the relevant policies in the store Σ_2. The rule evaluates e_p and applies it to V_f, returning the policy check V_p that is a faceted value containing booleans. The rule chooses a program counter pc such that the policies are satisfied. This corresponds to a label assignment that determines the channel f and the value to print R.

[F-PRINT]

$$\frac{\begin{array}{c} \Sigma, e_1 \Downarrow_{\emptyset} \Sigma_1, V_f \\ \Sigma_1, e_2 \Downarrow_{\emptyset} \Sigma_2, V_c \\ \{\ k_1 \ \ldots \ k_n\ \} = closeK(labels(e_1) \ \cup \ labels(e_2), \Sigma_2) \\ e_p = \lambda x.true \ \wedge_f \ \Sigma_2(k_1) \ \wedge_f \ \ldots \ \wedge_f \ \Sigma_2(k_n) \\ \Sigma_2, e_p \ V_f \Downarrow_{\emptyset} \Sigma_3, V_p \\ \text{pick } pc \text{ such that } pc(V_f) = f, pc(V_c) = R, \ pc(V_p) = true \end{array}}{\Sigma, \text{print } \{e_1\} \ e_2 \Downarrow V_p, f : R}$$

$$\begin{array}{lcl} closeK(K,\Sigma) & = & \text{let } K' = \bigcup_{k \in K} \ labels(\Sigma(k)) \text{ in} \\ & & \text{if } K' = K \\ & & \quad \text{then } K \\ & & \quad \text{else } closeK(K', \Sigma) \end{array}$$

B. Proof of Lemma 1

Lemma 1 (A)**.**

$$L(\langle\langle k \ ? \ V_1 : V_2 \rangle\rangle) = \begin{cases} L(V_1) & \textit{if } k \in L \\ L(V_2) & \textit{if } k \notin L \end{cases}$$

Proof. By case analysis on the definition of $\langle\langle k \ ? \ V_1 : V_2 \rangle\rangle$. Let $x = L(\langle\langle k \ ? \ V_1 : V_2 \rangle\rangle)$.

- If $x = L(\langle k \ ? \ F_1 : F_2 \rangle)$ for some non-table values F_1 and F_2, then this case holds since
  - $x = L(F_1)$ if $k \in L$.
  - $x = L(F_2)$ if $k \notin L$.
- If $x = L(\langle\langle k \ ? \text{ table } T_1 : \text{table } T_2 \rangle\rangle)$, then $x = L(\text{table } T)$ where
  $T = \{(B \cup \{k\}, \bar{s}) \mid (B, \bar{s}) \in T_1, \neg k \notin B\}$
  $\cup \ \{(B \cup \{\neg k\}, \bar{s}) \mid (B, \bar{s}) \in T_2, k \notin B\}$.
  And so
  $x = \{(\emptyset, \bar{s}) \mid (B, \bar{s}) \in T_1, \neg k \notin B, B \cup \{k\} \sim L\}$
  $\cup \ \{(\emptyset, \bar{s}) \mid (B, \bar{s}) \in T_2, k \notin B, B \cup \{\neg k\} \sim L\}$.
  - If $k \in L$, then $B \cup \{\neg k\} \not\sim L$ and $B \cup \{k\} \sim L => \neg k \notin B$, and so
    $x = \{(\emptyset, \bar{s}) \mid (B, \bar{s}) \in T_1, B \sim L\}$
    $= L(\text{table } T_1)$, as required.
  - If $k \notin L$, then this case holds by a similar argument as the previous case.

□

C. Proof of Lemma 2

Lemma 2 (B)**.**

$$L(\langle\langle B \ ? \ V_1 : V_2 \rangle\rangle) = \begin{cases} L(V_1) & \textit{if } B \sim L \\ L(V_2) & \textit{if } \neg(B \sim L) \end{cases}$$

Proof. The proof is by induction and case analysis on the derivation of $L(\langle\langle B \ ? \ V_1 : V_2 \rangle\rangle)$. Let $x = L(\langle\langle B \ ? \ V_1 : V_2 \rangle\rangle)$.

- If $B = \emptyset$, then $B \sim L$, so $x = L(V_1)$ as required.
- Otherwise, $B = B' \cup \{k\}$.
  - If $B \sim L$, then
    $x = L(\langle\langle k \ ? \ \langle\langle B' \ ? \ V_1 : V_2 \rangle\rangle : V_2 \rangle\rangle)$
    $= L(\langle\langle B' \ ? \ V_1 : V_2 \rangle\rangle)$ by Lemma 1, since $k \in L$
    $= L(V_1)$ by induction, as $B' \sim L$.
  - Otherwise, $B \not\sim L$, then
    - if $k \notin L$, then $x = L(V_2)$ by Lemma 1.

