# End-To-End Policy-Agnostic Security 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
UC Santa Cruz

Stephen Chong
Harvard University

## Abstract

Protecting sensitive data often requires implementing repeated security checks and filters throughout a program. This task is especially error-prone in web programs, where data flows between applications and databases. To reduce the opportunity for privacy leaks, we present Jacqueline, a web framework that automatically enforces security policies that restrict where sensitive data may flow. In Jacqueline, programmers specify information flow policies separately from the rest of the program. In turn, the remainder of the program is *policy-agnostic*: parametric with respect to the policies. The Jacqueline runtime differentiates outputs based on the policies and viewer by simulating simultaneous multiple executions. We demonstrate that this approach provides strong theoretical guarantees and is also practical. We formalize Jacqueline's object-relational mapping and prove end-to-end policy compliance. Our formalism uses standard relational operations and thus allows us to implement Jacqueline as an extension of the Django Python framework using an unmodified SQL database. We demonstrate the feasibility of the approach through three application case studies: a course manager, a health record system, and a conference management system that we have deployed to run a workshop. We compare to code written with hand-implemented policies, showing that not only does Jacqueline reduce lines of policy code, but also that the Jacqueline runtime has reasonable, and often negligible, overheads.

## 1. Introduction

From social networks to electronic health record systems, web programs increasingly process sensitive information. A standard way of managing sensitive data involves implementing repeated checks and filters throughout the program. However, missing access checks, incorrect computation of the viewer, and incorrect policy propagation can all release sensitive data to unauthorized viewers.

Interactions with databases further complicate the task of protecting sensitive data in web programs. In particular, the programmer must now reason about how sensitive data flows through both application code and database queries. Reasoning across the application-database boundary has led to leaks in systems from the HotCRP conference management system [4] to the social networking site Facebook [44]. Indeed, the patch for the recent HotCRP bug involves policy checks across application code and database queries.

We address the problem of protecting sensitive data in web programs by reducing the opportunity for error. We propose a *policy-agnostic* programming paradigm that allows the programmer to specify information flow policies separately from the rest of the applications. We present Jacqueline, a web framework that allows the programmer to specify policies only once, alongside data schemas. Jacqueline manages policy dependencies and guarantees end-to-end policy compliance across the application and database. A key advantage of Jacqueline is that it *works with unmodified relational databases*, allowing the programmer to use the policy-agnostic model without giving up the benefits of an optimized database.

### 1.1 Policy-Agnostic Programming

Jacqueline is based on the policy-agnostic programming model of the Jeeves language [10, 45]. Jeeves programs express computations independently of information flow policies. The runtime ensures that program behavior complies with the policies. Policies may depend on sensitive values: for example, who is allowed to learn a secret value may depend on the value itself. Jeeves provides strong guarantees, ensuring that policy enforcement does not itself leak sensitive information. However, Jeeves is unsuited for building realistic web applications for the following reasons:

- *No guarantees when interoperating with databases.* For performance reasons, web applications rely heavily on interactions with commodity databases. Unfortunately, a common problem with language-based approaches is that guarantees apply only to programs running entirely within the language. Indeed, Jeeves's policy enforcement guarantees fail when there is any interaction with an external database.
- *Expensive execution model.* Jeeves may explore exponentially many possible execution branches based on the possible viewers. This can become prohibitively expensive when sensitive values each have their own policies.

Jacqueline overcomes these limitations and enables policy-agnostic programming for web programs of realistic scale. A key insight is that rather than needing to modify existing databases to include policy checks, we can create a policy-agnostic *object-relational mapping* (ORM). With a standard ORM, the programmer does not write database queries directly, but instead relies on the framework to manipulate data between applications and databases. With a policy-agnostic ORM, the programmer relies on the framework to manipulate both data and policies. The challenge becomes, then, to design an ORM that can track sensitive data and policies through database queries when the database is not aware of sensitive values or policies. We observe that an ORM can do this through judicious manipulaton of meta-data. Jacqueline improves upon Jeeves in the following ways:

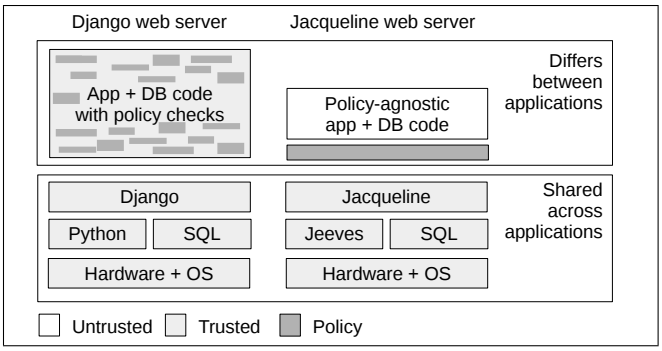


**Figure 1:** Application architecture in Django vs. Jacqueline.

- *End-to-end guarantees for database-backed applications.* We formalize the Jacqueline ORM in terms of standard relational operators to track sensitive values and policies through database queries. We prove that this yields a policy compliance property across the application and database.
- *Correctness-preserving optimization approach.* Jacqueline does not need to assume the viewer is unknown until output because it is common for web frameworks to track the viewing context. As soon as the runtime knows the viewing context, it can prune alternate execution branches. We formalize this optimization, show that it preserves end-to-end policy compliance, show that it allows Jacqueline to have reasonable overheads in practice, and demonstrate it is necessary for non-trivial computations involving sensitive values.

### 1.2 Advantages and Contributions

Several advantages of our approach make the programming model especially appealing. The first is that the language runtime manages the policies, thus removing the need to trust the remaining application code of the web server—we need to trust only the policies. The second is that rather than simply preventing forbidden outputs, Jacqueline adapts program behavior to adhere to policies, simulating multiple executions of the program based on the possible ways the policies will need to be enforced. An additional benefit is that separating the policy specification and enforcement from the rest of the code decreases the amount of policy code needed. In Jacqueline, the programmer writes the policy associated with data once, and the runtime automatically enforces the policy. By contrast, in most other security-conscious web frameworks, the programmer must implement policies by writing checks and filters throughout the application and database code. In Figure 1 we compare the architecture of a Jacqueline program to that of a program using the popular Python web framework Django [2]. We show that in Jacqueline, 1) application and database code do not need to be trusted, 2) policies are localized, and 3) the size of policy code is smaller due to automatic policy enforcement.

