Preventing Information Leaks with Policy-Agnostic Programming

by

Jean Yang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2015

© Massachusetts Institute of Technology 2015. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
August 28, 2015

Certified by . . . .
Armando Solar-Lezama
Associate Professor without Tenure
Department of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering
Chair, Department Committee on Graduate Students

Preventing Information Leaks with Policy-Agnostic Programming

by

Jean Yang

Submitted to the Department of Electrical Engineering and Computer Science
on August 28, 2015, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

As a solution to the problem of information leaks, I propose a *policy-agnostic* programming paradigm that enforces security and privacy policies by construction. I present the implementation of this paradigm in a new language, Jeeves, that automatically enforces *information flow* policies describing how sensitive values may flow through computations. In Jeeves, the programmer specifies expressive information flow policies separately from other functionality and relies on the language runtime to customize program behavior based on the policies. Jeeves allows programmers to implement information flow policies once instead of as repeated checks and filters across the program. To provide strong guarantees about Jeeves programs, I present a formalization of the dynamic semantics of Jeeves, define non-interference and policy compliance properties, and provide proofs that Jeeves enforces these properties.

To demonstrate the practical feasibility of policy-agnostic programming, I present Jacqueline, a web framework built on Jeeves that enforces policies in database-backed web applications. I provide a formalization of Jacqueline as an extension of Jeeves to include relational operators and proofs that this preserves the policy compliance guarantees. Jacqueline enforces information flow policies end-to-end and runs using an unmodified Python interpreter and SQL database. I show, through several case studies, that Jacqueline reduces the amount of policy code required while incurring limited overheads.

Thesis Supervisor: Armando Solar-Lezama
Title: Associate Professor without Tenure

Acknowledgments

When my advisor and I first met, I was a first-year Ph.D. student recovering from my first conference paper submission and he had arrived on campus fresh after his Ph.D. Over the years, there was probably a nontrivial amount of mutual skepticism and fear of the uncertain future over 3 a.m. post-deadline cakes and unfavorable paper reviews at various times of day. No matter what, he supported me in what I wanted to do and believed in the work. I am fortunate to have found such an open-minded, sharp, and caring advisor in Armando Solar-Lezama. I am impressed with how much he made himself available to workshop my ideas, research and otherwise. Not every professor has the insight to recognize that "everyone likes Ryan Gosling."

Hearing others' complaints about thesis committees being too involved or too apathetic made me appreciate my own thesis committee even more. Discussions with my committee members Martin Rinard and Nickolai Zeldovich have improved not only the presentation of my thesis work, but also my views on research. Martin reminds me that research should be shocking and Nickolai reminds me that research should be useful.

My collaborations with other students have made me appreciate what a special place MIT is in terms of both scientific curiosity and technical competence. One of the turning points in the development of my Jeeves programming language was when my groupmate Kuat Yessenov convinced me to switch from implementing it as a standalone interpreter in OCaml to embedding it in Scala by implementing an initial Scala embedding in one week. Another turning point in my thesis work was when I began collaborating with my Masters of Engineering student Travis Hance. I had agreed supervise Travis Hance's thesis work both because of his impressive résumé and because he said the research I described sounded "fun." It turned out that technical competence and fun were the two traits I came to appreciate most about him. Travis finished his proposed task of switching our implementation strategy to Python in less than a semester, giving us a full second semester to collaborate closely on both the initial design and implementation of the Jacqueline web framework

and the Haskell Ryan Gosling meme. Finally, the willingness of my undergraduate and high school research assistants—Lena Abdalla, Amadu Durham, Jesse Klimov, Pat Long, Ariel Jacobs, and Benjamin Shaibu—to build applications in an evolving, often poorly-documented research language case was immensely helpful towards evaluating the design and implementation of Jeeves.

My outside collaborations have improved both my thesis work and my understanding about how to present and position the work. At the POPL conference in 2012, I presented the initial semantics for policy-agnostic programming with Jeeves and Tom Austin, then a Ph.D. student at UC Santa Cruz, presented a semantics for faceted execution, a complementary execution approach. I had the good fortune of being scheduled to give my talk in an earlier session. During Tom's talk, Philip Wadler, a senior member of our field, stood up and asked, "But how is this different from Jean's work?" This led to a discussion that led to a several-year collaboration with Tom and his Ph.D. advisor, Cormac Flanagan. Out of this collaboration came the faceted semantics for policy-agnostic programming. Steve Chong joined us in building on this to develop a formalization of the Jacqueline web framework.

During my Ph.D., my two internships at Microsoft Research gave me new perspectives on both research directions and ways to conduct research. My mentors Nikhil Swamy and Juan Chen introduced me to dependent type systems and language-based security. Chris Hawblitzel changed the way I think about system design and provable correctness.

One of the aspects of MIT I loved most was the community it fosters. I have enjoyed interacting with the other members of the Programming Languages and Software Engineering Group, especially as the group has grown and solidified over the course of my Ph.D. Saman Amarasinghe, Rob Miller, and Daniel Jackson have given excellent advice as I explored how I want to be as a researcher. Graduate school would not have been nearly as enjoyable or intellectually growthful without the company of my extended cohort: Eunsuk Kang, Aleksandar Milicevic, Sasa Misailovic, Joe Near, and my groupmate Rishabh Singh. The other members of the Computer-Aided Programming group—Alvin Cheung, Sicun Gao, Shachar

Itzhaky, Jimmy Koppel, Nadia Polikarpova, Evan Pu, Xiaokang Qiu, Rohit Singh, Zenna Tavares, Zhilei Xu, and Kuat Yessenov—have also provided excellent feedback, conversation, and commiseration.

My experience working on backend privacy at Facebook shaped my perspective on the relative importance of different problems. Dwayne Reeves identified the opportunity to apply formal methods to checking privacy policies and patiently fielded all of my questions about Facebook's internal infrastructure for three months as my deskmate and internship mentor. Venkat Venkataramani provided the managerial support to pursue this problem. Yiding Jia's code reviews made me a better Haskell programmer. Pieter Hooimeijer took on-demand walks with me whenever I wanted to talk through technical problems. Harry Li helped me navigate my internship and the logistics of my follow-up work.

My undergraduate professors Greg Morrisett and Norman Ramsey first piqued my interest in the applications of logic to program correctness. I would have probably left Computer Science for a less male-dominated field had it not been for the encouragement of my other professors, Margo Seltzer and Radhika Nagpal.

My work has been supported by the National Science Foundation Graduate Research Fellowship, the Facebook Fellowship, and the Levine Fellowship.

My work would not be nearly as meaningful without my family and friends. I would like to thank my parents and sister for their unwavering support and for not always encouraging me. In fact, that they discouraged me from so many things is how I ended up becoming a researcher in Computer Science. I would also like to thank my friends for the pep talks, for the meals cooked, and for the adventures.

Finally, I would like to thank Eunsuk Kang for agreeing to file my thesis while I attended the New Faculty Retreat at Carnegie Mellon University and Chinmay Kulkarni for his collaboration on this Acknowledgments section at the retreat. Peter Bailis, Michael Keller, Sara Watson, and Cliff Chang also helped improve the readability of this document.

Contents

1 Introduction **13**

1.1 The Policy-Agnostic Approach . . . 14

1.2 Jeeves, a Language for Automatically Enforcing Information Flow Policies . . . 15

1.3 Jacqueline, a Policy-Agnostic Web Framework . . . 16

1.4 Advantages of the Policy-Agnostic Approach . . . 18

1.5 Contributions of the Thesis . . . 20

1.5.1 Key Technical Challenges . . . 21

1.5.2 Thesis Overview . . . 22

2 Background and Related Work **23**

2.1 The Problem of Constructing Secure Systems . . . 23

2.1.1 When Encryption and Access Control Are Not Enough . . . 23

2.1.2 The Leaky Enforcement Problem . . . 25

2.1.3 Programmer Burden with Information Flow Checking . . . 26

2.1.4 Limitations of Multi-Execution . . . 28

2.1.5 Limitations of Existing Web Frameworks . . . 29

2.1.6 How Policy-Agnostic Programming Fills the Gaps . . . 30

2.2 Comparing to Other Language-Based Techniques . . . 31

2.2.1 Restrictions of Aspect-Oriented Programming . . . 31

2.2.2 Limitations of Prior Work in Executing Specifications . . . 32

2.2.3 Relationship to Variational Data Structures . . . 33

2.2.4 Relationship to Acceptability-Oriented Programming . . . 33

2.2.5 Relationship to Declarative Domain-Specific Languages . . . 34

I Policy-Agnostic Programming for Information Flow 35

3 Policy-Agnostic Programming in the Jacqueline Web Framework 36

3.1 Schemas and Policies in Jacqueline 37

3.1.1 Secret Values and Public Values 38

3.1.2 Specifying Policies 39

3.2 Policy-Agnostic Controller Code 40

3.3 Computing Concrete Views 42

4 Jeeves, a Language for Automatically Enforcing Information Flow Policies 44

4.1 Sensitive Values and Policies 44

4.2 Policy-Agnostic Programs 47

4.3 Producing Concrete Values 47

4.4 Handling Dependencies Between Sensitive Values and Policies 48

4.5 Policy Language Limitations 50

II Reasoning about Policy-Agnostic Programs 52

5 Semantics and Guarantees for Faceted Execution of Jeeves 53

5.1 Core Semantics 53

5.2 Properties 62

5.2.1 Projection Theorem 62

5.2.2 Termination-Insensitive Non-Interference 64

5.2.3 Termination-Insensitive Policy Compliance 65

6 Faceted Execution for Database-Backed Applications 68

6.1 Solution Overview 69

6.1.1 Executing Relational Queries with Facets 71

6.1.2 Early Pruning Optimization 74

6.2 Formal Semantics and Policy Compliance 75
6.2.1 Syntax and Formal Semantics 75
6.2.2 Formal Semantics 77
6.2.3 End-to-End Policy Compliance 82
6.2.4 Early Pruning 84

