Jelf: a Web Framework for Automatic Enforcement of Privacy Policies

by

Travis J. Hance

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

Copyright 2014 Travis J. Hance. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Signature redacted

Author: .........................................................

Department of Electrical Engineering and Computer Science

May 2, 2014

Signature redacted

Certified by: ..................................................

Prof. Armando Solar-Lezama

Thesis Supervisor

Signature redacted

Accepted by: .............................................

Prof. Albert R. Meyer

Chairman, Masters of Engineering Thesis Committee
Jelf: A Web Framework for Automatic Privacy Policy Enforcement

by

Travis Hance

Submitted to the Department of Electrical Engineering and Computer Science on May 2, 2014, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

As people share more personal data on the web, it is increasingly important to correctly enforce policies on sensitive data. To address this problem, we have developed Jelf, a web framework that allows the programmer to separate the implementation of information flow policies from the rest of the functionality. The framework builds on previous work on the Jeeves programming language [28, 7] for automatically enforcing privacy policies. Our approach is novel in that it provides end-to-end guarantees by mediating interactions between the front-end, application, and database layers. The programmer only needs to specify information flow policies once for automatic enforcement across the web framework.

To build Jelf, we have integrated Jeeves with Python and extended the Django web framework. Jelf consists of a Django template layer, a Python Jeeves application layer, and a Jeeves-compatible database layer. Our Python integration does not require changing the Python interpreter: we use have implemented our solution as a dynamic source transformation and a runtime library. The programmer may use Jelf with Python 2.7 and a standard SQL database.

We have used Jelf to implement a conference management system. We describe the implementation and performance of this conference management system, as well as our experience using and running Jelf. Jelf policies comprise less than 3% of the code base and are concentrated in one place. We have deployed this system to collect submissions and reviews for an actual workshop.

Thesis Supervisor: Armando Solar-Lezama
Title: Associate Professor
Acknowledgments

I would like to thank Jean Yang for overseeing my work, for helping me write this thesis, and for her incredible moral support. I would also like to thank Armando Solar-Lezama for supervising the project and providing feedback on the writing.

I would also like to thank Cormac Flanagan and Thomas Austin for feedback.

More thanks go to Evan Pu for letting me use his desk while I worked and to my friends for moral support, especially Umaimah who initially encouraged me to pursue the M.Eng. at the start of the year.

Finally, many thanks go to my parents Sandra and Darrell, without whose wonderful support I could not have attended this institution.
Contents

1 Introduction 13

2 The Programmer’s View of Jelf 17
  2.1 Schemas in Jelf ........................................... 18
  2.2 Policy-Agnostic Queries .................................. 19
  2.3 Constructing Views ...................................... 21

3 Overview of Faceted Execution 23
  3.1 Policy Constructs ........................................ 23
  3.2 Faceted Execution ....................................... 24
  3.3 Producing Outputs ...................................... 26
  3.4 Supporting the Jelf Database Layer .................... 27

4 Jelf Implementation 29
  4.1 Python Jeeves Implementation .......................... 29
    4.1.1 Faceted Data ....................................... 29
    4.1.2 Python Source Transformation ........................ 30
    4.1.3 Handling Policies ................................... 32
    4.1.4 Faceted lists ...................................... 32
  4.2 Jelf Object Relational Mapping ........................ 34
    4.2.1 SQL Representation of Faceted Values ............... 35
    4.2.2 Querying in Jelf ................................... 36
    4.2.3 Connecting with the Application Layer ............. 37
4.2.4 Extending the Django ORM

5 Case Study: Conference Management System
5.1 Implementation
5.1.1 Comparison to HotCRP Implementation
5.2 Running the System
5.2.1 Overview of Representative Actions
5.2.2 Running Times
5.2.3 Deployment Experience

6 Discussion
6.1 Security Discussion
6.2 Decision to Use Python
6.3 Decision to Use Django

7 Related Work

8 Future Work

9 Conclusions
List of Figures

2-1 Sample schema policy in Jelf. ................................. 18

4-1 Python Jeeves Transformation ............................... 32
4-2 Possible representations of a faceted list ................... 33
List of Tables

4.1 Standard ORM vs. Jelf ORM ........................................ 34
4.2 Example of SQL queries made by Jelf. ........................ 37
5.1 Lines of code and policy in our Jelf conference management system. . 39
5.2 Lines of code and estimated lines of policy in HotCRP. . . . . . . . 41
5.3 Running times for CMS with hard-coded policies .......... 42
5.4 Running times for CMS with Jeeves policies ................. 42
Chapter 1

Introduction

As people put more of their lives online, programmers are under increasing pressure to ensure that web applications correctly implement the expected privacy policies. Enforcing these policies in web applications can be particularly difficult because the programmer must reason about how policies and data interact globally across the front-end, application, and database layers.

For example, consider an application involving socially shared user locations. Even enforcing a seemingly simple policy that a user’s fine-grained GPS location should only flow to “friends” is nontrivial. In such systems, users are often able to access sensitive values not just directly, for instance by viewing a user profile, but also indirectly, for instance by performing queries over the data. On the social networking website Facebook, users can tag posts and photographs with locations that other users can then search over using Graph Search. Policy enforcement becomes even more complicated when the results of these searches may then be re-shared.

Previous work presented Jeeves [28, 7], a language-based approach for mitigating programmer burden in enforcing policies. Jeeves allows the programmer to separately implement information policies from other functionality while relying on the runtime to produce suitable outputs. In Jeeves, the programmer defines different views for each sensitive value, along with policies that establish which views are allowed to flow to which contexts. For instance, in the location example, the programmer can define a high-confidentiality view that is the precise GPS location, a low-confidentiality view
that shows the country, and a policy that the high-confidentiality view may only flow to “friends.” The rest of a Jeeves program may be policy-agnostic: the programmer needs only to be aware that different views may flow through the program, but not the policies determining when a view may be shown. Jeeves has a faceted execution model that simulates simultaneous executions on the views and uses the policies to determine which result to show.

We describe Jelf, a Python-based web framework that applies the policy-agnostic programming model to mainstream web development. Jelf addresses the complexities of handling web applications that span multiple runtimes and have persistent state. In Jelf, the programmer associates policies and low-confidentiality views with data once, as part of the data schemas. The framework guarantees that the policies are satisfied by the end-to-end behavior of the system. Jelf does more than just check for bugs: the framework adapts program behavior to produce outputs consistent with the viewer’s permissions.