We make the following contributions:

- **Policy-agnostic web programming paradigm.** We propose a paradigm for database-backed web applications that allows the programmer to specify information flow policies once and rely on the framework to customize program behavior. We implement this paradigm in the Jacqueline web framework.
- **Semantics and end-to-end guarantees.** We formalize the Jacqueline ORM using $\lambda^{JDB}$, an extension of Jeeves with relational operators, and prove end-to-end policy compliance. We show how the semantics corresponds to an implementation strategy using an unmodified relational database.
- **Optimization approach for web programs.** To avoid the exponential exploration of execution branches that may occur with Jeeves, we formalize an "Early Pruning" optimization and prove that it preserves policy compliance. We demonstrate that it decreases Jacqueline's overheads and is necessary for non-trivial computations involving sensitive values.
- **Demonstration of practical feasibility.** We implemented Jacqueline using an unmodified Python interpreter and unmodified SQL database. We demonstrate the expressiveness and performance of Jacqueline through several case studies, including a conference management system that we have deployed to run an academic workshop. We compare Django code with hand-implemented policies, showing that not only does Jacqueline reduce lines of policy code, but also that the automatic policy enforcement has reasonable overheads.

With Jacqueline, we demonstrate an approach for database-backed web application development that both provides strong theoretical guarantees and exhibits good performance in practice.

## 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 be able to see that she has an event 7pm next Tuesday, but not that it is a party. Everyone else may be able to see that there is a private event at Schloss Dagstuhl, but not the details of the event.

Enforcing policies in our calendar requires computations to be policy-aware. For instance, it is increasingly common for calendar systems to support queries such as "Who are my friends in Schloss Dagstuhl at 7pm Tuesday?" It is also common for the results of such a query to be broadcast to a set of users, each with their own permissions. Because a single fixed viewer may not be initially known and because viewers may be *computed from* sensitive values, we need information flow policies rather than access control policies. Using popular programming paradigms, the programmer must implement policies as repeated checks and filters across application and database code. Using Jacqueline, the programmer needs to provide only a single declarative specification of each security policy.

### 2.1 Policy-Agnostic Model-View-Controller Framework

Jacqueline is a *model-view-controller* (MVC) web framework where the *model* describes the data, the *view* describes the page layouts, and the *controller* describes computation over the data to produce views. Standard ORM frameworks abstract over interactions with an underlying database, allowing the programmer to specify *data schemas* for the model in the same language as the controller. Jacqueline additionally abstracts over the implementation of information flow policies. The Jacqueline runtime takes responsibility for tracking sensitive values and policies between applications and databases. Jacqueline produces outputs that adhere to the policies with end-to-end guarantees of policy compliance.

Jacqueline supports *policy-agnostic* application code and database queries. Once the programmer associates information flow policies with data fields, the rest of the program looks like a Django program. The programmer needs to be aware that policies may affect the values flowing through the program, *e.g.* defaults rather than sensitive values, but does not need to know the specifics of the policies. In Figure 2 we show the API for individual JacquelineModel data records and for sets of records JeevesQuerySet. The programmer may call these APIs exactly as they would call the corresponding Django APIs for Model and QuerySet. Note that both ORMs abstract over implicit joins from foreign keys.

```
class JeevesQuerySet(QuerySet):
  all()
  delete()
  filter(**kwargs)
  orderby(**kwargs)
  get(**kwargs)

class JacquelineModel(Model):
  create(*args, **kwargs)
  delete()
  save(*args, **kwargs)
```

**Figure 2:** The Jacqueline ORM API. The argument *args denotes an optional list of arguments. The argument **kwargs denotes an optional dictionary of arguments. The **filter** method takes, for instance, arguments for field equalities to filter on.

```
1  class Event(JacquelineModel):
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(JacquelineModel):
26   event = ForeignKey(Event, null=True)
27   guest = ForeignKey(UserProfile, null=True)
```

**Figure 3:** Jacqueline schema fragment for calendar events.

### 2.2 Schemas and Policies in Jacqueline

Continuing with our calendar example, we show a sample schema for the Event and EventGuest data objects in Figure 3. A Django schema is a Python class inheriting from Model with field names, field types, and optional methods. A Jacqueline schema is a Python class inheriting from JacquelineModel with field names, field types, optional policies, and optional methods. We define the Event class with fields name, location, description, and visibility, where visibility is the 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. The definition for EventGuest (line 25), with foreign keys to the Event and UserProfile (definition not shown) tables, is exactly as it would be in Django.

#### 2.2.1 Secret Values and Public Values

In Jacqueline, *sensitive values* encapsulate multiple views: a *secret* view available only to viewers with sufficient permissions and a *public* view available to all over views. The Jacqueline runtime simulates simultaneous executions on both views. Jacqueline guarantees that if a viewer does not have access to the secret view, the system will produce all outputs as if the secret view never existed. In Jacqueline, if a data field has a policy, the actual value is the secret view. Jacqueline requires the programmer to 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 information flow policy prohibits 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, so we could compute the public value using the row fields as well. The Jacqueline ORM uses naming conventions (*e.g.* the jacqueline_get_public prefix) to find the appropriate methods to compute public views.

Jacqueline allows sensitive values to behave as either the secret value or public value, depending on the viewing context (*i.e.* the user viewing a page). Computation sinks such as **print** take an additional (implicit) argument corresponding to the viewer. Jacqueline tracks the viewer, uses that along with the policies to determine the value to display. For instance, **print** carolParty.name displays "Carol's surprise party" to some viewers and "Private event" to others, depending on the policies. Note that the programmer does not need to designate the viewer, as this is something that the framework can track.

#### 2.2.2 Specifying Policies

The programmer specifies *information flow policies* that determine how sensitive values may flow through derived values. On line 21 we implement the information flow 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). The framework tracks the viewing context corresponding to the argument ctxt, for which the programmer determines the type and value. Here, ctxt corresponds to the user looking at the page.

Policies may contain arbitrary code: our policy queries the database, looking up in the EventGuest table (line 25) whether a given guest is associated with the event. Policies may depend on sensitive values: the EventGuest.guest field may have its own policies associated. Jacqueline enforces policies with respect to the row values at the time a value is created and the state of the system at the time of output. The jacqueline_restrict_event policy refers to the contents of the EventGuest table when a user views a page.

### 2.3 Policy-Agnostic Application Code and Database Queries

Jacqueline uses *faceted execution* [10] to simulate simultaneous multiple executions on the different *facets* of a sensitive value. The programmer calls create in Jacqueline the same way as in Django:

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

The Django ORM simply inserts the specified record into the database. In contrast, for the name field, the Jacqueline ORM 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 Jacqueline runtime maps labels to policies. For computation sinks such as **print**, the runtime assigns labels based on policies and the viewing context.

Once the programmer associates policies with sensitive data fields, the rest of the program may be *policy-agnostic* and look as the equivalent policy-free Django program would. The Jacqueline runtime evaluates faceted values by evaluating each of the facets. For

instance, evaluating "Alice's events: " + str(alice.events) yields the resulting faceted value guarded by the same label k:

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

Those with sufficient permissions, the 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, thus preventing *implicit flows*.