III Executing Policy-Agnostic Programs 85

7 Implementing a Policy-Agnostic Web Framework 86
7.1 Python Embedding of the Jeeves Runtime 86
7.1.1 Faceted Execution 87
7.1.2 Evaluating Policies at Computation Sinks 87
7.1.3 Garbage-Collecting Labels and Policies 88
7.1.4 Representing lists 88
7.1.5 Jacqueline ORM 88
7.1.6 Decision to Use Python 89
7.2 Limitations 90

8 Jacqueline in Practice 91
8.1 Applications 92
8.2 Code Comparisons 92
8.2.1 Django Conference Management System 93
8.2.2 HotCRP 94
8.3 Performance 94
8.3.1 Representative Actions 95
8.3.2 Stress Tests 96
8.3.3 Early Pruning Optimization 97

9 Conclusions 98
9.1 Summary of Contributions 99
9.2 Conclusions 101

A Proofs for Faceted Execution of λ^{jeeves} **116**

A.1 Proof of Projection 116

B Proofs for λ^{JDB} **121**

B.1 Rules from λ^{jeeves} 121

B.2 Proof of Lemma 5 122

B.3 Proof of Lemma 6 123

B.4 Lemma 7 124

B.5 Proof of Theorem 4 (Projection) 125

List of Figures

1-1 Application architecture in Django vs. Jacqueline. 19

3-1 Jacqueline schema fragment for calendar events. Policy code is shown with a gray background. 37

3-2 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. 40

4-1 Jeeves syntax. 45

4-2 "Constructor" code for an event record in Jeeves. 48

5-1 The source language λ^{jeeves} 54

5-2 Runtime syntax for λ^{jeeves}evalution. 55

5-3 λ^{jeeves} expression evaluation. 56

5-4 Auxiliary functions for λ^{jeeves}evaluation. 57

5-5 Faceted evaluation semantics for application and statements. 60

5-6 Semantics for derived encodings. 61

6-1 λ^{JDB} syntax. 76

6-2 Runtime syntax and contexts rules for faceted evaluation of λ^{JDB}. . . 77

6-3 Rules for evaluation λ_J subset of λ^{JDB}. 78

6-4 Rules for evaluation with relational operations. 79

7-1 Representing faceted lists: (a) is the naive representation and (b) is the optimized representation. 88

8-1 Distribution of policy code with Jacqueline and Django conference management systems. 93
8-2 Times to view profiles for a single paper and single user, in Jacqueline and Django. 95
8-3 Times to view list of summary information for all papers and all users, in Jacqueline and Django. 96
8-4 Jacqueline stress tests for other case studies. 97
8-5 The course manager stress test performs well with the Early Pruning optimization and times out otherwise. 97

List of Tables

6.1 SQL code and example tables, with and without policies. 70
6.2 Translated ORM queries in Django vs. Jacqueline. 70

Chapter 1

Introduction

From social networks to electronic health record systems, a growing number of programs compute using sensitive information. Access permissions are not only becoming more complex [39,77], but also application code bases are growing to have millions of lines of code [50]. When programming these software systems, there are many opportunities to inadvertently leak information.

Leaks often occur not because information is unprotected, but because it is protected incorrectly. For instance, applications can fail to prevent *indirect leaks* through values computed using sensitive values. In the HotCRP conference management system [53], paper authors are not supposed to initially see scores for their submitted papers, but a bug [55] allowed authors to deduce their scores through the search interface. Another source of leaks is the *incorrect viewer* problem, where the program resolves permissions for a viewer different from the actual viewer. Another bug in the HotCRP conference management system [53] allowed any user to preview a password reminder email for another user [108]. Here, the programmer failed to realize the viewer of the preview was different from the recipient of the email.

Prior work in *information flow* prevents such leaks, but prevention does not address the issue of programmer burden from *policy spaghetti*, the intertwined code for policies and other functionality. Information flow techniques track the flow of sensitive information through computations. These tools simply accept or reject flows, leaving the programmer responsible for constructing programs that do not leak information.

Static, language-based approaches [21, 34, 35, 49, 74, 76, 83, 99, 100, 102] prevent programs from running at all, while dynamic approaches [16,20,22,36,44,94,108] cause unpredictable runtime behavior. In both cases, the programmer remains responsible for implementing information flow concerns as repeated checks and filters and handling the different possible execution paths for different viewers. Adding new functionality or implementing new policies requires reasoning about intertwined code for policies and other functionality across the application.

An additional source of programmer burden is in preventing *leaky policy enforcement*, where the results of permission checks leak information about sensitive values. Another HotCRP bug displayed a "Congratulations!" message when the final-copy submission deadline was set, but only in the presence of the "Accepted" status [54], allowing authors to infer acceptances before they had become official. This leak occurred because the policy to display additional information did not take into account that the "Accepted" status was a sensitive value with its own policy. With label-based information flow checking approaches [8, 44, 63, 74], for instance, the programmer is trusted to manage dependencies between permissions that are encoded in terms of Boolean checks. Even if the information flow checking approach can detect these leaks, the programmer needs to construct the program correctly with respect to these dependencies.

1.1 The Policy-Agnostic Approach

The research I present in this thesis addresses these shortcomings by automating the enforcement of information flow policies. I present a new paradigm, *policy-agnostic programming*, that reduces information leaks by preventing opportunities for error. The goal is to allow programmers to write security- and privacy-conscious code that looks as similar as possible to code written without policies in mind. This paradigm allows the programmer to associate data values with policies that describe how values may flow through a program. The programmer may implement the rest of the program in a way that is *agnostic* to the policies. A language runtime is

responsible for customizing program behavior to adhere to the policies. The runtime also manages policy dependencies, including mutual dependencies between policy computations and other computations.

The two main challenges in making policy-agnostic programming feasible are 1) factoring out a specification of correct information flow and 2) customizing program behavior based on the policies in a scalable way. First, we want a specification of information flow that does more than simply rejecting disallowed flows. If the viewer may not see a sensitive location value, for instance, it may be appropriate to compute an output using the corresponding city or country instead. The challenge is in determining how to specify this behavior in a way that is not deeply intertwined with the computation. Second, ensuring that the program behaves according to the specification may require nontrivial modifications to program execution. I choose a dynamic approach because for many of these policies, the relevant information, for instance the viewing context, is only available at runtime. A challenge here is in ensuring that the approach does not incur excessive overheads.

1.2 Jeeves, a Language for Automatically Enforcing Information Flow Policies

In this thesis, I factor out information flow policies in the design of Jeeves [10, 106], a language for automatically enforcing information flow policies for security and privacy. Jeeves programs specify information flow policies once, associated with sensitive data. Suppose Alice and Bob want to plan a surprise party for Carol at 7pm next Tuesday at Schloss Dagstuhl. Jeeves allows us to specify that the secret details of the party are sensitive and that only guests should be able to view this information. Jeeves also allows us to specify that everyone else sees an alternate default value. For instance, perhaps only the guests see "surprise party" for the event description, while everyone else sees "group discussion." The Jeeves runtime enforces policies to ensure that the appropriate value is used in all computations. If Carol searches for

"parties near Wadern at 7pm next Tuesday," her surprise party should not appear in the search results. Jeeves policies may depend on sensitive values: for instance, the policy on the guest list can require membership in the list itself. Underlying Jeeves is a well-defined formal semantics and a provable guarantee that runtime policy enforcement does not leak sensitive information.

I designed Jeeves to be *language-agnostic*, sufficiently general to work either on its own or embedded into a host language. The semantics I present allows Jeeves to be implemented either as a standalone interpreter or embedded into a subset of an existing compiled or interpreted language. I present implementations of Jeeves as embeddings in Scala [106], a statically typed language that is typically compiled [78], and Python, a dynamically typed language that is typically interpreted [82]. I intend for Jeeves to be a *kernel language*, an "assembly language" containing the key constructs for policy-agnostic programming. These constructs allow us to create of higher-level abstractions that can take advantage of the guarantees Jeeves provides while implementing domain-specific optimizations.

1.3 Jacqueline, a Policy-Agnostic Web Framework

To demonstrate the feasibility of the policy-agnostic approach for building web applications, I present the Jacqueline web framework [105]. While Jeeves provides a useful programming model and strong formal guarantees, it is unsuited for building realistic web applications for the following reasons:

- *Guarantees do not extend to interoperation 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 Jeeves programs interact with an external database.