Our implementation uses Python’s meta-programming capabilities to dynamically source-transform Python so that it executes according to Jeeves’s faceted semantics. To ensure that the guarantees are preserved even in the presence of persistent state, we have extended the Django Python web framework [1] to implement a facet-aware object-relational mapping (ORM) layer. The Jelf ORM mediates access to the database and ensures that privacy is guaranteed even across sessions. By using these mechanisms, Jelf provides the benefits of policy-agnostic programming in the context of a familiar programming model: Jelf programs look like Django programs where data schemas have additional declarations for information flow policies. Jelf requires no extensions to the Python interpreter and works with a standard SQL database.

To evaluate the usability and performance of Jelf, we have implemented a simple conference management system. We describe its implementation and compare it to the implementation of the popular HotCRP conference management system [3, 17]. While HotCRP has policies intertwined with code throughout the code base, a Jelf application has policies contained within a single file. Instead of having to trust the entire code base regarding policy implementation, we only have to trust a concentrated
amount of code in Jelf. We have deployed the system to run a small workshop.

Our main contributions are as follows:

- We present Jelf, a web framework that automatically enforces information flow policies with end-to-end guarantees.

- We describe an implementation of the Jelf application layer that the programmer can use with an unmodified Python interpreter.

- We describe an object-relational mapping that handles policy enforcement throughout queries and across sessions. We have implemented this as an extension of the Django web framework.

- To demonstrate the feasibility of Jelf, we implemented a conference management system that we have deployed for a small workshop.
Chapter 2

The Programmer’s View of Jelf

Standard web frameworks make it easier for the programmer to manipulate data across the frontend, application and database layers of a web application. Key to this functionality is an object-relational mapping (ORM) layer that mediates access between the application and the underlying data-store—in this case, a SQL database. The ORM layer gives the programmer the illusion of accessing and manipulating persistent objects, while the framework takes responsibility for backing those objects into the database. In such ORMs, the programmer provides a data schema that describes the record fields.

Jelf extends traditional web framework capabilities by allowing programmers to specify different possible application behaviors based on information flow policies. In particular, Jelf extends the standard Django ORM layer to support information flow policies. In an ORM, schemas provide a centralized description of all the persistent data in the application. In Jelf, the programmer may extend the schema with centralized information flow policies. The programmer may define a public view for each data element, along with a policy that defines who should have access to the actual value versus the public view. To determine which version flows to the viewer, the runtime evaluates the policies under output contexts describing the viewer and other application-relevant information.
class UserProfile(Model):
    username = CharField()
    location = CharField()

    # more fields here
    # ...

@public_value("location")
def jeeves_get_public_location(user):
    return getCountry(user.location)

@label_for("lc", "location")
@jeeves
def jeeves_restrict_loc(user, ctxt):
    return (user.isFriends(ctxt) and
            user.location.isNear(ctxt.location))

Figure 2-1: Sample schema policy in Jelf.

2.1 Schemas in Jelf

To illustrate this with an example, consider a system like the Glympse mobile application [2], which allows users to share their GPS coordinates with specific users for specific periods of time. In our example, the main data type will be the UserProfile containing all the information about a user, including his or her location. Let us assume that the following policy protects user locations: a user’s exact location should only be visible to her friends who are nearby (according to some distance threshold); everyone else should only be allowed to know the country where she is currently located.

We show the UserProfile schema in Figure 2-1. Through line 6, we have standard Django-style schema declaration code describing the different fields in the UserProfile. It gets more interesting on line 8, where we start adding policy code that only the Jelf ORM can handle. In Jelf, the programmer defines public views of the fields by methods with the @public_value decorator. In line 8 of our example, we define the method jeeves_get_public_location for specifying the public view of the location which, according to our policy, is the country where the user is currently located. The @label_for decorator marks that jeeves_restrict_loc defines a policy protecting the field location. The first argument, "lc", is an arbitrary identifier called the label class, whose use we describe in Section 4.2.3. This policy returns True only if the user is friends with the user described by the
context, and the locations of the two users are near according to the \texttt{isNear} method. Here, we protect only a single field, but a policy is also capable of protecting multiple fields. Jelf stores these \texttt{UserProfile} records with additional fields in the database to associate them with these policies.

2.2 Policy-Agnostic Queries

For the rest of the application, the programmer only needs to take into account the public values that may flow through the program, but not the specific policies. The runtime system is responsible for ensuring that every output is consistent with the policies. Jelf stores objects in the database in a facet-aware manner, augmenting tables with additional columns corresponding to labels and using the schemas to attach policies to the labels.

For example, from within the application code, the programmer can request from the ORM layer the profile of user Joe, with username "joe", and output Joe's location as follows:

```python
joe = UserProfile.objects.get(
    username="joe"
)[0]
print "Joe's location is " + joe.location
```

If this code is executed in a context where the viewer is a friend of Joe and is nearby, the output will show Joe's precise location:

"Joe's location is 13 1st St. Ada OK 74821"

If, on the other hand, the viewer does not satisfy the criteria dictated by the policy, the output will be:

"Joe's location is United States"

Jelf can prevent implicit information leaks involving conditional logic on sensitive values. Consider, for example, the following code:

```python
targetLoc = "13 1st St. Ada OK 74821"
usersInLocation = UserProfile.objects.get(
    location = targetLoc)
```
Jelf ensures that a friend of Joe who is nearby will see Joe in the output from this code, but a viewer who does not fit these criteria will not see Joe in the output. To such a viewer, Joe's location is "United States".

Jelf preserves privacy guarantees across sessions. For example, Jelf may take protected data out of the database, perform a computation on it, and store that data back into the database with policies intact and the guarantee that data marked as private can have no influence on any value displayed to that user. For example, suppose we had a model `LocationData`, and we executed

```python
locData.userCount = len(usersInLocation)
locData.save()
```

The count `len(usersInLocation)` is appropriately protected. If we later retrieve this same `LocationData` record and display the `userCount` value to a user who is not a friend of Joe, the user will see a value consistent with the fact that from the point of view of that user, Joe’s location is "United States".

If the policies change, the programmer only needs to change the schema, and the system will now behave according to the new policy. For example, if the programmer decides that low-privilege viewer should see the location as "undisclosed", a simple change to the public view is sufficient to ensure that the system now behaves according to the new policy:

```python
@public_value("location")
def jeeves_get_public_location(user):
    return "undisclosed"
```

The programmer can just as easily change the policy defined in method `jeeves_restrict_up`.