The Jacqueline ORM extends faceted execution to database queries. For instance, consider the query:

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

While the Django ORM simply issues the corresponding database query for matching Event rows, the Jacqueline ORM manipulates faceted values to prevent leaks of sensitive information. Recall that the location field of carolParty is ⟨k ? "Schloss Dagstuhl" : "Undisclosed location"⟩. 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 the following 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 branch. The call to save writes ⟨k ? carolPartyNew : carolParty⟩, where carolPartyNew is the updated value. If a viewer cannot see the actual value of carolParty.location, the viewer will also not be able to see the updated description field.

### 2.4 Computing Concrete Views

At computation sinks such as **print**, the runtime uses the viewing context and policies to produce concrete, non-faceted outputs. The runtime does this by producing a system of constraints on the labels. Printing carolParty.name to alice produces the following constraint:

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

The runtime evaluates this constraint in terms of the guest list at the time of output. Because policies are program functions, labels are the only free variables in the fully evaluated constraints. There is always a consistent assignment to the labels: since policies can only force labels to be False, assigning all labels to False is always valid.

The policy enforcement mechanism handles dependencies between policies, including mutual dependencies between policies and sensitive values. Suppose, for instance, that the policy on guest lists depended on the list itself:

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

This policy says that there must be an entry in the EventGuest table where the guest field is the viewer ctxt. This creates a circular dependency: the policy for the guest field depends on the value of the guest field. There are two valid outcomes for a viewer who has access: either the system shows the fields as empty or the system shows the actual fields. To handle situations like this, Jacqueline has a notion of *maximal functionality* and shows values unless policies require otherwise.

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. To handle these dependencies, a system must either 1) model these dependencies across the application code and queries, as we do, or 2) allow policies to be executed in a trusted "omniscient" context. Unfortunately, the latter is common practice.

## 3. Solution Overview

Austin *et al.*'s faceted semantics for Jeeves [10] provide strong guarantees, but they have the following problems for web applications. First of all, the guarantees only hold for programs that run entirely within a faceted Jeeves runtime, preventing Jeeves programs from interoperating with commodity databases. In addition, the Jeeves semantics may explore exponentially many possible execution paths.

We make policy-agnostic programming practical for web programs in the following ways:

- We extend Jeeves's faceted semantics and guarantees to include unmodified relational databases.
- We develop an optimization based on the observation that the viewing context is often predictable.

In this section, we describe our ORM framework by example, as well as the Early Pruning optimization. We formalize both in Section 4.

### 3.1 Executing Relational Queries with Facets

We designed the Jacqueline ORM to track sensitive values and policies through database queries when the database is not aware of sensitive values or policies. The ORM is able to do this by 1) using meta-data to represent faceted values in the database and 2) marshalling values to and from the database representation to the application-level faceted representation. Our representation allows us to use the following SQL queries unmodified: **CREATE**, **UPDATE**, **SELECT** ... **WHERE** ..., **JOIN**, and **ORDER BY**. Our solution works with any non-SQL relational database as well.

To describe our mapping, we first introduce the concept of a *faceted row*, a faceted value containing leaves that are non-faceted SQL records. (Any record containing faceted values may be rewritten to be of this form.) The Jacqueline ORM stores each faceted row as multiple SQL rows We map each faceted row to multiple SQL rows by augmenting records with meta-data columns corresponding to 1) an identifier jac_id, chosen uniquely for each faceted row, and 2) an identifier jac_vars describing which facet the SQL row corresponds to, using a string-encoded description of labels, for instance "k1=True,k2=True".

We provide examples of our mapping in Table 1, showing a version without policies on the left-hand side and a version with policies on the right-hand side. The faceted value ⟨k ? "Carol's surprise party" : "Private event"⟩ is stored as two rows in the Event table with the same jac_id of 1. The secret facet has a jac_vars value of "k=True" and the public facet has a jac_vars value of "k=False". For nested facets, we store more labels in the jac_vars column. For instance, the following faceted value gets encoded as three database rows where the jac_vars strings are "k1=True,k2=True", "k1=True,k2=False", and "k1=False":

```
⟨k1 ? ⟨k2 ? "Carol's surprise party" : "Party"⟩
    : "Private event"⟩
```

**Django**

```
CREATE TABLE Event COLUMNS (
  id INTEGER PRIMARY KEY,
  name VARCHAR(128),
  location VARCHAR(128),
);
```

| id | name | location |
|---|---|---|
| 1 | "Carol's ... party" | "Schloss Dagstuhl" |

**Jacqueline**

```
CREATE TABLE Event COLUMNS (
  id INTEGER PRIMARY KEY, # ignored
  name VARCHAR(128),
  location VARCHAR(128),
  jac_id INTEGER,
  jac_vars VARCHAR(128),
);
```

| id | name | location | jac_id | jac_vars |
|---|---|---|---|---|
| 1 | "Carol's ... party" | "Schloss Dagstuhl" | 1 | "x=True" |
| 2 | "Private event" | "Undisclosed location" | 1 | "x=False" |

**Table 1.** SQL code and example tables, with and without policies.

**Django 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';
```

**Jacqueline Query**

```
SELECT EventGuest.event, EventGuest.guest,
    EventGuest.jac_id, EventGuest.jac_vars,
    UserProfile.jac_vars
  FROM EventGuest
  JOIN UserProfile
    ON EventGuest.guest_id = UserProfile.jac_id
  WHERE UserProfile.name='Alice';
```

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

#### 3.1.1 Queries That Track Sensitive Values

Our representation of faceted rows allows the Jacqueline ORM to issue standard SQL queries for selections, projections, joins, and sorts. The ORM can simply rely on the correct marshalling of query results into faceted rows for tracking sensitive values and policies through queries. No modification of the database is necessary.

Our SQL representation of faceted values allows us to rely on faceted execution to lift the projection operator. Consider the query **SELECT** $*$ **from** Event **WHERE** location = "Schloss Dagstuhl" on the rows from Figure 1. Issuing the **SELECT**...**WHERE** on the augmented database will return only the rows that match:

| ... | location | jac_id | jac_vars |
|---|---|---|---|
| ... | "Schloss Dagstuhl" | 1 | "k=True" |

Reconstructing the facet structure yields the faceted value:

```
⟨ k ?
  [{..., location="Schloss Dagstuhl", ...}]
  : [] ⟩
```

Since the initial location field is guarded by label k, the results are also guarded by label k.

The Jacqueline tracks sensitive values and policies through joins by manipulating the meta-data appropriately. Rows from joins that occur based on sensitive values will be appropriately guarded by the appropriate path conditions. To prevent the join from leaking information, the ORM takes into account the jac_vars fields from both tables.[1] The ORM also ensures that foreign keys, references into another table, reference faceted rows with jac_id 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 jac_id rather than id. In the **SELECT** clause, we include the User.jac_vars as well as the EventGuest.jac_vars field.

[1] The ORM maintains the invariant that all tables have the correct jac_vars columns. We can migrate tables without these columns to comply.

The representation also allows us to take advantage of SQL's **ORDER BY** functionality for sorting. Suppose we had faceted records, each with a single field f, with values $\langle$a ? "Charlie" : "$***$"$\rangle$, $\langle$b ? "Bob" : "$***$"$\rangle$, and $\langle$c ? "Alice" : "$***$"$\rangle$. On the left we show the database representation and on the right we show the records ordered by the field f (where jid and jvars are abbreviations for jac_id and jac_vars, respectively):

| f | jid | jvars |
|---|---|---|
| "Charlie" | 0 | "a=True" |
| "$***$" | 0 | "a=False" |
| "Bob" | 1 | "b=True" |
| "$***$" | 1 | "b=False" |
| "Alice" | 2 | "c=True" |
| "$***$" | 2 | "c=False" |

| f | jid | jvars |
|---|---|---|
| "$***$" | 0 | "a=False" |
| "$***$" | 1 | "b=False" |
| "$***$" | 2 | "c=False" |
| "Alice" | 2 | "c=True" |
| "Bob" | 1 | "b=True" |
| "Charlie" | 0 | "a=True" |

We can use the standard SQL **ORDER BY** procedure without leaking information because the secret values are stored in different rows from the public values. The ORM is responsible for enforcing the policies so that, for instance, an output context with the permitted labels $\{a, \neg b, c\}$ would see ["$***$", "Alice", "Charlie"].

While the Jacqueline ORM can use SQL queries for selects, joins, and sorts, there is no equivalent aggregate functions, for instance **COUNT** or **SUM**. Using aggregate queries in the database could leak information, as they combine values across facets. Jacqueline performs these operations in memory using the Jeeves runtime.

#### 3.1.2 Updating Data and Policies

Jacqueline's representation of faceted rows ensures that any action involving a row facet is visible only to those with the appropriate permissions. The Jacqueline ORM implements save, updating meta-data and potentially deleting rows, such that all corresponding rows are updated appropriately. (The ORM computes default public values based on the state at the time of the save, using the entire row as the argument to the jacqueline_get_public function.) If the program invokes save in branches that depend on faceted values, Jacqueline creates facets that incorporate the path conditions.

Storing labels as meta-data makes it straightforward to 1) add policies to data that previously had no policies and 2) update policies on sensitive values. To add policies, the programmer needs to manipulate only the meta-data columns (jac_vars and jac_id). The programmer can add policies to legacy data by writing a database migration that adds the meta-data columns. To update policies using existing labels, the programmer can simply update the policies in the application code.

### 3.2 Early Pruning Optimization

With Jeeves, much of the overhead comes from executing with all possible views until a computation sink, as faceted values may grow exponentially in the number of labels. Whenever the viewer is not known, executing with all possible paths is necessary. This happens, for instance, when the program computes the viewer based on sensitive information, for instance when sending mail to all invitees of an event. Another case is when the program computes sensitive values to be written to the database, as the system usually cannot know the viewer of future database queries.

In many cases, however, a useful correctness-preserving optimization is to prune facets as soon as the runtime knows the viewer. As soon as the runtime knows the viewer, it can discard unnecessary facets. Doing this optimization involves being able to determine 1) the value of the viewing context and 2) that the state relevant to the policies will not change until output. In general, determining when we can perform this optimization requires non-trivial static analysis.

Two properties of web programs make this optimization feasible. First of all, the framework often knows the viewing context ahead of time, as it is often the session user. Secondly, computation sinks are easy to identify in model-view-controller web frameworks. The most common information-leaking computation sinks involving writing to the database and rendering a page. Most controller functions either read from the database or write to the database, but not both. This allows us to implement functionality that, for "get" requests, speculates on when the viewer is known, rolling back to the beginning of the controller function to perform faceted execution when the hypothesized viewer is incorrect. The Early Pruning optimization is especially helpful in the common case because many pages that require substantial computation do not also involve writes to the database. We can also perform an Early Pruning optimization for saves by adding extra code that limits the visibility of a save operation to certain viewers, provided that the programmer knows the viewers ahead of time.

## 4. Formal Semantics and Policy Compliance

In this section, we capture the key ideas underlying Jacqueline in an idealized core language called $\lambda^{JDB}$. We prove that $\lambda^{JDB}$ satisfies the key security property of termination-insensitive non-interference and policy compliance.When public values do not depend on secret values, $\lambda^{JDB}$ satisfies an end-to-end non-interference property.

### 4.1 Syntax and Formal Semantics

The language $\lambda^{JDB}$ extends the language $\lambda^{jeeves}$ [10] with support for databases, which we model as relational tables. Figure 4 summarizes the $\lambda^{JDB}$ syntax, with the constructs from $\lambda^{jeeves}$ marked in gray. The $\lambda^{jeeves}$ language, in turn, extends the standard imperative $\lambda$-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 $k$. For all other principals, the faceted expression behaves exactly like $e_L$.

The language $\lambda^{JDB}$ extends $\lambda^{jeeves}$ with support for databases, which we model as relational tables, 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, $\lambda^{JDB}$ 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* ($\pi_{\bar{i}}\ e$), which returns a new table containing columns $\bar{i}$ 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 $\bar{e}$ 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" $\forall \overline{A}, B.(B \to \overline{A} \to B) \to B \to \text{table } \overline{A} \to B$.

| | | |
|---|---|---|
| $e ::=$ | | *Term* |
| | $x$ | variable |
| | $c$ | constant |
| | $\lambda x.e$ | abstraction |
| | $e_1\ e_2$ | application |
| | ref $e$ | reference allocation |
| | $!e$ | dereference |
| | $e_1 := e_2$ | assignment |
| | $\langle k \ ? \ e_H : e_L \rangle$ | faceted expression |
| | label $k$ in $e$ | label declaration |