- *Expensive execution model.* The Jeeves runtime 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) that uses Jeeves constructs to manage policies. 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 is 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. An ORM can do this through judicious manipulation of meta-data. Jacqueline improves upon Jeeves in the following ways:

- *End-to-end guarantees for database-backed applications.* Jacqueline's semantics extend Jeeves with relational operators in order to track sensitive values and policies through database queries. This yields a provable 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. Therefore, as soon as the runtime knows the viewing context, it can prune alternate execution branches. This optimization preserves end-to-end policy compliance, allows Jacqueline to have reasonable overheads in practice, and is necessary for non-trivial computations involving sensitive values.

Another major advantage of using Jacqueline over Jeeves is that Jacqueline substantially raises the level of abstraction. Not only does Jeeves lack support for building web applications, but the programmer must also manage the association of policies with sensitive values and the designation of the viewer. Jacqueline abstracts over these details, taking responsibility for attaching policies to values upon database

writes and keeping track of the user associated with the current session. Jacqueline provides a general-purpose solution for policy-agnostic programming in web frameworks. While I present an implementation of Jacqueline that extends the popular Python web framework Django [2], one could use a different language and implementation strategy.

In this thesis, I demonstrate the advantages of using Jacqueline for building realistic web applications. I present results from implementing several case studies, including a conference management system used to run an academic workshop. These case studies suggest that Jacqueline is sufficiently expressive for the policy needs of web applications. I compare an implementation using Jacqueline to an implementation with manually implemented policies using Django to show that Jacqueline not only concentrates the policy code in a single location, but also decreases the total amount of policy code that the programmer needs to write. I show through stress tests and representative benchmarks that not only do applications written using Jacqueline exhibit reasonable performance, but they also often have negligible overheads compared to applications written using unmodified Django.

1.4 Advantages of the Policy-Agnostic Approach

Jacqueline demonstrates several advantages that make the policy-agnostic programming model especially appealing for large, security-conscious software systems. The first is that the language runtime manages the policies, thus removing the need to trust the remaining application code of the web server. The second is that rather than simply preventing forbidden outputs, Jacqueline adapts program behavior to adhere to policies. An additional benefit is that factoring out the policy specification decreases the amount of policy code needed. For these reasons, the programmer can update policy code independently of other code, reducing policy spaghetti and the likelihood that new code can introduce an information leak.

By contrast, in most other security-conscious web frameworks, the programmer must implement policies by writing checks and filters throughout the application

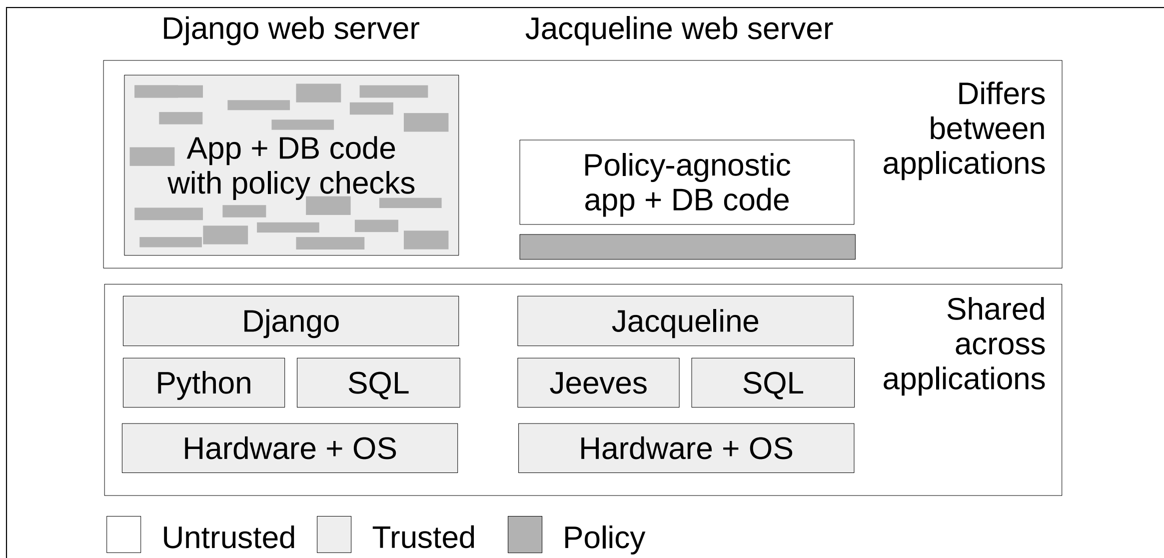


Figure 1-1: Application architecture in Django vs. Jacqueline.

and database code. Even using dynamic information flow systems that automatically prevent information leaks at runtime [44, 108], the programmer must construct the program to not leak information in order to avoid uncaught exceptions and silent failures. In Figure 1-1 we compare the architecture of a Jacqueline program to that of a program using the popular Python web framework Django [2]. 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. This reduces both programmer burden and the opportunity for error.

The work I describe focuses on preventing server-side information leaks that result from programmer error. Our attack model treats the user as the attacker and assumes the programmer is not adversarial. The policy-agnostic approach eliminates the class of errors resulting from missing and incorrectly-implemented access checks. Systems-level attacks such as SQL injection, directory traversal, cross-site reference forging, and cross-site scripting are outside the scope of this approach. Note that this does not mean policy-agnostic runtimes and web frameworks cannot protect against such attacks. The Jacqueline web framework inherits Django's standard integrity mechanisms for these specific attacks.

1.5 Contributions of the Thesis

In this thesis, I demonstrate the feasibility of the policy-agnostic approach for enforcing privacy and security in web applications. The main contributions are as follows:

- **The policy-agnostic programming paradigm.** To reduce the opportunity for information leaks, I present a programming model that factors out information flow policies. The policy-agnostic approach allows the programmer to specify information flow policies separately from other functionality and rely on the runtime to customize program behavior based on the policies. The automated enforcement prevents indirect leaks, reduces the opportunity to compute the incorrect viewer, reduces policy spaghetti, and prevents leaky enforcement. I implement this model in the Jeeves language.

- **A solution to the *leaky enforcement* problem.** Prior work on information flow allows the programmer to leak information through policy enforcement. Tools can only prevent leaks, leaving the programmer responsible for correctly managing dependencies between sensitive values and permission checks. In the policy-agnostic paradigm, the language runtime manages policy dependencies, thus preventing leaky enforcement.

- **A faceted execution semantics for policy-agnostic programming.** If we are to rely on a language runtime to enforce policies, it is important to reason about the guarantees the runtime provides. I present a dynamic semantics for Jeeves that extends the standard imperative lambda calculus with constructs for automatically enforcing information flow. I provide proofs for guarantees of termination-insensitive non-interference and policy compliance. The semantics describe a programming model that can be implemented as a standalone interpreter or embedded into an existing host language.

- **A policy-agnostic semantics for database-backed applications.** Modern web applications rely heavily on commodity databases for performance. To support

interoperation with relational databases, I present an extension of the Jeeves semantics with relational operators that maps naturally onto the implementation of an object-relational mapping for relational databases. I provide proofs that this extends Jeeves's policy compliance guarantees.

- **A demonstration of feasibility for database-backed applications.** A major question about the policy-agnostic approach is whether it can scale for realistic applications. I present the Jacqueline web framework and demonstrate that not only is the programming model useful for reducing the amount of policy code, but that it has reasonable overheads compared to code written with manually implemented policies. I present application case studies and evidence that policy-agnostic programming 1) reduces the amount of trusted policy code that the programmer needs to write and 2) has reasonable overheads compared to code with manually-implemented policies.

1.5.1 Key Technical Challenges

Realizing practical policy-agnostic programming required addressing several technical challenges, including:

- **Designing a scalable language for enforcing information flow policies.** Procedural programming paradigms do not provide support for factoring out information flow, as information flow policies can affect all points in the program. Constraint functional programming models are a better fit, but existing constraint functional programming languages [45, 46, 64, 73] involve substantial runtime search and thus scale poorly. Policy-agnostic programming is sufficiently expressive to capture information flow policies for a variety of applications yet can be executed with reasonable overheads.

- **Formulating policy compliance guarantees for Jeeves.** Jeeves differs from previous approaches to information flow in two key ways. First, rather than checking that a program adheres to a specification of permitted information

flows, the Jeeves runtime adapts program behavior to produce outputs that adhere to specifications. Second, Jeeves allows policies to depend on sensitive values. Existing semantics for information flow do not model this dependence. I provide theorems of termination-insensitive non-interference and policy compliance that describe these guarantees.

- **Extending Jeeves with relational operators.** Prior work on language-based approaches provides guarantees only within a single runtime. To extend Jeeves's guarantees to work with unmodified relational databases, I present an extension of the Jeeves semantics with relational operators and proofs that this extends the guarantees. This allows us to use Jeeves for database-backed applications with an unmodified relational database.