The Jelf ORM supports `SELECT..WHERE`, `JOIN`, and `ORDER BY`. We have wrapped around a standard SQL database in such a way that we can migrate a legacy database to a faceted Jelf database by adding Jelf-related columns to each table and adding rows corresponding to low-confidentiality values.
2.3 Constructing Views

Jelf uses the Jeeves programming language's faceted execution model, simulating multiple simultaneous executions on sensitive and public values. The runtime keeps track of the possible outputs and then computes the value to output based on the viewer. As we said before, Jelf tracks the viewer and additional contextual information that may be required by the policies.

The security guarantees offered by Jeeves are quite strong: as long as the programmer associates the correct policies with schemas and makes queries through the Jelf ORM, any user that interacts with the application will only see outputs that are consistent with the views allowed by the policies.
Chapter 3

Overview of Faceted Execution

We base Jelf’s execution model on the faceted execution approach for policy-agnostic programming presented in the Jeeves language [28, 7]. In Jeeves, the programmer specifies different views for sensitive values by creating faceted values, which may then be associated with policies. The Jeeves runtime simulates simultaneous executions on different facets and using the policies to project a single facet for each output.

With Jelf, we have integrated the faceted execution approach across the web framework. We have implemented the Jelf application layer as an implementation of Jeeves in Python. The Jelf database layer uses this implementation to support faceted database data items and queries. In this chapter, we describe the Jeeves constructs `mkLabel`, `restrict`, and `mkSensitive` for faceted execution. Note that Jelf programmers do not need to use these constructs directly, as Jelf uses them internally to manage sensitive data and policies.

3.1 Policy Constructs

Policy constructs in Jeeves allow the programmer to associate any sensitive value with different views and policies about the disclosure of those views. In Jeeves, the programmer uses labels to associate sensitive values with policies. In the Python embedding of Jeeves, the programmer creates a label by calling a `mkLabel` function:

```python
a = mkLabel()
```
The variable \( a \) becomes associated with a fresh label that can then be used in creating sensitive values:

\[
x = \text{mkSensitive}(a, 42, 0)
\]

This creates a faceted expression \((a ? 42 : 0)\), with label \( a \), *high-confidentiality facet* 42, and *low-confidentiality facet* 0. The label \( a \) guards access to the high-confidentiality facet: 42 may only flow to the viewer if the policies associated with \( a \) permit. Labels take on the values \{high, low\} corresponding to the high- and low-confidentiality facets. The runtime assigns values to labels based on the output channel. Using labels gives the programmer the flexibility to associate multiple policies with the same value or multiple values with the same policy.

In Python Jeeves, the programmer introduces policies through the \texttt{restrict} construct:

```python
restrict(a,
    lambda viewer: bob.isFriends(viewer))
restrict(a,
    lambda viewer:
        bob.location.isNear(viewer.location))
```

Policies take a label and a *policy function* that takes an output channel and returns a Boolean value corresponding to whether the given label is permitted to be high. To process these calls to \texttt{restrict}, the runtime environment maps label \( a \) to the conjunction of these policies. To produce concrete results (say, from outputting), the runtime uses the policy functions to determine what to output.

### 3.2 Faceted Execution

Faceted execution simulates multiple simultaneous executions on faceted values. The runtime executes operations involving a faceted value \((a ? 42 : 0)\) by executing the operation on both the high-confidentiality facet 42 and on the low-confidentiality facet 0 and creating a faceted result guarded by the same label \( a \). For instance, evaluation of the expression \( 3 + (a ? 42 : 0) \) produces the resulting facet \((a ? 45 : 3)\). In general, calling a function \( f \) on a faceted expression \((x ? y : z)\) yields the result of evaluating \((x ? f(y) : f(z))\).
Note that \( y \) and \( z \) may contain faceted values; the function calls get pushed down to the facet "leaves."

The two interesting cases in faceted execution involves conditionals and assignments. For conditionals, faceted execution prevents *implicit flows*, information leaks resulting from conditional checks involving sensitive values. For instance, the following code could accidentally leak information about sensitive value \( x \):

```python
if x == 42:
    return "42"
else:
    return "something"
```

With facets, however, the runtime associates *any* computation on a sensitive value with the associated policies, guaranteeing that implicit flows cannot happen. Suppose variable \( x \) has value \( \langle a ? 42 : 0 \rangle \). The condition \( x == 42 \) evaluates to the faceted expression \( \langle a ? \text{True} : \text{False} \rangle \), meaning that the condition is \text{True} if label \( a \) is high and \text{False} otherwise. The Jeeves runtime evaluates faceted conditions by evaluating each facet as the condition and putting the results into a facet under the appropriate label. In this case, the function would return \( \langle a ? "42" : "something" \rangle \).

Faceted execution also handles the case where the program may have implicit leaks through assignments under sensitive conditions. Suppose \( x \) is a sensitive value in the following code:

```python
if x == 42:
    msg = "42"
else:
    msg = "something"
```

In this case, the program could leak information if the viewer can see that \( \text{msg} \) has been set to "42" despite the viewer not being able to see the high-confidence facet of \( x \). To prevent this, the Jeeves runtime keeps track of labels corresponding to path conditions while evaluating conditionals. The runtime takes the path conditions into account when performing assignments, so in this case \( \text{msg} \) would be assigned the value \( \langle a ? "42" : "something" \rangle \).
3.3 Producing Outputs

Effectful computations such as `print` take an additional output channel argument so the runtime can determine the appropriate facet to project:

\[ \text{print } \{ \text{alice} \} \text{ msg} \]

The Jeeves runtime evaluates `msg` to a faceted tree of values where each of the leaves is guarded by a set of label assignments and associated with a different possible output. In this case, we have \( (a \ ? \ "42" : \ "something") \).

In order to decide which facet leaf to output, the runtime applies the output channel `alice` to the policy functions in order to produce a system of constraints determining when each label is permitted `high`. From the policies in this example, we have the following two constraints determining when label `a` must be `low`:

\[ (a == \text{high}) \Rightarrow \\
(bob.\text{isFriends}(bob) \land \\
\quad bob.\text{location}.\text{isNear}(alice.\text{location})) \]

The runtime evaluates the methods `isFriends` and `isNear` with respect to the program state at the time of output; the result of evaluation may be a faceted value. Note that even though policies may refer to arbitrary program code, they are declarative in that the runtime treats them as constraints. Thus, the programmer can layer policies rather than having blocks of nested conditionals. In the resulting constraints, the labels are the only free variables and the system assumes a label may be `high` unless a policy constrains it to be `low`.