| | restrict$(k,e)$ | policy specification |
| | row $\bar{e}$ | create a table |
| | $\sigma_{i=j}\ e$ | select rows where fields are equal |
| | $\pi_{\bar{i}}\ e$ | project columns |
| | $e_1 \bowtie e_2$ | join or cross-product of tables |
| | $e_1 \cup e_2$ | union of tables |
| | fold $e_f\ e_p\ e_t$ | table fold |
| $S ::=$ | | *Statement* |
| | let $x = e$ in $S$ | let statement |
| | 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 4:** $\lambda^{JDB}$ syntax.

### 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 $\Sigma$ evaluate to $V$, producing a new store $\Sigma'$. 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 $k$. Association with $\neg k$ means the computation is visible only to principals *not* authorized to see $k$.

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 $(B, \bar{s})$ in the database includes a set of branches $B$ describing who can see that row. For example, the expression $\langle k$ ? row "Alice" "Smith" :

**Runtime Syntax**

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

**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 $\lambda^{jeeves}$ Subset** $\boxed{\Sigma, e \Downarrow_{pc} \Sigma', V}$

$$\frac{}{\Sigma, V \Downarrow_{pc} \Sigma, V} \quad \text{[F-VAL]}$$

$$\frac{\Sigma, e[x := V] \Downarrow_{pc} \Sigma', V'}{\Sigma, (\lambda x.e) \ V \Downarrow_{pc} \Sigma', V'} \quad \text{[F-APP]}$$

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

$$\frac{a \notin dom(\Sigma)}{\Sigma, !a \Downarrow_{pc} \Sigma, 0} \quad \text{[F-DEREF-NULL]}$$

$$\frac{k \notin pc \ \text{ and } \neg k \notin pc \qquad \Sigma, e_1 \Downarrow_{pc \cup \{k\}} \Sigma_1, V_1 \qquad \Sigma_1, e_2 \Downarrow_{pc \cup \{\neg k\}} \Sigma', V_2 \qquad V' = \langle\langle \ k \ ? \ V_1 : V_2 \rangle\rangle}{\Sigma, \langle k \ ? \ e_1 : e_2 \rangle \Downarrow_{pc} \Sigma', V'} \quad \text{[F-SPLIT]}$$

$$\frac{}{\Sigma, !a \Downarrow_{pc} \Sigma, \Sigma(a)} \quad \text{[F-DEREF]}$$

$$\frac{k \in pc \qquad \Sigma, e_1 \Downarrow_{pc} \Sigma', V}{\Sigma, \langle k \ ? \ e_1 : e_2 \rangle \Downarrow_{pc} \Sigma', V} \quad \text{[F-LEFT]}$$

$$\frac{\Sigma' = \Sigma[a := \langle\langle \ pc \ ? \ V : \Sigma(a) \rangle\rangle]}{\Sigma, a := V \Downarrow_{pc} \Sigma', V} \quad \text{[F-ASSIGN]}$$

$$\frac{\neg k \in pc \qquad \Sigma, e_2 \Downarrow_{pc} \Sigma', V}{\Sigma, \langle k \ ? \ e_1 : e_2 \rangle \Downarrow_{pc} \Sigma', V} \quad \text{[F-RIGHT]}$$

$$\frac{E \neq [] \wedge e \text{ not a value} \qquad \Sigma, e \Downarrow_{pc} \Sigma', V' \qquad \Sigma', E[V'] \Downarrow_{pc} \Sigma'', V''}{\Sigma, E[e] \Downarrow_{pc} \Sigma'', V''} \quad \text{[F-CTXT]}$$

$$\frac{\Sigma, \langle k \ ? \ S[V_H] : S[V_L] \rangle \Downarrow_{pc} \Sigma', V'}{\Sigma, S[\langle k \ ? \ V_H : V_L \rangle] \Downarrow_{pc} \Sigma', V'} \quad \text{[F-STRICT]}$$

**Evaluation with Relational Operations**

$$\frac{}{\Sigma, \text{row } \bar{s} \Downarrow_{pc} \Sigma, (\text{table } (\epsilon, \bar{s}))} \quad \text{[F-ROW]}$$

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

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

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

$$\frac{}{\Sigma, (\text{table } T_1) \cup (\text{table } T_2) \Downarrow_{pc} \Sigma, (\text{table } T_1.T_2)} \quad \text{[F-UNION]}$$

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

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

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

**Figure 5:** Faceted evaluation of $\lambda^{JDB}$.

row "Bob" "Jones"$\rangle$ evaluates to the following table [2]:

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

We do not model the facet identifier row jac_id. It is useful in the implementation but 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:

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

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. We extend this operator to sets of branches:

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

We show the faceted evaluation rules in Figure 5. The key rule is [F-SPLIT], describing how evaluation of a faceted expression $\langle k \ ? \ e_1 : e_2 \rangle$ involves evaluating the sub-expressions 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: that is, 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 $V_L$. Thus, for example, the evaluation of $1 + \langle k \ ? \ 2 : 3 \rangle$ reduces to the evaluation of $\langle k \ ? \ 1+2 : 1+3 \rangle$, where $S$ in this case is $1 + \ \bullet$ . The rules [F-SELECT], [F-SELECT], [F-PROJ], [F-JOIN], and [F-UNION] formalize the relational calculus operators on tables of faceted rows. These rules are mostly straightforward.

The rules for folding over tables 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. $\lambda^{JDB}$ expressions do not perform I/O, while $\lambda^{JDB}$ statements include the effectful construct print $\{e_v\}$ $e_r$ that prints expression $e_r$ under the policies and viewing context $e_v$. The $\lambda^{jeeves}$ semantics describes how, for printing, the runtime assigns labels based on the policies and viewers and projects a single facet based on the label assignment. The $\lambda^{jeeves}$ rules for declaring new labels and attaching policies to labels are in Appendix A.

[2] 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 cons unnaturally mixed values will get stuck.

### 4.3 End-to-End Policy Compliance

Austin *et al.* have proven policy compliance guarantees for $\lambda^{jeeves}$ [10], showing the faceted semantics have 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 this property extends to $\lambda^{JDB}$, yielding guarantees of end-to-end policy compliance for database-backed applications.

The proof of policy compliance involves extending the *projection* property of $\lambda^{jeeves}$. A key property of $\lambda^{jeeves}$ is that a single execution with faceted values *projects* to multiple different executions without faceted values. If a viewer has access only to the public facet of an expression, then faceted execution is output-equivalent to executing with only the public facet in the first place.

To prove this property, we first define what it means to be 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

$$\forall k \in B . k \in L$$
$$\forall \neg k \in B . k \notin L$$

We extend views to values:

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

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 $\lambda^{jeeves}$'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 $\lambda^{jeeves}$ [10]: data is revealed to an external observer only if it is allowed by the policy specified in the program.

### 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:

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

We prove the Projection Theorem holds with this extension.

# 5. Implementation

We implemented Jeeves as an embedding in Python and Jacqueline as an extension of the Django web framework [2]. Our code is available at [link to repository removed for double-blind reviewing].