- **Optimizing Jacqueline for web applications.** The Jeeves semantics are potentially expensive to execute because they consider all possible program paths based on the different ways sensitive values may be displayed. The work I present uses observations about typical web frameworks to formulate and implement an *early pruning* optimization that makes it so that in the common case, the overheads are reasonable and often negligible.

1.5.2 Thesis Overview

In Part I, I present the programming model for the Jacqueline web framework (Chapter 3) and explain the key constructs of the underlying language Jeeves (Chapter 4). I present the semantics and guarantees of Jeeves and Jacqueline in Part II. In Chapter 5 I present the Jeeves semantics and guarantees. In Chapter 6 I present the semantics for Jeeves extended with relational operators, the translation to relational databases, and the extension of the guarantees. Part III is about practical considerations. In Chapter 7 I describe the implementation of Jacqueline as a Python web framework and in Chapter 8 I demonstrate the practical feasibility of Jacqueline through evaluation on several application case studies.

Chapter 2

Background and Related Work

In this chapter, we examine prior solutions to helping programmers construct secure systems, the gaps they leave, and how policy-agnostic programming addresses the issues. We also discuss why existing programming solutions are insufficient for automating the enforcement of information flow policies and the key ways in which policy-agnostic programming differs.

2.1 The Problem of Constructing Secure Systems

The major prior approaches to constructing secure systems include *encryption*, *access control*, and *information flow checking*. Using these approaches, problems that may still arise include indirect flows, computing the incorrect viewer, leaky policy enforcement, and policy spaghetti. We describe the prior work and how policy-agnostic programming fills the gaps.

2.1.1 When Encryption and Access Control Are Not Enough

Work in encryption and access control focuses on techniques for protecting individual sensitive values. For the calendar example about planning Carol's surprise party, we can protect event details by encrypting them, ensuring that in case of a data breach, only viewers with the appropriate *secret keys* can successfully decrypt

the sensitive data to access the contents. Encryption techniques can limit the amount of information secret key holders may learn. *Functional encryption* [19, 90] allows holders of the secret key to learn only functions over encrypted values and *secure multi-party execution* [48, 107] allows multiple parties to jointly compute functions over inputs while keeping the inputs private. With encryption approaches, however, *key management* remains an issue. It is outside the scope of encryption to determine what values are private to whom and which viewers receive the secret key.

Access control approaches partially address the issue of key management. Prior work defines specification languages and implementation strategies for checking complex access permissions. Using the Margrave policy analysis tool [37,39,101], for instance, the programmer can specify permissions and issue queries about checks that the engine uses a model checker to resolve. Rubicon [77] allows the programmer to express access checks embedded in Ruby programs and Sunny [68] supports access control checking in model-based, event-driven programs. Related work in logics for access and authorization characterizes trade-offs between expressiveness and analyzability [12–15,43,60,61,75,80]. Prior work on access control has two main issues. First, once access is granted, the programmer is responsible for ensuring data flows to the viewer for which permission was granted. Second, the separation of permission checks from other computations leaves the programmer responsible for managing dependencies of checks on sensitive values.

With encryption and access control, the program can still leak information through indirect flows and incorrect viewer computation. Once the program decrypts a value or resolves an access check, the programmer is responsible for ensuring that derived values are not revealed under disallowed circumstances. If computations, for instance search queries, involve sensitive values, the programmer needs to prevent indirect flows by ensuring that the results of the computations flow only to viewers who have access to the sensitive values involved in the computations. Giving the programmers responsibility for ensuring values flow to the appropriate viewers can also lead to the incorrect viewer problem, which arises when the actual viewer of a sensitive value or derived value differs from the intended viewer. Recall

the HotCRP bug that allows users to preview password reminder emails for any other user [108]. Especially when the viewer is different from the user driving the computation, the programmer risks resolving permissions for the incorrect viewer.

2.1.2 The Leaky Enforcement Problem

Leaking policy enforcement is a risk when the programmer is responsible for managing the interaction between permission checks. Leaky enforcement occurs when resolving permissions for one sensitive value leaks information from other sensitive values. In Chapter 1, we introduced this problem in relation to the HotCRP bug, where the presence of another value leaked the sensitive "Accepted" status. This problem arises in encryption and in access control approaches that maintain a stratification between the data being protected and the data used for checking permissions. With encryption, data must be decrypted in order to be used in checks. For instance, to check the policy that the viewer needs to be a member of the surprise party guest list in order to see it, the program needs to first decrypt the guest list. Whenever permissions do not capture dependencies on sensitive values, the programmer is trusted to prevent leaky enforcement. Much of the prior work on access control uses policies that depend on sensitive data assume "omniscient access." That is, policies can access all data without restrictions [80].

Work on preventing leaky enforcement in database access control addresses the issue that it is increasingly common for access policies to support arbitrary queries. Hippocratic databases [7] differentiate between users that own a database table and users that own data within the table. Follow-up work [59] examines how policies for such databases may depend on data contained within a table. Rizvi *et al.* [87] describe query rewriting rules for determining whether a query is authorized, allowing only queries based on authorized views. Olson *et al.*'s *Reflective Database Access Control (RDBAC)* [80] propose *reflective access policies* that allow access policy decisions to depend on data contained in other parts of the database. Olson *et al.* motivate the problem by citing how popular databases such as Oracle's Virtual Private Database

allow for arbitrary code and providing examples for how policy interactions can lead to security vulnerabilities. Prior work on leaky enforcement focuses on access control policies in databases. The work in this thesis prevents leaky enforcement for information flow across the imperative application code as well as across database queries.

2.1.3 Programmer Burden with Information Flow Checking

Information flow approaches prevent indirect flows and reduce the opportunity to resolve permissions for the incorrect viewer. Prior work tracks how sensitive values flow through computations in order to detect leaks. Static, language-based approaches [34, 35, 47, 49, 74, 76, 83, 99, 100, 102] analyze the program and check that any program that is permitted to run does not have leaks. Prior work on static approaches [89, 97] formulates guarantees such as *non-interference*: the property that secret values do not affect public outputs. Dynamic approaches [16, 20–22, 44, 56, 94, 108] allow the programmer to define checking functions to be used at system boundaries. A runtime tracks the flow of sensitive values, either halting the program with an exception or silently failing if policies are violated. Prior work on dynamic approaches requires minimal changes to existing programs, at the expense of using a modified, slower runtime. The Fabric system [8, 63] and Le Guernic *et al.*'s automata-based monitoring [58] combine static and dynamic techniques. IFDB [93] uses a decentralized label-based model for checking information flow in databases.

The main issue with information flow approaches is that of policy spaghetti: the programmer remains responsible for correctly constructing a program that exhibits the desired behavior. To support the search "parties near Wadern at 7pm next Tuesday," the programmer must ensure that only viewers who know the event is a surprise party for Carol can see the event in the search results. If we want Carol to see that she has an event somewhere in Wadern while guests see the actual location, the program needs to additionally manage the different views. Implementing and updating policies and other functionality requires reasoning about the interaction

of policies and other functionality across the application.

Two additional issues with static approaches are *restricted policies* and the *annotation burden* of specifying permitted flows across the program. For instance, Jif [74] supports variable annotations describing principals that own them and the principals that are allowed to see them. The following code declares variables x and y that are owned by Alice, where Bob can see x and both Bob and Chuck can see y:

```
int {Alice→Bob} x;
int {Alice→Bob, Chuck} y;
```

Jif ensures that only permitted flows may occur in the program:

```
x = y; // OK: policy on x is stronger
y = x; // BAD: policy on y is not as strong as x
```

This approach guarantees that forbidden flows will not occur, but at the cost of limited expressiveness and programmer burden. These policies only allow the programmer to specify whether a principal can see a value. They do not allow, for instance, *conditional policies* that allow a principal to see a value if some predicate P is true. In addition, the programmer must correctly construct the program and specify the annotations. Dependently typed approaches [99] improve expressiveness, but at the cost of undecidable static checking that restricts both policy and program complexity. Follow-up work has extended Jif with both more expressive dynamic labels [111] to improve expressiveness and label polymorphism and label inference [74] to mitigate programmer burden. Even so, annotation burden has motivated researchers to look to approaches that are primarily dynamic [23].

Dynamic approaches decrease the annotation burden, but at the cost of unhandled exceptions and silent failures. Dynamic information flow systems such as Flume [56], Resin [108], and Hails [44] track the flow of sensitive values to prevent information leaks at different levels of granularity. These approaches allow the programmer to define permission checking code to be executed at output channels. These policies are expressive, as they may contain program code, but at the cost of unpredictable runtime behavior. The runtime system tracks sensitive values and either raises an exception or fails silently if sensitive values or their derived val-

ues flow to disallowed channels. While these approaches prevent leaks, they do not provide support for constructing programs that do not leak information. The programmer must still reason about policy spaghetti across the code. Russo and Sabelfeld [88] discuss more trade-offs between static and dynamic information flow analyses for security.