Constraints may have circular dependencies between labels that arise from cases where a policy depends on the sensitive value it protects. For instance, the policy for a location can depend on the distance between the viewer and the location or the policy for a guest list can depend on membership in the list. Jeeves handles such dependencies correctly with respect to the policies.
3.4 Supporting the Jelf Database Layer

The Jelf object-relational mapping conceptually allows the programmer to store data items as faceted objects. The Jelf database layer augments tables with additional fields to store labels persistently and uses the schema methods to construct low-confidentiality facets and policies, using the \texttt{mkLabel}, \texttt{mkSensitive}, and \texttt{restrict} constructs. Each Jelf database row has its own fresh labels; the programmer can specify in the schemas whether to share labels between columns in a row. In Jelf, the framework is responsible for authentication and keeping track of users so the programmer does not have to manually provide contexts.
Chapter 4

Jelf Implementation

We have implemented Jelf as a web framework in Python, extending the Django web framework [1] on top of a standard SQL database. Our code is available at https://github.com/jeanqasaur/jeeves. We first describe the Python Jeeves implementation and then describe how we use this to implement the Jelf database layer.

4.1 Python Jeeves Implementation

Our Python embedding of Jeeves consists of 1) the definition of a data type for faceted objects, 2) overloading Python operators to support evaluation of faceted objects, and 3) a source transformation to handle Python functionality that cannot be overloaded (e.g., assignments). In Python Jeeves, the programmer can write code that looks like Python code but operates according to the Jeeves semantics simply by importing the JeevesLib library and annotating classes and functions with @jeeves.

4.1.1 Faceted Data

Python represents data as objects, so embedding Jeeves in Python involves figuring out how to represent and execute with faceted objects. The problem of executing faceted data has two parts: marking data as faceted and performing the faceted execution.
Our Jeeves implementation defines a `Facet` class which stores a label and two child values. A Jeeves faceted value is represented as a tree of `Facet` objects whose leaves are the possible facets.

In Python, an object is effectively a dictionary mapping field names to values. During faceted execution, an object’s fields might be faceted values, either faceted primitive values like `ints` or `bools` or faceted references to other objects. In some cases, a field may even exist only in some execution paths. In this case, we use a special object `Unassigned()` to mark that the field does not actually exist. For example, a field might have the faceted value \((x \ ? 0 : \text{Unassigned()})\). This means the field has value 0 under \(x = \text{high}\) and does not exist under \(x = \text{low}\).

Any code run with faceted objects needs to support the faceted execution. The programmer uses the `@jeeves` decorator which performs a source code transformation, adding support for the faceted values. All methods of all objects that the programmer uses need to be facet-aware. Naturally, instances of any Jelf model are facet-aware. A programmer-defined object can be made facet-aware with the `@jeeves` decorator. Unfortunately, it is not possible to use objects from arbitrary libraries without working to make them support Jeeves. To do so, the programmer needs to modify the source code, adding `@jeeves` annotations. Of course, the code involved would have to be within the Python subset that Jeeves supports.

4.1.2 Python Source Transformation

The `@jeeves` macro performs dynamic source transformation to handle the Python constructs that we cannot handle with overloading.

First of all, we need the faceted values and objects to evaluate correctly under all Python operators. For example, \(a+b\) needs to evaluate correctly when either \(a\) or \(b\) is a faceted value. For many operators such as `+`, this can be done by implementing methods such as `_add_` and `__radd__`. If one of \(a\) or \(b\) is a faceted value, evaluation of \(a+b\) will call special code to handle the faceted semantics. However, this approach does not generalize to operators like `in` or `and`, which Python does not allow us to overload. Therefore, the source transformation transforms expressions like `a and b` into calls to
Jeeves library functions, like \texttt{jand(lambda : a, lambda : b)}. (It is important to lazily evaluate \texttt{b} with the lambda, due to python’s short-circuiting behavior with the \texttt{and} operator.)

\textit{Conditionals} (including the inline if-else “ternary operator”) need special support. For example, consider an if-statement like the one in Figure 4-1(a). Suppose \texttt{cond} is a faceted value, say \texttt{(x ? True : False)}. Then, the resulting value of \texttt{a} should depend on the label \texttt{x}. Somehow, we need to change the semantics of the if-statement so that it executes the body with a \textit{path condition}, specifying which labels should be assumed to be \texttt{high} or \texttt{low} for the current execution. In this example, it should evaluate \texttt{a = 0} with the path condition \texttt{x = high} and \texttt{a = 1} with the path condition \texttt{x = low}. We replace if-statements with the Jeeves library function \texttt{jif}, as in Figure 4-1(a), which calls the if-body and the else-body with the appropriate path-conditions. We handle for-loops in a similar way, as shown in Figure 4-1(b).

The semantics of the \textit{assignment operator} need to be changed because an assignment needs to take into account the path conditions as well. Note that we can overload assignment to object fields by implementing an object’s \texttt{__setattr__} method. However, assignments to local variables are more complicated, since ordinary, local variable assignment cannot be overloaded. Therefore, we simply replace a function’s local scope with a \texttt{Namespace} object, which can implement the \texttt{__setattr__} method. This is shown in Figure 4-1(c). Our source transformation determines to which scope each local variable belongs and makes it an attribute of the corresponding \texttt{Namespace}.

To implement this source transformation, we use the library MacroPy, which enables source transformation macros. MacroPy allows us to define the \texttt{@jeeves} source transformation. When a source file is imported, MacroPy’s import hook searches the file for the \texttt{@jeeves} decorator and applies our source transformation. MacroPy decorators like our \texttt{@jeeves} are therefore \textit{not} traditional Python decorators, which execute at runtime rather than import time. Our transformation supports a subset of Python’s syntax which includes if-statements, for-loops, and return statements.
<table>
<thead>
<tr>
<th>Original code</th>
<th>Transformed code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if cond:</td>
<td>def if_body():</td>
</tr>
<tr>
<td>a = 0</td>
<td>a = 0</td>
</tr>
<tr>
<td>else:</td>
<td>def else_body():</td>
</tr>
<tr>
<td>a = 1</td>
<td>a = 1</td>
</tr>
<tr>
<td></td>
<td>jif (cond, if_body,</td>
</tr>
<tr>
<td></td>
<td>else_body)</td>
</tr>
<tr>
<td>(a)</td>
<td></td>
</tr>
<tr>
<td>for e in elems:</td>
<td>def body(e):</td>
</tr>
<tr>
<td></td>
<td>do_stuff(e)</td>
</tr>
<tr>
<td></td>
<td>jfor (elems, body)</td>
</tr>
<tr>
<td>(b)</td>
<td></td>
</tr>
<tr>
<td>def f():</td>
<td>def f():</td>
</tr>
<tr>
<td>a = 5</td>
<td>ns = Namespace()</td>
</tr>
<tr>
<td>b = 7</td>
<td>ns.a = 5</td>
</tr>
<tr>
<td>c = a</td>
<td>ns.b = 7</td>
</tr>
<tr>
<td></td>
<td>ns.c = ns.a</td>
</tr>
<tr>
<td>(c)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4-1: How we transform (a) if-statements, (b) for loops, and (c) local variables.