## 5.1 Python Embedding of the Jeeves Runtime

Our embedding allows programmers to write programs that run according to Jeeves semantics simply by importing our library and annotating classes and functions with the `@jacqueline` decorator. The decorator indicates that the class or function is to execute according to the faceted semantics. The library exports functions for creating labels, creating sensitive values, attaching policies, and producing non-faceted values based on policies. 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 special `Facet` data type to store information about faceted values. During faceted execution, an object's fields might be faceted values, either faceted primitive values (*e.g.* **int**, **bool**) or faceted references to other objects. A field 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 overloads operators (except operator such as **in** and **and** that do not support overloading) and performs a dynamic source transformation using the macro library `MacroPy` [5]. The source transformation intercepts the standard evaluation of conditionals, loops, assignments, and function calls. The runtime also keeps track of path conditions corresponding to label assumptions in the current branch. Since the scope of a Python variable is determined by where it is assigned in the source code, the implementation handles local assignment by replacing a function's local scope with a special `Namespace` object that determines the scope of each local variable.

### 5.1.2 Evaluating Policies at Computation Sinks

The runtime keeps an environment that maps labels to policies for the purpose of using policies to de-facet values. Effectful computations take two arguments: the expression to show and an additional argument corresponding to the output context. If there are no mutual dependencies between policies and sensitive values, the runtime simply evaluates policies to determine label values. Otherwise, the runtime creates a system of constraints in order to find an assignment for label values consistent with the policies. The implementation 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 ORM

We implemented Jacqueline's ORM as an extension of Django's ORM. The Jacqueline ORM creates schemas with additional meta-data columns for keeping track of facets. All queries through the ORM manipulate the meta-data columns in addition to the actual columns. The ORM reconstructs facets from the meta-data. The ORM looks up policies from object schemas when reconstructing facets and adds the policies to the Jeeves runtime environment. We implement the Early Pruning optimization by reconstructing only the relevant facets when the runtime knows the viewer.

# 6. Jacqueline in Practice

To evaluate the expressiveness and performance of Jacqueline we built 1) a conference management system, 2) a health record manager, and 3) a course management system. We evaluate Jacqueline along the following dimensions:

- **Expressiveness.** We worked with two programmers who were not involved in Jacqueline development to ensure that Jacqueline provides a natural programming interface. One of the applications we built is a conference management system we have deployed to run a real workshop [1].
- **Code architecture.** We compare the implementation of the Jacqueline conference management system to an implementation of the same system in Django, as well as the HotCRP conference management system. We demonstrate that Jacqueline helps with both centralizing policies and with size of policy code.
- **Performance.** We demonstrate that Jacqueline can handle data from hundreds of simulated users in the database. We show that for representative actions, Jacqueline has comparable performance to the Django equivalent. 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.

## 6.1 Applications

We have developed the following applications using Jacqueline.

**Conference management system.** Our conference management system supports user registration, update of profile information, designation of roles (*i.e.* PC member), paper and review submission, and assignment of reviews. Users may be authors, PC members, or the PC chair; only the PC chair can designate users as PC members. The administrator specifies the PC chair when configuring the system. The PC chair has additional privileges: for instance, assigning reviewers to papers. Permissions depend on the current stage of the conference: submission, review, or decision.

**Health record manager.** We implemented a health record system based on a representative fragment of the privacy standards described in the Health Insurance Portability and Accountability Act (HIPAA) [11, 36]. The HIPAA standards describe how individuals and entities (such as hospitals and insurance companies) may view a patient's medical history depending on the information and the viewer's role. An example policy is that information about an individual's hospital visits is visible to the individual, the individual's insurance company, and to the site administrator. Policies may also depend on more stateful properties, for instance whether there exists a waiver permitting information release.

**Course manager.** Our course management tool allows instructors and students to organize assignments and submissions. Relying on Jacqueline to manage policies allows us to experiment with more complex policies than are normally in a course manager: for instance, stateful policies that depend on submission history or the activity of other students in the course.

## 6.2 Code Comparisons

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

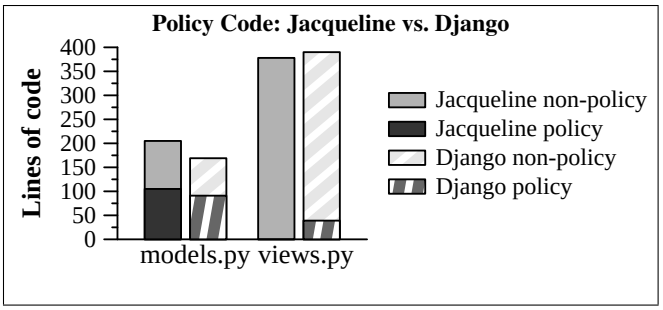


**Figure 6:** Distribution of policy code with Jacqueline and Django conference management systems.

**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.

#### 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 Jacqueline code requires additional imports and function decorators because we have implemented Jacqueline by extending Python and Django. (With our current implementation, each class and function executing according to the faceted semantics requires the `@jacqueline` decorator. Policies require additional decorators.)

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, the programmer needs to implement policies throughout the controller file `views.py` as well. These policies increase the overall code size. The Jacqueline implementation has 106 total lines of policy code, whereas the Django implementation has 130 lines. These additional lines of policy code manifest as repeated checks and filters across `views.py`. Thus, Django requires auditing of all of `models.py` and `views.py` (~575 total lines of code) to ensure policy compliance. In contrast, Jacqueline requires only auditing `models.py` (~200 lines of code), reducing the size of the application-specific trusted computing base by 65%.

#### 6.2.2 HotCRP

Policies and functionality are intertwined across the HotCRP conference management system [6], written mostly using PHP and SQL. There are 191 occurrences alone of checks for whether the viewer is the PC chair or has the appropriate conflict status, as well as dynamically generated SQL queries based on analogous conditional checks. The policy code is in at least 24 of the 82 files. A programmer needs to edit code across the system to add policies or fix bugs. The HotCRP bug we mentioned in the introduction involved 40 additions and 25 deletions, including adding checks in dynamically generated SQL, in multiple places across two files [4].

### 6.3 Performance

We evaluated the performance of our system on representative actions and stress tests compared to an implementation written using vanilla Django. We also evaluated the effectiveness of the Early Pruning optimization, demonstrating its necessity for nontrivial computations involving sensitive values.