  - otherwise $k \in L$, so $B' \not\sim L$.
    Therefore, $x = L(\langle\langle B' \ ? \ V_1 : V_2 \rangle\rangle) = L(V_2)$, as required.

□

D. Lemma 3

If a set of branches is compatible with view L, then we can execute only using that view. We prove an additional lemma that if pc is not visible, then execution should not affect the environment under projections of L.

Lemma 3 (C)**.** *If pc is not visible to L and*

$$\Sigma, e \Downarrow_{pc} \Sigma', V$$

then $L(\Sigma) = L(\Sigma')$.

Proof. By induction on the derivation of $\Sigma, e \Downarrow_{pc} \Sigma', V$ and by case analysis on the final rule used in that derivation.

- The following cases hold because $\Sigma = \Sigma'$: [F-VAL], [F-DEREF-NULL], [F-DEREF], [F-CTXT], [F-APP], [F-LEFT], [F-RIGHT], [F-ROW], [F-SELECT], [F-PROJECT], [F-JOIN], [F-UNION], [F-FOLD-EMPTY], and [F-FOLD-INCONSISTENT].
- Cases [F-APP], [F-LEFT], [F-RIGHT], [F-STRICT], [F-CTXT], and [F-FOLD-INCONSISTENT] hold by induction.
- For case [F-SPLIT], we note that since $pc \sim L$, $\forall k.pc \cup \{k\} \not\sim L$ and $pc \cup \{\neg k\} \not\sim L$. Therefore, this case also holds by induction.
- Similarly, for case [F-FOLD-CONSISTENT], since $pc \not\sim L$, $\forall B.pc \cup B \not\sim L$, and so this case holds by induction.
- For case [F-REF], $\forall a'$ where $a' \neq a, \Sigma(a') = \Sigma'(a')$.
  Since $pc \not\sim L, L(\Sigma(a)) = 0$ by Lemma 1, as required.
- For case [F-ASSIGN], $\forall a'$ where $a' \neq a, \Sigma(a') = \Sigma'(a')$.
  Since $pc \not\sim L, L(\Sigma(a)) = L(\Sigma'(a))$ by Lemma 1, as required.

□

This lemma is also useful in the proof of the Projection Theorem.

E. Proof of Theorem 1 (Projection)

For convenience, we restate Theorem 1.
Suppose $\Sigma, e \Downarrow_{pc} \Sigma', V$. *Then for any view L for which pc is visible,*

$$L(\Sigma), L(e) \Downarrow_{\emptyset} L(\Sigma'), L(V)$$

The proof extends L to project evaluation contexts. They may project away the hole and so map evaluation contexts to expressions, in which case filling the result is a no-op.

We capture in the following lemma the property that if a branch B is inconsistent with the program counter pc, at most one of B and pc may be visible to any given view L.

Lemma 4. *If B is inconsistent with pc and* $pc \sim L$*, then* $B \not\sim L$.

With these properties established, we now prove projection.

Proof. By induction on the derivation of $L(\Sigma), L(e) \Downarrow_{\emptyset} L(\Sigma'), L(V)$ and by case analysis on the final rule used in that derivation.

- The following cases hold trivially: [F-VAL], [F-DEREF], [F-DEREF-NULL], [F-ROW], [F-PROJECT], and [F-UNION].
- For case [F-SELECT], $e = \sigma_{i=j}$ (table T), so

  $$\Sigma, \sigma_{i=j} \ (\text{table } T) \Downarrow_{pc} \Sigma, (\text{table } T')$$

  where $T' = \{(B, \bar{s}) \mid s_i = s_j\}$.
  This case holds since $L(\text{table } T) = \{(\emptyset, \bar{s}) \mid (B, \bar{s}) \in T, B \sim L\}$ and $L(\text{table } T') = \{(\emptyset, \bar{s}) \mid (B, \bar{s}) \in T, B \sim L, s_i = s_j\}$,

- For case [F-JOIN], $e = (\text{table } T_1) \bowtie (\text{table } T_2)$, so

  $$\Sigma, (\text{table } T_1) \bowtie (\text{table } T_2) \Downarrow_{pc} \Sigma, (\text{table } T)$$

  where $T = \{B.B', \bar{s}.\overline{s'}) \mid (B, \bar{s}) \in T_1, (B', \overline{s'}) \in T_2\}$.
  $L(T) = \{(B.B', \bar{s}.\overline{s'}) \mid (B, \bar{s}) \in T_1, (B', \overline{s'}) \in T_2, B.B' \sim L\}$, so this case holds.
- For case [F-CTXT], $e = E[e']$. By the antecedents of this rule

  $$\begin{array}{l} E \neq [] \\ e' \text{ not a value} \\ \Sigma, e' \Downarrow_{pc} \Sigma_1, V' \\ \Sigma_1, E[V'] \Downarrow_{pc} \Sigma', V \end{array}$$