4.1.3 Handling Policies

The Python Jeeves runtime stores a policy environment. To resolve calls to `print` and to produce concrete Jelf template values, the runtime has a procedure that collects policies and applies the output channel in order to produce constraints. Our implementation uses the Python bindings of the Z3 SMT solver [22] to resolve constraints on the Boolean labels. For each new output context, we use a new Z3 context and generate fresh constraints. Note that while Z3 is a mature SMT solver with sophisticated theories, we only use the SAT subset to check the satisfiability of Boolean label assignments. Previous work describes how Jeeves prioritizes label assignments assigning more labels to high [28]. The Jeeves language design guarantees that Z3 will always find a solution consistent with the constraints: assigning all labels to low always satisfies the constraints.

4.1.4 Faceted lists

Larger lists tend to depend on many more label variables, resulting in exponentially large facet trees. However, if we choose a better way to represent our list, we can still get good performance in the common cases.
Consider a faceted list such as the one in Figure 4-2(a). Naively, we could represent it as a facet tree like anything else. Note, however, that this example has an interesting property: each possible list is a subsequence of [0,1,2], but the 0 is missing when x is low, and the 2 is missing when y is low. Thus we could represent it instead as in Figure 4-2(b): we maintain a list of values and label conditions. When we concretize this list, we return the subsequence of values that correspond to conditions which evaluate to true.

This representation is even more appealing when we consider the list from the perspective of constructing lists via `cons` as is done in many functional languages. This is shown in Figure 4-2(c).

This representation is appropriate for most of the use-cases that occur in the Jelf code we wrote. Very often we have a list of values, where we want each value to be in the list if it is a public value and not in the list otherwise. Also, this enables us to perform any operation corresponding to a `map` or a `filter` (e.g., a Python list comprehension) in linear time.

Unfortunately, this does not solve the exponential blow-up problem when we try to do a calculation, such as a sum, that depends on all elements of the list. However, this exponential blow-up only depends on the number of labels, so one way to mitigate this is to use fewer labels. If that is impossible, the only solution would be to do the computation after concretization, if the policies do not depend on the result.
### 4.2 Jelf Object Relational Mapping

The Jelf object relational mapping allows the programmer to define the types of data objects and their fields, along with 1) an optional public (low-confidentiality) value for each field and 2) an optional information flow policy associated with when the actual value of the field may be revealed, as opposed to the public value. The Jelf ORM augments tables with additional facet-related metadata and produces faceted query results associated with the appropriate policies. It supports SELECT, WHERE, JOIN, and ORDER BY.

In Table 4.1 we provide a high-level, side-by-side comparison of how the Jelf ORM differs from a standard non-faceted ORM. On the left-hand side we show for a standard Python ORM (here, Django) versus our Jelf ORM.

### Table 4.1: Python object code, SQL code, and sample data in a standard Python ORM (here, Django) versus our Jelf ORM.
Python ORM what the programmer writes, the SQL table schemas that the ORM translates this into, and what the data looks like in the database. We show the Jelf faceted version on the right-hand side. In a standard ORM, the programmer defines an object with fields and types. The ORM is responsible for creating the appropriate corresponding SQL table and performing updates and queries on that table correctly. In the Jelf ORM, the programmer specifies public values and policies along with the field definitions. Along with programmer-defined fields, the Jelf ORM also inserts the additional fields jeeves_id and jeeves_vars to keep track of labels and their values. The Jelf ORM potentially creates multiple rows for every new object. Typically, it creates two rows: one corresponding to the new value (the high-confidentiality facet) and one corresponding to the default public value (the low-confidentiality facet).

4.2.1 SQL Representation of Faceted Values

We now describe in more detail how we represent faceted values in a standard SQL database. In a standard SQL database, the programmer defines schemas that determine the columns in each record. A table consists of all records with the same schema. A record corresponds to a row in a database table. Rows may have a primary key for uniquely identifying each record. Jelf uses a primary key for every table, as does vanilla Django. Some columns of a row may also be foreign keys: a foreign key in one table points to another record (possibly in a different table) by primary key.

The Jelf ORM allows us to conceptualize every value stored in a table as a faceted value. The challenge in mapping facets onto the database involves keeping track of high- and low-confidentiality facets of a faceted value, especially when one of these facets may itself contain a faceted value. Jelf stores each conceptual row in a table as multiple actual rows. The actual rows together represent a conceptual row whose fields may be faceted. To keep track of these rows, we add a few new columns. One is the facet key jeeves_id which identifies multiple actual rows as being in the same conceptual row. Like the primary key, the jeeves_id is a value like any other value. It is chosen randomly for each conceptual row and always public. The other is the labels column jeeves_vars, which stores a string-encoded description of the labels and values.
under which the actual row should be considered. As we show in Table 4.1, the faceted value \(\langle x ? \text{"The Paper" : "[redacted]"} \rangle\) is stored as two rows in the papers table, each with different primary keys but with the same jeeves_id of 1. The high-confidentiality facet has a jeeves_vars value of "x=True" and the low-confidentiality facet has a jeeves_vars value of "x=False".

This approach extends naturally to cases where we might have nested facets. For nested facets, we store more labels in the jeeves_vars column. For instance, the faceted value

\[
\langle x ? \langle y ? \text{"The Paper" : "[redacted]" : "[redacted]" : "[redacted]" } 
\]

gets encoded as three database rows where the jeeves_vars strings are "x=True;y=True", "x=True;y=False", and "x=False".

Because we now use facet keys rather than primary keys for referring to faceted records, we need to take this into account when handling foreign keys. We want a foreign key to reference a conceptual row rather than an actual row, so reference it via the facet key jeeves_id rather than the primary key of a row, id.