Leaky enforcement remains an issue with information flow checking. Because static analyses must be conservative, typical static information flow approaches support a static mapping between principals and data they can see. This forces a stratification between sensitive values and the permission checking code. As a result, the programmer must enforce such policies outside of the checking tool, leaving the program vulnerable to leaky enforcement. Dynamic information flow approaches *do* allow dependencies of policies on sensitive values. Dynamic techniques prevent leaky enforcement, but the prevention manifests as unhandled exceptions or silent failures. To avoid these behaviors, the programmer must correctly manage policy dependencies.

2.1.4 Limitations of Multi-Execution

Multi-execution is another dynamic information flow checking approach that prevents leaks by executing using multiple values. Capizzi *et al.* [24] describe *shadow execution*, an approach that guarantees non-interference by executing multiple copies of the program. Devriese and Piessens's *secure multi-execution* strategy [36] applies this approach to JavaScript code. Kashyap *et al.*'s scheduling-based dynamic approach [51] partitions a program into sub-programs for each security level for ensuring timing and termination non-interference. The *faceted execution* semantics [9] simulate simultaneous multiple executions within a single runtime.

Policy spaghetti remains an issue with multi-execution. Multi-execution treats programs in a black-box manner, deciding whether a program written as-is may leak a sensitive value. The approach does not modify the program itself and so is only able to transform a program that leaks information into one that outputs arbitrary values.

If the programmer has constructed a program to be non-interferent, then secure multi-execution will show the result of executing on the sensitive values. Otherwise, the execution will output the result of computing on an arbitrary alternate value. Like other checking approaches, the goal of multi-execution is to prevent a missing check from turning into an information leak, but the programmer is expected to write a non-interferent program.

With multi-execution, the programmer remains responsible for preventing leaky policy enforcement. As with the static approaches, prior work [9, 24, 36, 51] requires the programmer to encode policies in terms of whether a given principal is high-privilege. To allow a policy such as "only party guests may see the sensitive guest list," the programmer must make the users sensitive in order to make the list high-privilege. In this case, the approach does not prevent leaks. In the other case, the list cannot be sensitive and the approach also cannot prevent against leaks.

2.1.5 Limitations of Existing Web Frameworks

Prior work on information flow in web frameworks focuses on checking and thus also has the problems of policy spaghetti and leaky enforcement. Passe [17] dynamically analyzes applications to enforce policies about what information may be leaked from database queries. Hails [44] uses dynamic information controls to prevent leaks across the application and database. SIF [31], SELinks [32], SeLINQ [92], and the work of Lourenço and Caires [67] use label-based approaches for verifying information flow in database-backed applications. Ur/Web [28] uses static dependent types to check information flow properties in web programs. Tracking sensitive values in the database is also related to prior work in data provenance [6, 81], especially recent work in provenance for security [1, 26] that uses the history of how values were computed for enforcing security properties.

2.1.6 How Policy-Agnostic Programming Fills the Gaps

The policy-agnostic approach factors out the specification of information flow policies to eliminate policy spaghetti and prevent leaky enforcement. Policy-agnostic programming allows the programmer to specify each information flow policy once instead of as checks and filters across the program. A language runtime is responsible for managing dependencies between policies and with the rest of the program. Unlike secure multi-execution, policy-agnostic runtimes modify runtime behavior in order to produce outputs that adhere to policies. Policy-agnostic programming provides a layer of policy management on top of secure multi-execution to address the issues of policy spaghetti and leaky enforcement. Policy-agnostic programming may be used in conjunction with prior work on preventing information leaks. In fact, the original semantics for the Jeeves programming language for policy-agnostic programming [106] is based on symbolic execution and developed independently of the multi-execution work. One of the contributions of this thesis includes a faceted semantics for policy-agnostic programming [10] that combines multi-execution with policy-agnostic programming.

Jeeves's guarantees have some interesting theoretical dimensions with respect to prior work. Because Jeeves allows mutual dependencies between sensitive values and policy computations, the non-interference property is similar to a *declassification* property. Prior work on secure declassification [30, 62] provides mechanisms for describing policies that describe when sensitive information may be revealed; for instance, to check passwords or after all bids have been submitted in a sealed auction. Formally characterizing declassification and robustness, the property that an attacker cannot affect the state in such a way that it leaks information [109], are not within the scope of this thesis. It is also not within the scope of this thesis to address what happens when policies need to change [99] or when information needs to be erased or made less accessible based on the needs of the system [29].

2.2 Comparing to Other Language-Based Techniques

The policy-agnostic approach provides a new way to 1) specify behavior with respect to information flow concerns and 2) automate the implementation of these specifications. In this section, we describe why existing programming paradigms are not enough for automating information flow.

2.2.1 Restrictions of Aspect-Oriented Programming

It may seem that we can automate information flow concerns using *aspect-oriented programming* [52], which has the goal of factoring out *cross-cutting concerns* that recur across a program. Frameworks for aspect-oriented programming allow the programmer to separately specify *component* code and *aspect* code. The frameworks automate the process of weaving in aspect code at component boundaries. The programmer specifies a *pointcut*, a set of *join points* in the program where aspect code should be executions. Aspects have restricted influence in the rest of the program because they do not interact within components. Even with Smith's *generative approach* that allows the programmer to express aspects as logical invariants and reconstructs the call stack to implement them [95, 96], aspects are useful for concerns that are more syntactically than semantically intertwined with the program, for example error logging and maintaining consistency between a data model and graphical views.

Aspect-oriented programming is not a good fit for information flow because customizing program behavior based on information flow requires an approach that is more semantically intertwined with the host language. Consider the checks required in enforcing that Carol's surprise party details are not leaked. The program needs a permission check not only when event details are accessed directly, but whenever *derived values* are accessed. Derived values prevent us from encoding information flow checks as aspects. Determining which values are derived values requires tracking the flow of information within components, as sensitive values can affect computations both directly, by being part of the computations themselves, and indirectly, by being captured in assignments under sensitive conditional checks.

Aspects are insufficiently expressive because they only support weaving in code at component boundaries.

2.2.2 Limitations of Prior Work in Executing Specifications

In thinking about how to automate information flow, we can also look to work in *executable specifications* [71], for instance Squander [69]. The goal of executing specifications is to allow the programmer to provide properties the code should adhere instead of writing programs operationally. The work in executing specifications focuses on replacing subroutines with executable specifications. With policy-agnostic programming, specifications are associated with computations over specific program values. The specifications do not need to describe self-contained subroutines: they may have mutual dependencies with the rest of the program.

Towards automating program concerns, we can look to *angelic nondeterminism*. Floyd proposed an angelic nondeterminism that simplifies specifications for backtracking algorithms [40]. Bodik *et al.* [18] show this operator is useful in the development and testing of deterministic programs. The SPR [66] and Prophet [65] systems demonstrate the operator is useful for bug patching. Angelic nondeterminism is typically expensive to support because it involves searching over an extensive program space. Sensitive values in Jeeves behave similarly to angelic values, but their specifications define a smaller search space than prior work and thus the algorithms used to resolve their behavior are also different.

The best candidate from existing work for automating information flow concerns is *constraint functional programming*. Mück *et al.* [73] present a calculus that integrates constraints into a functional programming model. Similar programming models have been implemented in Mercury [46], Escher [64], and Curry [45]. Constraint functional programming languages are sufficiently expressive to support the encoding of information flow concerns, but the issue is performance. Prior work on constraint functional languages supports general constraints. In order to do so, the runtimes execute the programs as logic programs, performing runtime search to

produce results. In Jeeves, on the other hand, constraints may only affect the values of labels, allowing the Jeeves runtime to search over a limited, decidable space.

Related work in dynamic approaches produces values to adhere to a specification, but the approaches are too restrictive for expressing information flow. Program repair approaches such as Demsky's data structure repair [33], the Plan B [91] system for dynamic contract checking, and Kuncak *et al.*'s synthesis approach [57] all target localized program expressions rather than global, intertwined concerns.

2.2.3 Relationship to Variational Data Structures

Variational data structures [103] are similar in philosophy to sensitive values in policy-agnostic programs. Variational data structures encapsulate properties related to program customization. Variational data structures are useful for parameterizing programs with respect to different possible options. For instance, for planning a trip, a program may be parametric with respect to specific itinerary or cost requirements. While this work focuses on representations of variations and variation-aware algorithms, the policy-agnostic programming work focuses on the guarantees a language semantics can provide when interacting with values that encapsulate multiple views. By making the language runtime aware of sensitive values, policy-agnostic programming is able to provide strong runtime guarantees.

2.2.4 Relationship to Acceptability-Oriented Programming

The idea of continuing program execution despite entering a failure state is related to *acceptability-oriented programming* [85]. Rinard argues that instead of attempting to build a system free of errors, the system designer should identify the key properties the execution must satisfy in order to be acceptable to users. Rinard *et al.* [86] demonstrated the effectiveness of the approach for enhancing server availability. The follow-up work has moved towards characterizing the approximate behavior of programs when executing with an arbitrary specification [25]. Our work focuses instead on obtaining a specification of alternate behavior from the programmer and

automating its implementation.