We measured running 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 [3] for functional and load testing to time HTTP requests from another machine across the network. We ran all tests using the `--simple-fetch` option to exclude CSS and images. We averaged running times over 10 rapid sequential requests. We show results only from sequential requests because how well Jacqueline handles concurrent users compared to Django simply depends on the amount of available memory.

#### 6.3.1 Representative Actions

We measured the time it takes for our system to do view the profiles for a paper and user as there is more data in the database. We show these numbers, as well as comparisons to Django, in Figure 7. The time it takes to load these profiles is under two milliseconds and roughly equivalent to the time it takes to do the equivalent action in the Django implementation. For viewing a single paper, Jacqueline actually performs better than the Django implementation. This is because in a few cases, the implementation needs iterate over collections of data rows again in order to apply policy checks. In the Jacqueline implementation, the programmer can simply rely on the framework to attach the policies.

#### 6.3.2 Stress Tests

In Figure 8 we show results for showing an increasing number of papers and users for conference management systems implemented in Jacqueline and Django. In these tests, the system is resolving 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. In these results, Jacqueline has a 1.75x overhead for showing all papers. The overhead comes from Jacqueline fetching both versions of data from the database before resolving the policies. Integrating policies more deeply with the database could reduce this overhead. Note that there is no solver overhead, as there are no mutual dependencies between sensitive values and policies.

Results for the other case studies show similar promise for Jacqueline's ability to scale. In Figure 9 we show stress test data from our health record manager and course manager. Jacqueline resolves policies for rendering hundreds of data records in seconds. Most systems will not load over a thousand data rows at once, especially when each row value has its own privacy policy involving calls to the database. A more realistic website would load such a page in fragments and consolidate policies.

#### 6.3.3 Early Pruning Optimization

We found the Early Pruning optimization to be necessary when the program performs nontrivial computations over sensitive values. In the course manager stress test, the page that shows all courses also looks up the instructors for each course, leading to blowup: before the course is known, the system must look up all possible instructors. We show in Figure 10 how for just 8 randomly generated courses and instructors, the system begins to hit memory limits. Early Pruning makes it possible to write such programs in Jacqueline. As long as the computation to determine a viewer is simple, Early Pruning can simplify other computations after the viewer is known.

**CFM Stress Tests**

| View all papers | | | View all users | | |
|---|---|---|---|---|---|
| Papers | Jacq. | Django | Users | 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 |

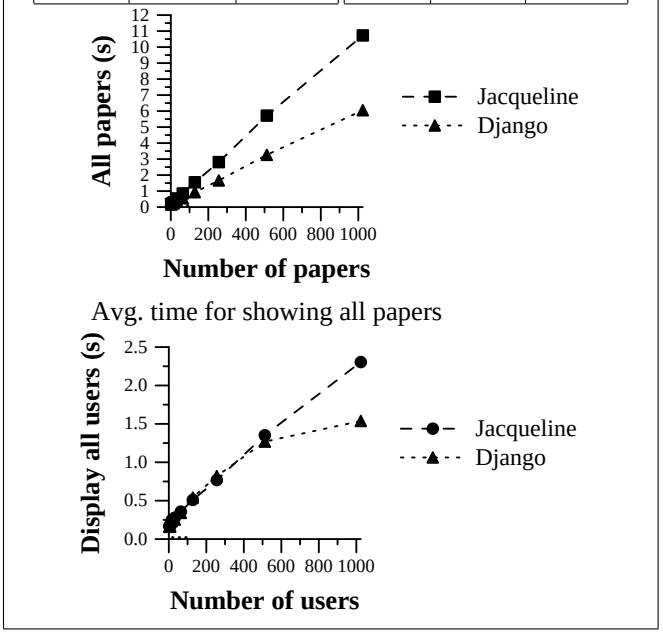


**Figure 8:** Times to view list of summary information for all papers and all users, in Jacqueline and Django.

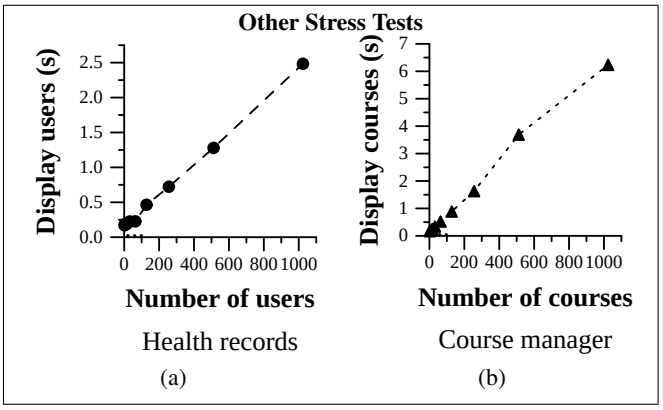


**Figure 9:** Jacqueline stress tests for other case studies.

**Showing All Courses, with and without Pruning**

| 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:** The course manager stress test performs well with the Early Pruning optimization and times out otherwise.

## 7. Related Work

There are many approaches that check programs, statically and dynamically, to prevent information leaks. Using these approaches, the programmer still needs to implement the policy checks and filters correctly across the program. Policy-agnostic solutions mitigate programmer burden by using the language runtime to customize program executions to adhere to policies. Approaches for checking information flow include the following:

- **Integrated query languages.** The SeLINQ system [38] builds on Cheney *et al.*'s theory of language-integrated query [19] to track information flow across the application and database in an embedded query language [38]. Lourenço and Caires have developed a type-based information flow analysis for tracking across database-backed applications [30].
- **Web frameworks.** Passe [13] dynamically analyzes applications to enforce policies about what information may be leaked from database queries. The Hails web framework [26] also separates out information flow policies from the rest of the program and enforces them using the LIO system for dynamic information flow controls [41]. The SIF web framework [21] uses a label-based approach and tracks all information-flow end-to-end to verify the correctness of programs with respect to stated policies. Ur/Web [20] uses static dependent types to check information flow properties in web programs.
- **Static, language-based checking.** Language-based approaches for verifying information flow security include Jif [34], Fabric [9, 29], Fine [17], F* [42], flow locks [14, 15]. IFDB checks information flow policies in databases [39].
- **Dynamic, systems-based checking.** Work on capabilities [12, 31] and dynamic, system-based informational flow control techniques [46] insert checking routines into programs.
- **Provenance-based checking.** Jacqueline's tracking of sensitive values in the database is also related to work in data provenance [7, 25, 37], especially recent work in provenance for security [8, 18] that uses the history of how values were computed for enforcing security properties.

Jacqueline differs from access control approaches in the same way: for instance, the Rubicon verification based on bounded model checking [35], the Margrave policy analyzer [24], and the Sunny approach [32] for model-based, event-driven programming.

Related security approaches include symbolic execution [28] and secure multi-execution [16, 22, 23], which executes a separate process for high- and low-confidentiality values to guarantee non-interference by construction. Faceted execution avoids overhead when code does not depend on confidential data. In addition, the policy-agnostic paradigm mitigates programmer burden by factoring

out policies from the rest of the code and supporting policies that may depend on sensitive values.

Sensitive values in Jeeves and Jacqueline are similar to variational data structures [43], values that encapsulate properties related to program customization. Aspect-oriented programming [27, 40] has similar goals of separating program concerns. Policy-agnostic programming goes beyond these approaches in customizing program behavior because the semantics allow properties of data to determine control flow.

## 8. Conclusions

Policy-agnostic programming prevents information leaks by reducing opportunity for programmer error. The approach mitigates programmer burden by allowing the programmer to separate the implementation of information flow policies from the rest of the functionality. Previous work on the Jeeves programming language [10, 45] defines a semantics for policy-agnostic programming. Unfortunately, Jeeves is unsuited for the scale of realistic web programs because 1) the guarantees do not extend when interoperating with commodity databases and 2) Jeeves has an expensive execution model that may explore exponentially many possible executions.