### 4.2.2 Querying in Jelf

This approach makes it simple to make queries over faceted data. In making queries, Jelf retrieves candidate query results from the database and uses the jeeves_id and jeeves_vars fields to reconstruct the facets. The jeeves_id field allows Jelf to find all components of a faceted value and the jeeves_vars field allows Jelf to correctly reconstruct facets.

The advantage of this simplistic storage scheme is that for the most part, the ORM can issue the same SELECT queries as it normally would, without worrying about facets.

The application-side receives a list of rows from the database server. By looking at the jeeves_vars column, Jelf transforms these rows into a faceted list of model instances (Python objects) using the faceted representation which we describe below.

In Table 4.2, we show two examples of such queries, as well as associated Python code. In the first example, we simply query for authors based on the name. Jelf simply ensures that the query includes jeeves_id and jeeves_vars in the SELECT clause. In the second
### Typical SQL Query

<table>
<thead>
<tr>
<th>Author.objects.filter(name=&quot;Travis&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SELECT</strong> author.id, author.name <strong>FROM</strong> author <strong>WHERE</strong> author.name='Travis';</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Paper.objects.filter(author__name=&quot;Travis&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SELECT</strong> paper.id, paper.title <strong>FROM</strong> paper <strong>JOIN</strong> author <strong>ON</strong> paper.author_id = author.id <strong>WHERE</strong> author.name='Travis';</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Jelf SQL Query</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SELECT</strong> author.id, author.name, author.jeeves_id, author.jeeves_vars <strong>FROM</strong> author <strong>WHERE</strong> author.name='Travis';</td>
</tr>
</tbody>
</table>

| **SELECT** paper.id, paper.title, paper.jeeves_id, paper.jeeves_vars, author.jeeves_vars **FROM** paper **JOIN** author **ON** paper.author_id = author.jeeves_id **WHERE** author.name='Travis'; |

#### Table 4.2: Example of SQL queries made by Jelf.

In the example, the **WHERE** clause filters on the results of a **JOIN**. In the **ON** clause, we use the **jeeves_id** rather than **id**, as discussed above. In the **SELECT** clause, we now have to include the **author.jeeves_vars** as well as the **paper.jeeves_vars**. This prevents the **JOIN** query from leaking information about the author.

### 4.2.3 Connecting with the Application Layer

When the application fetches a row from the database with a label that is not already in memory, Jelf has to attach the Jeeves policy to the label. In order to this, we name the label so that Jelf can find the policy to attach. The name of the label takes the form

{Model name}___{jeeves_id}___{label class}

The model name together with the **jeeves_id** uniquely identifies the object that the label was initially created for. The label class (*e.g.*, "ic" in Figure 2-1) allows Jelf to look up the policy method in the object schema, in case there is more than one per schema. Jelf calls this method, passing it the label’s associated object in order to create the policy.
4.2.4 Extending the Django ORM

We extended Django's ORM to allow the programmer to specify policies on the models, and to allow it to use a special database schema for storing faceted values, as described below. To do this, we wrote JeevesModel, a subclass of Django's Model. JeevesModel adds the extra necessary SQL fields. JeevesModel defines an interface mimicking a subset of the API of Django's ORM, most notably the filter method on a QuerySet, which corresponds to a WHERE clause of a SQL query. This includes filter queries which dereference foreign keys (thereby requiring JOIN clauses). These QuerySets are able to interact with our special database and return faceted Jeeves values.
Chapter 5

Case Study: Conference Management System

We describe the implementation of a conference management system using Jelf and compare it to the implementation of the HotCRP conference management system [17, 3]. We measure performance of our conference management system and look at faceted execution overheads. To test the feasibility of our approach, we have deployed our conference management system in a small workshop.

5.1 Implementation

Our Jelf conference management system has features for submitting papers, and viewing papers and reviews. Users are either authors, PC members, or the PC chair; only the PC chair can designate users as PC members. The PC chair has additional privileges: for instance, assigning reviewers to papers. Paper fields such as author have

<table>
<thead>
<tr>
<th>Language</th>
<th>Files</th>
<th>Code</th>
<th>Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript</td>
<td>3</td>
<td>1826</td>
<td>0</td>
</tr>
<tr>
<td>Python</td>
<td>14</td>
<td>1604</td>
<td>131</td>
</tr>
<tr>
<td>HTML</td>
<td>22</td>
<td>897</td>
<td>0</td>
</tr>
<tr>
<td>CSS</td>
<td>7</td>
<td>212</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>-</td>
<td>4539</td>
<td>131 (2.9%)</td>
</tr>
</tbody>
</table>

Table 5.1: Lines of code and policy in our Jelf conference management system.
policies associated about which users can view the sensitive version of the values. These permissions depend on the current stage of the conference: submission, review, or decision.

Using Jelf, the programmer only needs to declare policies once, alongside the data schema declarations. We counted the lines of policy code, which include the annotations and methods for specifying low-confidentiality facets and policies. We show the breakdown of the lines of code in Table 5.1. This policy code is 131 lines of the 201 lines in models.py and less than 2.9% of the entire code base. With Jelf, the programmer no longer needs to edit multiple files across multiple languages in order to add or change policies. In implementing or maintaining the code for the conference management system’s policies, the programmer only needs to edit this one file and this small amount of code. Even if the program involves more complex models and policies, this concentration of policy code should make system-building and maintenance, as well as security audits, easier.

5.1.1 Comparison to HotCRP Implementation

We compare our implementation to the implementation of HotCRP, a state-of-the-art conference management system used for managing submissions and reviews to academic conferences. HotCRP is an excellent example of the “policy spaghetti” that Jelf’s design is intended to mitigate. Policy code pervades the entire HotCRP codebase. Because the programmer needs to manage policies and access checks, almost all of the PHP files need to be policy-aware.

The anatomy of the HotCRP policy code is as follows. The HotCRP policies are implemented in across at least 24 of the 82 PHP files. Many policy functions are contained in a single file contact.php that is 1779 lines long; this file corresponds to what a Jeeves policy file may look like. These functions can be upwards of 100 lines of nested PHP conditionals. Other files call these functions when determining what may be shown; there are at least hundreds of such checks across the code base. Besides the functions in contact.php, there are additional checks across the program. For instance, there are 191 occurrences of checks involving the privChair and conflictType
variables checking whether the viewer is the PC chair or has the appropriate conflict status. There is also a fair amount of policy code associated with SQL queries: many SQL queries construct WHERE clauses with conditions based on policies.