2.2.5 Relationship to Declarative Domain-Specific Languages

Prior work uses declarative constraints to address domain-specific programming issues. Frenetic [41, 42] provides a declarative query language for software-defined networking. Engage [38] uses constraints to mitigate programmer burden in configuring, installing, and managing software deployments. In these domains, constraints describe the entire program execution and cannot be mixed into the rest of the program. An interesting direction of future work involves examining whether a policy-agnostic approach could improve expressiveness or performance in these domains.

Part I

Policy-Agnostic Programming for Information Flow

Chapter 3

Policy-Agnostic Programming in the Jacqueline Web Framework

A major advantage of the policy-agnostic approach is that the programmer can write policy-enforcing code that looks similar to policy-free code and rely on the runtime to customize program behavior. In this chapter, we introduce the programming model of the Jacqueline web framework [105] through an example. On the surface, Jacqueline looks similar to a standard *model-view-controller* (MVC) web framework. In a standard MVC framework, the *model* describes the data, the *view* describes the page layouts, and the *controller* describes computation over the data to produce views. In Jacqueline, the programmer additionally provides specifications of information flow policies alongside data schemas in the model.

With Jacqueline, we present a general policy-agnostic approach to model-view-based controller web programming. Our semantics and guarantees are language-agnostic with respect to the language runtime and work with any relational database. We implemented Jacqueline as an extension of the Django Python web framework, but we could have chosen a variety of other languages and implementation strategies. In this chapter, we show how programming in Jacqueline is similar to programming in vanilla Django without policies. Jacqueline additionally provides all the benefits of automatic end-to-end policy enforcement.

```python
class Event(JacquelineModel):
  name = CharField(max_length=256)
  location = CharField(max_length=512)
  time = DateTimeField()
  description = CharField(max_length=1024)

  # Public value for name field.
  @staticmethod
  def jacqueline_get_public_name(event):
    return "[private event]"

  # Public value for location field.
  @staticmethod
  def jacqueline_get_public_location(event):
    return "Undisclosed location"

  # Policies for name and location fields.
  @staticmethod
  @label_for('name', 'location')
  @jacqueline
  def jacqueline_restrict_event(event, ctxt):
    return (EventGuest.objects.get(
      event=self, guest=ctxt) != None)

class EventGuest(JacquelineModel):
  event = ForeignKey(Event, null=True)
  guest = ForeignKey(UserProfile, null=True)
```

Figure 3-1: Jacqueline schema fragment for calendar events. Policy code is shown with a gray background.

3.1 Schemas and Policies in Jacqueline

Recall our calendar example from Chapters 1 and 2. Alice and Bob want to plan a surprise party for Carol at 7pm next Tuesday at Schloss Dagstuhl and need to show different versions of event information to different parties. In this section, we show how Jeeves allows us to specify that the secret details of the party are sensitive and that only guests should be able to view this information. We show how Jeeves also allows us to specify that everyone else sees an alternate default value, for instance the coarser-grained country “Germany” for the location. In the next section, we show

how the Jeeves runtime enforces policies in all computations, including searches such as "Who are all of my friends at Schloss Dagstuhl at 7pm next Tuesday?"

A Jacqueline application consists of the model specifying the data and policies, policy-agnostic controller code, and policy-agnostic view code. In the model code, the programmer associates *policies* with data fields in *schemas* specifying the data types. Policies have two components: 1) an *information flow policy* describing when data fields may be visible to a given viewer and 2) an *alternate default value* that the program can use if the programmer does not have access to the sensitive value. The alternate default value can be a coarsening of the sensitive value: for instance, instead of the actual location "Schloss Dagstuhl," the corresponding town "Wadern." The Jacqueline runtime simulates simultaneous multiple executions on the different views to ensure that if a viewer does not have access to the sensitive data field, outputs are computed as if the sensitive field was the alternate value.

Continuing with our calendar example, we show a sample schema for the Event and EventGuest data objects in Figure 3-1. 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 23), with foreign keys to the Event and UserProfile (definition not shown) tables, is exactly as it would be in Django.

3.1.1 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 other viewers. The Jacqueline runtime simulates simultaneous executions on both views in order to guarantee that if a viewer does not have access to the secret view, the

system will produce 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 8 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]" in 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.

3.1.2 Specifying Policies

The programmer specifies *information flow policies* that determine whether a given viewer sees the actual data field or the alternate default value. On line 20 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 23) whether a given guest is associated with the event.

```
1 class JeevesQuerySet(QuerySet):
2   all()
3   delete()
4   filter(**kwargs)
5   orderby(**kwargs)
6   get(**kwargs)
7 
8 class JacquelineModel(Model):
9   create(*args, **kwargs)
10  delete()
11  save(*args, **kwargs)
```

Figure 3-2: 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.

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.

3.2 Policy-Agnostic Controller Code

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 3-2 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.

Jacqueline uses *faceted execution* [9, 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 *indirect 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.

3.3 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. We describe the mechanism for handling these dependencies in Section 4.4.

Chapter 4

Jeeves, a Language for Automatically Enforcing Information Flow Policies

Underlying Jacqueline is Jeeves, a language that automatically enforces information flow policies. We intend for Jeeves to be the "assembly language" for policy-agnostic programming, containing key constructs for manipulating sensitive values and policies. Jeeves extends the imperative λ-calculus with *faceted* sensitive values ($\langle \ell \; ? \; Exp_H : Exp_L \rangle$), a **label** construct for introducing labels that guard access to facets, and a **restrict** construct for introducing policies on labels. In this chapter, we present Jeeves using an ML-like concrete syntax, shown in Figure 4-1.

4.1 Sensitive Values and Policies

Jeeves programs associate information flow policies with sensitive values, allowing other functionality to be policy-agnostic. Recall from Chapter 3 that we represent sensitive values as *faceted values* encapsulating multiple views. The following faceted value behaves as either "Carol's surprise party" or "[private event]", depending on what the runtime assigns to the guard a:

```
⟨ a ? "Carol's surprise party" : "[private event]" ⟩
```

$$
\begin{array}{lcll}
x & & & \text{variables} \\
\ell & & & \text{labels} \\
p, r & & & \text{primitives, records} \\
k & ::= & p \mid r & \text{principals} \\
Label & ::= & \textbf{true} \mid \textbf{false} & \text{labels} \\
\tau & ::= & \textbf{int} \mid \textbf{bool} \mid \textbf{string} & \text{types} \\
 & \mid & \textbf{record}\ \overrightarrow{x : \tau} & \\
 & \mid & \tau_2 \to \tau_2 \mid \tau\ \textbf{ref} & \\
 & \mid & Label & \\
Exp & ::= & x \mid p \mid r \mid k & \text{expressions} \\
 & \mid & \lambda x : \tau . Exp & \\
 & \mid & Exp_1\ (\text{op})\ Exp_2 & \\
 & \mid & \textbf{if}\ Exp_1\ \textbf{then}\ Exp_t\ \textbf{else}\ Exp_f & \\
 & \mid & Exp_1\ Exp_2 & \\
 & \mid & \textbf{ref}\ Exp & \\
 & \mid & !\ Exp \mid x := Exp\ \textbf{in}\ Exp_b & \\
 & \mid & \langle \ell\ ?\ Exp_{high}\ :\ Exp_{low} \rangle & \\
 & \mid & \textbf{label}\ \ell\ \textbf{in}\ Exp & \\
 & \mid & \textbf{restrict}\ \ell :\ Exp_p\ \textbf{in}\ Exp & \\
Stmt & ::= & \textbf{let}\ x : \tau = Exp\ \textbf{in}\ Exp_b & \text{statements} \\
 & \mid & \textbf{print}\ \{Exp_c\} Exp &
\end{array}
$$

Figure 4-1: Jeeves syntax.

The string "Carol's surprise party" is the *secret*, or *high-confidentiality* facet available to viewers with permissions. The string "[private event]" is the *public*, or *low-confidentiality* facet available to other viewers. The Jacqueline web framework is responsible for creating faceted values where the actual field value is the secret facet and the alternate default value is the public facet. A Jeeves program has the flexibility of introducing faceted values at any point.

To enforce policies, the Jeeves runtime assigns values to labels based on program-defined policies. Labels take on the values { **true**, **false** }, where **true** corresponds to the high-confidentiality value and **false** corresponds to the low-confidentiality value. The program introduces labels using the **label** construct and policies using the **restrict** construct. This policy on label a says that the user must be the user alice to have high-confidentiality status:

```
label a in
  restrict a: λ(viewer: user).(viewer == alice) in ...
```

Policies are functions that take an argument corresponding to the output context and return a Boolean value describing when the implicated label may take on the value **true**. As mentioned before, the output context for policies can be of arbitrary type, determined by the program. In Jacqueline, the web framework helps keep track of the output context.

In Jeeves, the labels are assigned based on the conjunction of all relevant policies evaluating to **true**. For instance, the following two policies would indicate that only friends of users alice and bob have permission to view the high-confidentiality facet:

```
restrict a: λ(viewer: user).isFriends(viewer, alice) in
restrict a: λ(viewer: user).isFriends(viewer, bob) in ...
```

This makes it so that even if policies are declared elsewhere, there is a guarantee that they can only decrease permissions on high-confidentiality facets. This supports a distributed programming model.