We present Jacqueline, a policy-agnostic web framework that supports realistic web applications. With Jacqueline, we extend the policy-agnostic model to work across applications and databases. The main contribution is an object-relational mapping (ORM) framework that enforces policies throughout application code as well as database queries. We model the ORM by extending the policy-agnostic semantics with relational operators and prove end-to-end policy compliance. To address the performance issues with Jeeves, we formalize an Early Pruning optimization approach. We demonstrate that this optimization not only helps Jacqueline run with reasonable–and often negligible–overheads, but is also necessary for nontrivial computations involving sensitive data.

We demonstrate that a policy-agnostic web programming paradigm reduces the amount of policy code and trusted computing base without sacrificing expressiveness or performance. By giving web frameworks more responsibility in managing sensitive data, we can allow programmers to focus on the novel parts of their applications, instead of implementing policies as repeated checks and filters across the program.

## References


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

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

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

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

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

[6] HotCRP. http://read.seas.harvard.edu/~kohler/hotcrp, accessed May 1, 2014.

[7] U. A. Acar, A. Ahmed, J. Cheney, and R. Perera. A core calculus for provenance. *CoRR*, abs/1310.6299, 2013.

[8] P. Anderson and J. Cheney. Toward provenance-based security for configuration languages. In *USENIX*, 2012.

[9] 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.

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

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

[12] A. Birgisson, A. Russo, and A. Sabelfeld. Capabilities for information flow. In *PLAS*. ACM, 2011.

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

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

[15] N. Broberg and D. Sands. Paralocks: Role-based information flow control and beyond. 45(1), 2010.

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

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

[18] J. Cheney. A formal framework for provenance security. In *CSF*. IEEE Computer Society.

[19] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. ICFP '13. ACM, 2013.

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

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

[22] 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.

[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] J. N. Foster, T. J. Green, and V. Tannen. Annotated XML: Queries and provenance. PODS '08. ACM, 2008.

[26] 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.

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

[28] 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.

[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] F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *SIGMOD*, 2009.

[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] R. Perera, U. A. Acar, J. Cheney, and P. B. Levy. Functional programs that explain their work. ICFP '12. ACM, 2012.

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

[39] D. Schultz and B. Liskov. IFDB: Decentralized information flow control for databases. In *EuroSys*, 2013.

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

[41] D. Stefan and D. Mazières. Building secure systems with LIO. In *PLAS*, 2014.

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

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

[44] 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].

[45] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. ACM, 2012.

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

# A. Rules from $\lambda^{jeeves}$

These rules from $\lambda^{jeeves}$ [10] descrbe 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 $\alpha$-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. It is then joined with the existing policy for $k$, ensuring that policies can only become more restrictive.

$$\frac{k' \, fresh \qquad \Sigma[k' := \lambda x.true], e[k := k'] \Downarrow_{pc} \Sigma', V}{\Sigma, \text{label } k \text{ in } e \Downarrow_{pc} \Sigma', V'} \quad \text{[F-LABEL]}$$

$$\frac{\Sigma, e \Downarrow_{pc} \Sigma_1, V \qquad V_p = \langle\langle pc \cup \{k\} \; ? \; V : \lambda x.true \rangle\rangle \qquad \Sigma' = \Sigma_1[k := \Sigma_1(k) \wedge_f V_p]}{\Sigma, \text{restrict}(k,e) \Downarrow_{pc} \Sigma', V} \quad \text{[F-RESTRICT]}$$

# B. Proof of Lemma 1

**Lemma 1** (A)**.**

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

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

*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)**.**

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

*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* $L(\Sigma) = L(\Sigma')$. *If pc is not visible to L and*

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

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

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,*

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

For our proof, we extend $L$ to project evaluation contexts, but they may project away the hole, and so map evaluation contexts to expressions, in which case filling the result is a no-op.

We also note 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 $L$. This property is captured in the following lemma.

**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.

- Cases [F-VAL], [F-DEREF], [F-DEREF-NULL], [F-ROW], [F-PROJECT], and [F-UNION] hold trivially.
- For case [F-SELECT], $e = \sigma_{i=j}\ (\text{table}\ T)$, so
  $$\Sigma, \sigma_{i=j}\ (\text{table}\ T) \Downarrow_{pc} \Sigma, (\text{table}\ T')$$
  where $T' = \{(B, \bar{s}) \mid s_i = s_j\}$.
  Therefore, 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 $T = \{B.B', \bar{s}.\bar{s'}) \mid (B, \bar{s}) \in T_1, (B', \bar{s'}) \in T_2\}$.
  $L(T) = \{(B.B', \bar{s}.\bar{s'}) \mid (B, \bar{s}) \in T_1, (B', \bar{s'}) \in T_2, B.B' \sim L\}$, so this case holds.
- For case [F-CTXT], $e = E[e']$. By the antecedents of this rule
  $$E \neq []$$
  $$e' \text{ not a value}$$
  $$\Sigma, e' \Downarrow_{pc} \Sigma_1, V'$$
  $$\Sigma_1, E[V'] \Downarrow_{pc} \Sigma', V$$
  Note that $L(E[V']) = L(E)[L(V')]$, etc., so by induction
  $$L(\Sigma), L(e') \Downarrow_{\emptyset} L(\Sigma_1), L(V')$$
  $$L(\Sigma_1), L(E)[L(V')] \Downarrow_{\emptyset} L(\Sigma'), L(V)$$
  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
    $$\Sigma, S[V_1] \Downarrow_{pc \cup \{k\}} \Sigma'', V''$$
    $$\Sigma'', S[V_2] \Downarrow_{pc \cup \{\neg k\}} \Sigma', V'''$$
    $$V = \langle\langle k \ ? \ V'' : V''' \rangle\rangle$$
    - If $k \in L$, then by induction $L(\Sigma), L(S[V_1]) \Downarrow_{\emptyset} L(\Sigma''), L(V'')$.
      $L(\Sigma'') = L(\Sigma')$ 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], $e = \text{fold}\ V_f\ V_p\ (\text{table}\ (B, \bar{s}).T)$.
  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

$$\begin{array}{c} k \in pc \\ \Sigma, e_1 \Downarrow_{pc} \Sigma', V \end{array}$$

  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

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

  - 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

$$\begin{array}{c} a \notin dom(\Sigma) \\ \Sigma' = \Sigma[a := \langle\langle pc\ ?\ V' : 0 \rangle\rangle] \end{array}$$

  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.

□