We show an estimated lower bound on the lines of “policy code” in Table 5.2. We define policy code as code involved in access checks or defining functions used in these checks. The numbers are an underestimate: we counted the functions defined in contact.php, their uses, and the appearance of a few other variables known to be used for access checks. We sampled a few hundred lines of the code base to determine that dynamically-generated SQL queries involving policy checks were interspersed throughout the code. The policy code is at least 2,322 out of 42,006 lines of code in HotCRP. This code includes the 1,779 lines corresponding to contact.php and the over 500 calls to policy functions across 24 files.

Though HotCRP has many more features and more complex policies than our Jelf conference management system, the main take-away is that there is policy code scattered across the code base. It is difficult to even characterize what counts as “policy code” because it is so intertwined with functionality. Designing a privacy API on top of the database could mitigate some of these issues, and having a domain-specific privacy language could reduce the number of lines of policy code. Even with such an approach, however, privacy remains a global concern and the programmer remains responsible for making checks across the code base. In such a system, the trusted base for the policy code must be the entire code base, as opposed to in a Jelf system, where the trusted policy code can be concentrated in one location.
Table 5.3: Running times for conference management system with hard-coded policies.

<table>
<thead>
<tr>
<th>Action</th>
<th>Python</th>
<th>SQL</th>
<th>Template</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Showing available papers</td>
<td>0.03s</td>
<td>0.002s</td>
<td>0.02s</td>
<td>0.05s</td>
</tr>
<tr>
<td>Submitting a paper</td>
<td>0.06s</td>
<td>0.0s</td>
<td>–</td>
<td>0.07s</td>
</tr>
<tr>
<td>Viewing a paper</td>
<td>0.02s</td>
<td>0.0s</td>
<td>0.05s</td>
<td>0.07s</td>
</tr>
<tr>
<td>Submitting a review</td>
<td>0.04s</td>
<td>0.0s</td>
<td>0.01s</td>
<td>0.05s</td>
</tr>
</tbody>
</table>

Table 5.4: Running times for Jelf conference management system with Jeeves policies.

<table>
<thead>
<tr>
<th>Action</th>
<th>Python</th>
<th>SQL time</th>
<th>Z3 time</th>
<th>Template</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Showing available papers</td>
<td>0.12s</td>
<td>0.02s</td>
<td>0.03s</td>
<td>0.23s</td>
<td>0.48s</td>
</tr>
<tr>
<td>Submitting a paper</td>
<td>0.10s</td>
<td>0.006s</td>
<td>–</td>
<td>–</td>
<td>0.19s</td>
</tr>
<tr>
<td>Viewing a paper</td>
<td>0.04s</td>
<td>0.003s</td>
<td>0.01s</td>
<td>0.04s</td>
<td>0.15s</td>
</tr>
<tr>
<td>Submitting a review</td>
<td>0.07s</td>
<td>0.002s</td>
<td>0.01s</td>
<td>0.05s</td>
<td>0.15s</td>
</tr>
</tbody>
</table>

5.2 Running the System

To evaluate the feasibility of our approach, we measured the performance of our system on some representative actions and then tested and deployed it. We compare our Jelf implementation to an implementation with hard-coded policies.

5.2.1 Overview of Representative Actions

To characterize the development time vs. runtime tradeoffs involved in using Jeeves, we took measurements for a few representative actions: displaying the list of submitted papers, submitting a paper, displaying information about a paper (including reviews), and submitting a review.

Showing available papers is the most labor-intensive operation in our system, as it involves going through all the papers and their associated policies. The programmer simply writes a query that fetches all papers and then displays their titles and authors, but in Jelf this will only show the papers one is allowed to see according to the policy. For an author, this includes only papers that she has submitted. For a reviewer, this includes all papers that she is not conflicted with, and for the PC chair, this includes all papers. To produce this view, Jelf resolves the permissions on each paper in the database to determine whether to display it.
Submitting a paper involves taking input from the user and writing to the database. Executing this code involves the framework creating new labels and putting values in facets. From this point onward, a faceted value encapsulates the paper. The faceted value is associated with policies to prevent, for instance, users from seeing each other’s papers. The author field of the paper also becomes faceted to protect this information from reviewers. In our system, this request redirects to another page, so it does not involve displaying any values.

Viewing a paper involves resolving policies on information relevant to a paper. The page for viewing a paper shows the title, author, abstract, a link to the document itself, reviews, and comments. The author is not visible to any of the reviewers. Comments and reviews are visible to the (non-conflicted) reviewers, and reviews may be visible to the author, depending on the stage of the review process.

Submitting a review is similar to submitting a paper, except instead of redirecting to another page, it renders the paper page again.

5.2.2 Running Times

To evaluate the runtime overheads for Jelf policy enforcement, we measured system performance on the Jelf conference management system compared to performance on the same system, but with hard-coded policies. In Table 5.3 we show the running times for a conference management system with hard-coded policies. In Table 5.4 we show the running times for the conference management system running with Jelf policies. We cloned the data from our deployed site and ran some numbers on our Amazon EC2 Micro Instance. Our machine runs Ubuntu 13.10 on 64-bit Intel(R) Xeon(R) CPU E5-2650 0 with a 2.00GHz processor. The production data started with 23 users and seven papers.

We break down execution time into the following relevant components, shows in Tables 5.3 and 5.4. The “Python” column reflects the logic that takes place in fetching, saving, and updating data in the database. For the Jelf version with policies, computation takes place in the Jeeves runtime and works with faceted values. The “SQL” column corresponds to the time spent doing SQL queries. The “Z3” column
corresponds to the time Z3 spends on processing assertions and solving. The “template” column corresponds to the time the system spends projecting out values for creating views, excluding the Z3 time. The “total time” includes other time Django spends, for instance on processing requests.

As we can see, the page which shows all papers is the “stress test” of our system, as it has the largest overheads for the faceted Python execution and for the template evaluation, which includes all concretization and policy resolution. This page has the most labels involved: several labels for each paper. The faceted execution itself has an overhead of about 4x. Most of the total overhead is from the concretization in the template evaluation. This comes from repeated calls to resolve the policies for each data item and is particularly high because it is the first time the application is accessing the data. Caching mechanisms could reduce time for subsequent access.

5.2.3 Deployment Experience

We deployed our conference management system to run a workshop [5] that ultimately had 23 users and 7 submissions. In testing for deployment, we found a bug that led to us restructuring our code to be thread-safe. During deployment, we experienced no major issues, even when multiple issues were on the system at the same time. We had no complaints from users and found that the representative actions and their associated measurements characterized well the load on the system.
Chapter 6

Discussion

In this chapter we discuss issues related to security and choice of language and web framework.