  Note that $L(E[V']) = L(E)[L(V')]$, etc., so by induction

  $$\begin{array}{l} L(\Sigma), L(e') \Downarrow_{\emptyset} L(\Sigma_1), L(V') \\ L(\Sigma_1), L(E)[L(V')] \Downarrow_{\emptyset} L(\Sigma'), L(V) \end{array}$$

  Therefore, $L(\Sigma), L(E[e]) \Downarrow_{\emptyset} L(\Sigma'), L(V)$, as required.
- For case [F-STRICT], $e = S[\langle k \ ? \ V_1 : V_2 \rangle]$. By the antecedents of this rule

  $$\Sigma, \langle k \ ? \ S[V_1] : S[V_2] \rangle \Downarrow_{pc} \Sigma', V'$$

  We now consider each possible case for the next step in the derivation.
  - For subcase [F-LEFT], we know that $k \in pc, k \in L$ and

    $$\Sigma, S[V_1] \Downarrow_{\emptyset} \Sigma', V$$

    By induction, $L(\Sigma), L(\langle k \ ? \ S[V_1] : S[V_2] \rangle) \Downarrow_{\emptyset} L(\Sigma'), L(V')$.

  - Subcase [F-RIGHT] holds by a similar argument.
  - For subcase [F-SPLIT], $k \notin pc, \neg k \notin pc$ and

    $$\begin{array}{l} \Sigma, S[V_1] \Downarrow_{pc \cup \{k\}} \Sigma'', V'' \\ \Sigma'', S[V_2] \Downarrow_{pc \cup \{\neg k\}} \Sigma', V''' \\ V = \langle\langle k \ ? \ V'' : V''' \rangle\rangle \end{array}$$

    - If $k \in L$, then by induction we have $L(\Sigma), L(S[V_1]) \Downarrow_{\emptyset} L(\Sigma''), L(V'')$.
      $L(\Sigma'') = L(\Sigma')$ by Lemma 3, and $L(V) = L(V'')$.
      Therefore, $L(\Sigma), L(S[V_1]) \Downarrow_{\emptyset} L(\Sigma'), L(V')$, as required.

  - If $k \notin L$, then this case holds by a similar argument.
- For case [F-FOLD-EMPTY], we have

  $$\Sigma, \text{fold } V_f\ V_b\ (\text{table } \epsilon) \Downarrow_{pc} \Sigma, V_b$$

  Clearly, $L(\Sigma), \text{fold } L(V_f)\ L(V_b)\ L(\text{table } \epsilon) \Downarrow_{\emptyset} L(\Sigma), L(V_b)$.

- For case [F-FOLD-INCONSISTENT], we have

  $$e = \text{fold } V_f\ V_p\ (\text{table } (B, \bar{s}).T)$$

  . By the antecedents of this rule, we have

  $$\Sigma, \text{fold } V_f\ V_b\ (\text{table } T) \Downarrow_{pc} \Sigma', V$$
  $$B \text{ is inconsistent with } pc$$

  By Lemma 4, $B \not\sim L$.
  Therefore, $L(\text{table } (B, \bar{s}).T) = L(\text{table } T)$.
  By the [F-FOLD-EMPTY] rule,

  $$L(\Sigma), \text{fold } L(V_f)\ L(V_b)\ L(\text{table } (B, \bar{s}).T) \Downarrow_{\emptyset} L(\Sigma'), L(V)$$

  By induction, $L(\Sigma), L(\text{fold } V_f\ V_b\ (\text{table } T)) \Downarrow_{\emptyset} L(\Sigma'), L(V)$, as required.
- For case [F-FOLD-CONSISTENT], $e = \text{fold } V_f\ V_b\ (\text{table } T)$. By the antecedents of this rule, we have

  $$\Sigma, \text{fold } V_f\ V_b\ (\text{table } T) \Downarrow_{pc} \Sigma_1, V_1$$
  $$B \text{ is consistent with } pc$$
  $$\Sigma_1, V_f\ \bar{s}\ V_1 \Downarrow_{pc \cup B} \Sigma', V_2$$
  $$V = \langle\!\langle B\ ?\ V_2 : V_1 \rangle\!\rangle$$

  - If $B \sim L$, then $pc \cup B \sim L$.
    By induction,

    $$L(\Sigma), L(\text{fold } V_f\ V_b\ (\text{table } T)) \Downarrow_{\emptyset} L(\Sigma_1), L(V_1)$$
    $$L(\Sigma_1), L(V_f\ \bar{s}\ V_1) \Downarrow_{\emptyset} L(\Sigma'), L(V_2)$$