4.2 Policy-Agnostic Programs

As we described with Jacqueline execution, the Jeeves runtime simulates simultaneous executions on sensitive values to customize program behavior based on the policies and the viewer. For instance, consider the following code:

```
"Your friend is going to " +
  ⟨ a ? "Carol's surprise party" : "[private event]" ⟩;
```

The Jeeves runtime evaluations operations involving faceted values by evaluating the operations on each of the facets and putting the resulting values into a resulting faceted value. Evaluation of the code above yields:

```
⟨ a ? "Your friend is going to Carol's surprise party"
    : "Your friend is going to [private event]" ⟩
```

During program evaluation, the Jeeves runtime ensures that only users with the appropriate permissions can see the string with "Carols' surprise party". Everyone else sees the string containing "[private event]".

4.3 Producing Concrete Values

We described how in Jacqueline, the web framework customizes outputs based on the policies and the viewer. Underlying this functionality is Jeeves's *concretization* mechanism. For effectful statements such as **print**, the Jeeves runtime customizes the output based on the policies and viewer. Effectful statements take a parameter corresponding to the output context:

```
let msg: string = "Your friend is going to " + event;

(* Output: "Your friend is going to Carol's surprise party" *)
print { alice } msg;

(* Output: "Your friend is going to [private event]" *)
print { carol } msg;
```

To produce an assignment to labels, the Jeeves system translates this rule to the declarative constraint $\mathsf{a} \Rightarrow \mathsf{viewer} == \mathsf{alice}$: in order for label a to be assigned the

```
1 let mkEvent (name: string) (description: string)
2   (hosts: user list) (guests: user list)
3   (isFinalized: bool): event =
4   label canSeeEvent in
5   let e = ⟨ canSeeEvent ?
6               { name = name; description = description
7               ; hosts = hosts; guests = guests
8               ; isFinalized = isFinalized }
9             : { name = "[private event]"
10              ; description = "[private]"; hosts = []; guests = []
11              ; isFinalized = true } ⟩ in
12    restrict canSeeEvent: λ(viewer: user).(
13      ((isMember viewer e.hosts) or
14        (e.isFinalized and (isMember viewer e.guests))))
```

Figure 4-2: “Constructor” code for an event record in Jeeves.

value **true**, the value of the variable viewer must be equivalent to the value of the variable alice. The runtime collects policies until computation sinks. Each policy could additionally restrict a to be **false**. For maximal functionality, the Jeeves system tries to assign labels to **true**, setting labels to **false** only if the policies require it. Assigning all labels to **false** always yields a consistent solution because policies can only require labels to be **false**. This means that in Jacqueline, it is always acceptable to show outputs computed using the default values of fields.

4.4 Handling Dependencies Between Sensitive Values and Policies

In Chapters 1 and 2, we describe how one source of information leaks is *leaky enforcement*, leaks through policy enforcement. One reason these leaks arise is because there is a stratification between sensitive values and permission checks, making the programmer trusted to manage mutual dependencies between policies and sensitive values. In this section, we describe how Jeeves policies are expressive enough to capture these dependencies and how the Jeeves runtime manages them. As long as the viewer properly associates policies with sensitive values, the Jeeves runtime

produces results that adhere to all policies.

In Figure 4-2 we show an example "constructor" that takes values for the fields of an event value and creates a faceted event value associated with the appropriate policy. This code introduces a label canSeeEvent (line 4), uses the label to create a sensitive value that either shows the event information or shows default information, and specifies a policy (line 12) on the event. The policy says that in order for a viewer to see the actual event information, they must either be an event host or the event has been finalized and the viewer is a guest. As long as the programmer attaches the policies correctly, Jeeves handles the interaction of the policies.

In our example, there is a mutual dependency between the policy protecting guest list and the sensitive guest list itself. A viewer may view the contents of the guest list only if the viewer is a member of the guest list. To understand what happens in this situation, let us consider a simplified example:

```
let u: user =
  label b in
  let v: user = ⟨ b ? alice : nobody ⟩
  restrict b: λ(viewer: user).(viewer == v) in
    v
```

This code creates a user that is either alice or nobody guarded by label b and associates with b a policy that the viewer needs to be equal to this alice-or-nobody value. Now let us consider what happens when we print the name field of this user value:

```
print {alice} u.name
```

The value u.name evaluates to $\langle\, \mathsf{b}\ ?\ \mathsf{alice.name} : \mathsf{nobody.name}\,\rangle$. To resolve the output, the runtime needs to evaluate the policy associated with b. We evaluate the policy applied to the output context as follows:

$$
\begin{aligned}
& \mathsf{b} \Rightarrow \mathsf{alice} == \langle\, \mathsf{b}\ ?\ \mathsf{alice} : \mathsf{nobody}\,\rangle \\
\hookrightarrow\quad & \mathsf{b} \Rightarrow \langle\, \mathsf{b}\ ?\ \textbf{true} : \textbf{false}\,\rangle \\
\hookrightarrow\quad & \langle\, \mathsf{b}\ ?\ \mathsf{b} \Rightarrow \textbf{true} : \mathsf{b} \Rightarrow \textbf{false}\,\rangle
\end{aligned}
$$

The circular dependency makes it so that both outcomes are valid: b is **true** and we have **true** ⇒ **true**, or b is **false** and we have **false** ⇒ **false**. Stepping up a level, this means that the permissions allow either for user u to be alice or nobody. The behavior

we would like is for the runtime to show the higher-confidentiality value as long as it is allowed. To handle these dependencies, we have designed the Jeeves runtime for *maximal functionality*. If the policies allow a label to be **true** or **false**, the runtime will assign it to be **true**. This makes it so that the runtime shows the result from computing on secret values unless the policies disallow it.

The Jeeves runtime resolves circular dependencies by finding the *maximally functional fixed point*. Whenever there is a circular dependency, the runtime needs to find a fixed point to the constraints. The circular dependency leads to *multiple* fixed points. For the example above, the two solutions are b is **true**, allowing the user u to be the secret value alice, or b is **false**, requiring u to be the default value nobody. When there are two fixed points, there is a clear ordering between them. We discourage writing code with more than two fixed points and have implemented a dynamic analysis to produce a warning when these cases are detected.

We designed Jeeves so that constraint resolution is decidable and constraints are always consistent. The constraints that the Jeeves runtime produces take the form $\ell \Rightarrow P$, where ℓ is a Boolean label and P is a predicate containing only labels as free variables. The runtime evaluates all other variables in terms of runtime state at the time of output. The constraint environment is decidable because it contains only Boolean free variables. The constraints are always consistent because constraints can only force labels to be assigned to **false**. Assigning all labels to **false** is always consistent with any set of constraints of this form.

4.5 Policy Language Limitations

Our policy language has the following limitations:

- The policy language only allows restrictions that prevent the disclosure of the secret facet. There is no mechanism for specifying that the secret facet must be visible. This restriction prevents policy contradictions.

- We have formulated our policy guarantees with respect to individual output

instances. As a result, there is no way to specify policies that describe behaviors across multiple outputs. Jeeves does not, for instance, support policies that allow value A or B to be shown to a given viewer, but not both. The Jeeves runtime will show either A or B for each output, with no guarantees of showing only A or B. Providing guarantees across outputs is outside the scope of this thesis, but can be implemented by keeping track of prior outputs.

- Policy revocation is outside the scope of this thesis.

- As previously mentioned, there is only a ranking between fixed point solutions when there is a circular dependency between two labels. There is a clear ordering between two fixed point solutions, as they arise when a policy depends on the sensitive value it protects. This is not the case when there are more fixed points. To address this problem, we discourage writing code with more than two fixed points have implemented a dynamic analysis to produce a warning when these cases are detected. An area of future exploration involves determinizing the policy semantics based on programming patterns and needs.

Part II

Reasoning about Policy-Agnostic Programs

Chapter 5

Semantics and Guarantees for Faceted Execution of Jeeves

Another contribution of this thesis is a formalization of what it means for the language runtime to enforce information flow policies. In this chapter, we present the dynamic semantics for Jeeves and use it to prove guarantees of termination-insensitive non-interference and policy compliance. The semantics describe how Jeeves simulates simultaneous executions over facets of sensitive values in order to automatically enforce information flow policies. The formulation of the security properties takes into account that the policies and the viewer may depend on computations involving sensitive values.

5.1 Core Semantics

We model the semantics of Jeeves with λ^{jeeves}, a simple core language that extends the faceted execution semantics of Austin and Flanagan [9] with a declarative policy language for confidentiality. The λ^{jeeves} semantics describes how to evaluate faceted values, store policies, and use the policy environment to provide assignments to labels for producing concrete outputs. We use these semantics to prove non-interference and policy compliance guarantees.