6.1 Security Discussion

The goal of Jeeves is to prevent programmers from inadvertently leaking policies. Jelf applications trust the Python interpreter and database, as well as the programmer to correctly specify policies. In a Jelf application, there are only two places where the code needs to be trusted. The first is the schema and policy definitions themselves. Of course, if the policy is declared wrong, then there may be privacy leaks. The other place is when data is first input by the user, and the application first puts the data into a model and saves it. Until the programmer saves the data, the data is floating around free, and a programmer could leak it at that point. Outside of these areas of code, there is no such thing as a “missing access check” bug for viewing data.

We do not protect against a malicious programmer. A malicious programmer could, for example, access the “hidden” fields of facets to extract the private values. (There is no such things as a private field in Python.) Due to Python’s flexible and dynamic nature, we believe it would be very difficult to remove all similar vulnerabilities. However, Jeeves intends to guard against a careless programmer, and a programmer would have to act very intentionally to subvert the Jeeves runtime. It should be easy
to audit for such attacks.

6.2 Decision to Use Python

We had considered implementing Jelf in Scala, a language for which there already exists an implementation of Jeeves [7], but we chose Python because the dynamic approach of Jeeves fits more with the Python’s dynamic programming idioms. Jeeves’s policy-agnostic approach trades developer overhead for runtime overhead, making it more suitable for rapid prototyping and for building web applications at a smaller scale. While Scala is gaining increasing traction in industry and is in use at companies like Twitter and LinkedIn, it has not been the language of choice for the casual programmer building a recreational web application.

The biggest difference in the approach to embedding a language in Python is that while Scala is statically typed, Python is dynamically typed. Scala’s static types make the programmer think more carefully and be more verbose in the implementation. The trade-off is that the static types provide more guarantees about the cases that the implementation covers, making it more crucial to thoroughly test the code. Both Python and Scala allow for most, but not all, operators to be overloaded. With Scala, a work-around is to use compiler extensions like Scala-Virtualized [21] that allow programmers to overload key constructs. With Python, we were able to work around the lack of full overloading by doing a source transformation. In Python, this was straightforward because the AST is relatively simple and the Python MacroPy library [4] for implementing macros provides support.

6.3 Decision to Use Django

Our web framework is currently an ad hoc layer on top of Django. We decided to use Django because it is a well-known web framework with support for a variety of functionality. It turned out that we needed to customize nearly all of Django’s functionality that we used. Thus, a more lightweight framework may be a more optimal
choice for the base web framework.
Chapter 7

Related Work

Like the Hails web framework [16], Jelf is a web framework designed for security. However, Hails handles access control, while Jelf is concerned with information flow. Also, while Hails provides checks against security violations, it is still up to the programmer to write correct code respecting the policy, and the framework does not help them write that code. By contrast, Jeeves provides the programmer with a way to separate application logic from policy logic, making it easier to implement complex policies as well as enforce them.

Also related is the SIF web framework [13], built on top of the Jif programming language, which extends Java with information-flow control. SIF uses a label-based approach and tracks all information-flow end-to-end, verifying that programs are correct with respect to stated policies. Another language-based approach to web security is Ur/Web [12]. Ur is a dependently-typed functional language which uses static-typing to verify correctness of code to prevent common web vulnerabilities. Ur has the capability to use static analysis for information flow and access control, whereas Jeeves takes the dynamic approach, to track information control at run-time. Again, all of these approaches verify but do not help the programmer write code.

Jeeves follows a line of research in information flow of which Sabelfeld and Myers offer an extensive survey [25]. Jeeves has similar correctness goals as state-of-the-art languages for verifying information flow security: for instance, Jif [23], Fine [11], F* [27], and Fabric [19, 6]. IFDB applies an information flow approach to databases [26]. A
difference is that in these techniques, the goal is checking rather than helping produce
the correct outputs. Jeeves may also be compared to verification approaches for access
control such as Rubicon [24] and Margrave [15]. The goal is, again, verification rather
than mitigating programmer burden. In addition, access control is concerned with
checking at the data interface rather than how data flows.

Jeeves is related to systems that support inserting information flow checks, for
instance with flow locks [9], capabilities et al. [8, 20], and frameworks for executing
checking routines at output channels [29]. These approaches do not modify the runtime
semantics to facilitate tracking flow. There are also parallels with dynamic multi-
execution [10, 14] and symbolic execution [18] approaches. Facets can avoid overhead
when code does not depend on confidential data. In addition, Jeeves allows policies
that may depend on sensitive values.
Chapter 8

Future Work

Jelf has much room for improvement. For example, Python Jeeves currently supports only a specific subset of Python. We do not think it is necessary to support all of python to be useful; however, we do think that it could reasonably support more standard Python functionality than it does currently.

There are also many of optimizations that could speed up performance. On the faceted execution side, there is room to improve performance. As it currently stands, many code paths will be executed multiple times. For example, suppose a certain body of code needs to be executed under the path condition (as is common) $x = \text{high}$ or $y = \text{high}$. Currently, we do not support logical disjunction in the path condition, so the body will be evaluated twice: once with path condition $x = \text{high}$ and once with path condition $x = \text{low}$ and $y = \text{high}$. Supporting arbitrary formulas of labels ought to speed up the faceted execution. There is also a lot of potential for caching the results of labels for certain contexts, which will decrease the time spent concretizing values.

Right now, Jelf only enables the programmer to write policies for viewing a context, but not for writing data. Thus, the programmer still has to manually write access checks whenever a user attempts to update data. Write policies with Jeeves is an area of ongoing research. Furthermore, the current process of putting data from a form into a model is rather ad hoc. Since this is a stage in the pipeline where the data remains unprotected, it would be great to have a cleaner more systematic way to map forms onto models, similar to Django forms, for example.
Finally, more experimentation needs to be done, and we need more examples of Jelf being used to create web applications with complex privacy policies. More case studies can be done, especially with existing Python web applications.
Chapter 9

Conclusions

We describe Jelf, a web framework that mitigates programmer burden in implementing privacy policies. Our main contributions are as follows:

- We present Jelf, a web framework for automatically enforcing information flow policies that provides end-to-end guarantees.

- We describe an implementation of the Jelf application layer as that the programmer can use with a standard Python interpreter.

- We describe an object-relational mapping that we have implemented as a Django extension. Our ORM enforces information flow policies throughout queries and across sessions.

- To demonstrate the feasibility of Jelf, we implement a Jelf conference management system that we have deployed to run a small workshop.
Bibliography