    By Lemma 2, $L(V) = L(\langle\!\langle B\ ?\ V_2 : V_1 \rangle\!\rangle)$, as required.
  - Otherwise, $B \not\sim L$, and therefore $pc \cup B \not\sim L$. By Lemma 3, $L(\Sigma_1) = L(\Sigma')$.
    By induction, $L(\Sigma), L(\text{fold } V_f\ V_b\ (\text{table } T)) \Downarrow_{\emptyset} L(\Sigma_1), L(V_1)$.
    $L(\text{table } (B, \bar{s}).T) = L(\text{table } T)$.
    By Lemma 2, $L(V) = L(\langle\!\langle B\ ?\ V_2 : V_1 \rangle\!\rangle)$, as required.
- For case [F-LEFT], $e = \langle k\ ?\ e_1 : e_2 \rangle$.
  By the antecedents of this rule, we have

  $$k \in pc$$
  $$\Sigma, e_1 \Downarrow_{pc} \Sigma', V$$

  Since $k \in pc$, $L(e) = L(e_1)$.
  By induction, $L(\Sigma), L(e_1) \Downarrow_{\emptyset} L(\Sigma'), L(V)$.

- Case [F-RIGHT] holds by a similar argument.
- For case [F-SPLIT], $e = \langle k\ ?\ e_1 : e_2 \rangle$.
  By the antecedents of this rule, we have

  $$k \notin pc \qquad \neg k \notin pc$$
  $$\Sigma, e_1 \Downarrow_{pc \cup \{k\}} \Sigma_1, V_1$$
  $$\Sigma_1, e_2 \Downarrow_{pc \cup \{\neg k\}} \Sigma', V_2$$
  $$V = \langle\!\langle k\ ?\ V_1 : V_2 \rangle\!\rangle$$

  - If $k \in L$, then by induction $L(\Sigma), L(e_1) \Downarrow_{\emptyset} L(\Sigma_1), L(V_1)$.
    $L(\Sigma_1) = L(\Sigma')$ by Lemma 3, and by Lemma 1
    $L(V) = L(\langle\!\langle k\ ?\ V_1 : V_2 \rangle\!\rangle) = L(V_1)$, as required.
  - Otherwise $\neg k \in L$, so $L(\Sigma) = L(\Sigma_1)$ by Lemma 3.
    By induction, $L(\Sigma_1), L(e_2) \Downarrow_{\emptyset} L(\Sigma'), L(V_2)$,
    and by Lemma 1 $L(V) = L(\langle\!\langle k\ ?\ V_1 : V_2 \rangle\!\rangle) = L(V_2)$,
    as required.
- For case [F-APP], $e = (\lambda x.e'\ V')$. By the antecedents of this rule,

  $$\Sigma, e'[x := V'] \Downarrow_{pc} \Sigma', V$$

  We know that $L(e) = L(\lambda x.e'\ V') = L(e'[x := V'])$.
  By induction, $L(\Sigma), L(e'[x := V']) \Downarrow_{\emptyset} L(\Sigma'), L(V)$, as required.
- For case [F-REF], $e = \text{ref } V'$. By the antecedents of this rule

  $$a \notin dom(\Sigma)$$
  $$\Sigma' = \Sigma[a := \langle\!\langle pc\ ?\ V' : 0 \rangle\!\rangle]$$

  Without loss of generality, we assume that both evaluations allocate the same address a. Since $a \notin dom(\Sigma), a \notin dom(L(\Sigma))$.
  Also, we know that $\forall a' \in dom(\Sigma), \Sigma(a') = \Sigma'(a')$, and therefore $L(\Sigma(a')) = L(\Sigma'(a'))$.
  Since $pc \sim L$, $L(\Sigma'(a)) = L(\langle\!\langle pc\ ?\ V' : 0 \rangle\!\rangle) = L(V')$ by Lemma 2. Since $L(\langle\!\langle \emptyset\ ?\ V' : 0 \rangle\!\rangle) = L(V') = L(V)$, this case holds.
- For case [F-ASSIGN], $e = (a{:=}V)$. By the antecedent of this rule, $\Sigma' = \Sigma[a := \langle\!\langle pc\ ?\ V : \Sigma(a) \rangle\!\rangle]$. We know $\forall a' \in dom(\Sigma), \Sigma(a') = \Sigma'(a')$, and therefore $L(\Sigma(a')) = L(\Sigma'(a'))$.
  Since $L \sim pc$, $L(\Sigma'(a)) = L(\langle\!\langle pc\ ?\ V : \Sigma(a) \rangle\!\rangle) = L(V)$ by Lemma 2. And since $L(\langle\!\langle \emptyset\ ?\ V : \Sigma(a) \rangle\!\rangle) = L(V)$, this case holds.

□