We show the source syntax in Figure 5-1. The language λ^{jeeves} extends the λ-

Syntax:

$e ::=$		*Term*
	x	variable
	c	constant
	$\lambda x.e$	abstraction
	$e_1\ e_2$	application
	$\text{ref}\ e$	reference allocation
	$!e$	dereference
	$e := e$	assignment
	$\langle k\ ?\ e_1 : e_2 \rangle$	faceted expression
	$\text{label}\ k\ \text{in}\ e$	label declaration
	$\text{restrict}(k, e)$	policy specification
$S ::=$		*Statement*
	$\text{let}\ x = e\ \text{in}\ S$	let statement
	$\text{print}\ \{e\}\ e$	print statement
$c ::=$		*Constant*
	f	file handle
	b	boolean
	i	integer
x, y, z		*Variable*
k, l		*Label*

Standard encodings:

$$
\begin{aligned}
\mathit{true} &\stackrel{\text{def}}{=} \lambda x.\lambda y.x \\
\mathit{false} &\stackrel{\text{def}}{=} \lambda x.\lambda y.y \\
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 &\stackrel{\text{def}}{=} (e_1\ (\lambda d.e_2)\ (\lambda d.e_3))\ (\lambda x.x) \\
\text{if } e_1 \text{ then } e_2 &\stackrel{\text{def}}{=} \text{if } e_1 \text{ then } e_2 \text{ else } 0 \\
\text{let } x = e_1 \text{ in } e_2 &\stackrel{\text{def}}{=} (\lambda x.e_2)\ e_1 \\
e_1\ \wedge_f\ e_2 &\stackrel{\text{def}}{=} \lambda x.e_1\ x\ \wedge\ e_2\ x \\
e_1\ \wedge\ e_2 &\stackrel{\text{def}}{=} \text{if } e_1 \text{ then } e_2 \text{ else } \mathit{false}
\end{aligned}
$$

Figure 5-1: The source language λ^{jeeves}

Runtime Syntax

$$
\begin{array}{rclcl}
e & \in & \mathit{Expr} & ::= & \ldots \mid a \\
\Sigma & \in & \mathit{Store} & = & (\mathit{Address} \to_p \mathit{Value}) \cup (\mathit{Label} \to \mathit{Value}) \\
R & \in & \mathit{RawValue} & ::= & c \mid a \mid (\lambda x.e) \\
a & \in & \mathit{Address} & & \\
V & \in & \mathit{Val} & ::= & R \mid \langle k \mathbin{?} V_1 : V_2 \rangle \\
h & \in & \mathit{Branch} & ::= & k \mid \overline{k} \\
pc & \in & \mathit{PC} & = & 2^{\mathit{Branch}}
\end{array}
$$

Figure 5-2: Runtime syntax for λ^{jeeves}evalution.

calculus with expressions for allocating references (ref e), dereferencing (!e), assignment (e_1:=e_2), creating faceted expressions ($\langle k \mathbin{?} e_1 : e_2 \rangle$), specifying policy (restrict(k, e)), and declaring labels (label k in e). Additional statements exist for let-statements (let $x = e$ in S) and printing output (print $\{e_1\}$ e_2). Conditionals are encoded in terms of function application.

In λ^{jeeves}, values V contain *faceted values* of the form

$$\langle k \mathbin{?} V_{\mathrm{H}} : V_{\mathrm{L}} \rangle$$

A viewer authorized to see k-sensitive data will observe the private facet V_{H}. Other viewers will instead see V_{L}. For example, the value $\langle k \mathbin{?} 42 : 0 \rangle$ specifies a value of 42 that should only be viewed when k is true according to the policy associated with k. When the policy specifies false, the observed value should instead be 0.

A *program counter label pc* records when execution is influenced by public or private facets. For instance, in the conditional test

$$\text{if } (\langle k \mathbin{?} \mathit{true} : \mathit{false} \rangle) \text{ then } e_1 \text{ else } e_2$$

our semantics needs to evaluate both e_1 and e_2. The label k is added to pc during the evaluation of e_1. By doing so, our semantics records the influence of k on this computation. Similarly, $\overline{k}$ is added to pc during the evaluation of e_2 to record that the execution should have no effects observable to k. A *branch h* is either a label k

Expression Evaluation Rules $\boxed{\Sigma, e \Downarrow_{pc} \Sigma', V}$

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

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

$$\frac{\Sigma, e \Downarrow_{pc} \Sigma', V \quad V' = deref(\Sigma', V, pc)}{\Sigma, !e \Downarrow_{pc} \Sigma', V'} \quad \text{[F-DEREF]}$$

$$\frac{\Sigma, e_1 \Downarrow_{pc} \Sigma_1, V_1 \quad \Sigma_1, e_2 \Downarrow_{pc} \Sigma_2, V' \quad \Sigma' = assign(\Sigma_2, pc, V_1, V')}{\Sigma, e_1 := e_2 \Downarrow_{pc} \Sigma', V'} \quad \text{[F-ASSIGN]}$$

$$\frac{\Sigma, e_1 \Downarrow_{pc} \Sigma_1, V_1 \quad \Sigma_1, e_2 \Downarrow_{pc} \Sigma_2, V_2 \quad \Sigma_2, (V_1\ V_2) \Downarrow_{pc}^{\mathsf{app}} \Sigma', V'}{\Sigma, (e_1\ e_2) \Downarrow_{pc} \Sigma', V'} \quad \text{[F-APP]}$$

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

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

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

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

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

Figure 5-3: λ^{jeeves} expression evaluation.

Auxiliary Functions

$$
\begin{array}{lcll}
\mathit{deref} : \mathit{Store} \times \mathit{Val} \times \mathit{PC} & \rightarrow & \mathit{Val} & \\
\mathit{deref}(\Sigma, a, pc) & = & \Sigma(a) & \\
\mathit{deref}(\Sigma, \langle k\ ?\ V_{\mathrm{H}} : V_{\mathrm{L}} \rangle, pc) & = & \begin{cases} \mathit{deref}(\Sigma, V_{\mathrm{H}}, pc) & \text{if } k \in pc \\ \mathit{deref}(\Sigma, V_{\mathrm{L}}, pc) & \text{if } \overline{k} \in pc \\ \langle\!\langle\, k\ ?\ \mathit{deref}(\Sigma, V_{\mathrm{H}}, pc) : \mathit{deref}(\Sigma, V_{\mathrm{L}}, pc)\,\rangle\!\rangle & \text{o/w} \end{cases}
\end{array}
$$

$$
\begin{array}{lcll}
\mathit{assign} : \mathit{Store} \times \mathit{PC} \times \mathit{Val} \times \mathit{Val} & \rightarrow & \mathit{Store} & \\
\mathit{assign}(\Sigma, pc, a, V) & = & \Sigma[a := \langle\!\langle\, pc\ ?\ V : \Sigma(a)\,\rangle\!\rangle] & \\
\mathit{assign}(\Sigma, pc, \langle k\ ?\ V_{\mathrm{H}} : V_{\mathrm{L}} \rangle, V) & = & \Sigma' & \text{where } \Sigma_1 = \mathit{assign}(\Sigma,\ pc \cup \{k\}, V_{\mathrm{H}}, V) \\
& & & \text{and } \quad \Sigma' = \mathit{assign}(\Sigma_1, pc \cup \{\overline{k}\}, V_{\mathrm{L}}, V)
\end{array}
$$

Figure 5-4: Auxiliary functions for λ^{jeeves}evaluation.

or its negation $\overline{k}$. Therefore pc is a set of branches that never contains both k and $\overline{k}$, since that would reflect influences from both the private and public facet of a value.

The operation $\langle\!\langle\, pc\ ?\ V_1 : V_2\,\rangle\!\rangle$ creates a faceted value. The value V_1 is visible when the specified policies correspond with *all* branches in pc. Otherwise, V_2 is visible instead.

$$
\begin{array}{lcl}
\langle\!\langle\, \emptyset\ ?\ V_n : V_o\,\rangle\!\rangle & \stackrel{\text{def}}{=} & V_n \\
\langle\!\langle\, \{k\} \cup \mathit{rest}\ ?\ V_n : V_o\,\rangle\!\rangle & \stackrel{\text{def}}{=} & \langle k\ ?\ \langle\!\langle\, \mathit{rest}\ ?\ V_n : V_o\,\rangle\!\rangle : V_o \rangle \\
\langle\!\langle\, \{\overline{k}\} \cup \mathit{rest}\ ?\ V_n : V_o\,\rangle\!\rangle & \stackrel{\text{def}}{=} & \langle k\ ?\ V_o : \langle\!\langle\, \mathit{rest}\ ?\ V_n : V_o\,\rangle\!\rangle \rangle
\end{array}
$$

For example, $\langle\!\langle\, \{k, l\}\ ?\ V_{\mathrm{H}} : V_{\mathrm{L}}\,\rangle\!\rangle$ returns $\langle k\ ?\ \langle l\ ?\ V_{\mathrm{H}} : V_{\mathrm{L}} \rangle : V_{\mathrm{L}} \rangle$. We occasionally abbreviate $\langle\!\langle\, \{k\}\ ?\ V_{\mathrm{H}} : V_{\mathrm{L}}\,\rangle\!\rangle$ as $\langle\!\langle\, k\ ?\ V_{\mathrm{H}} : V_{\mathrm{L}}\,\rangle\!\rangle$.

The semantics are defined via the big-step evaluation relation:

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

This relation evaluates an expression e in the context of a store Σ and program counter label pc. It returns a modified store Σ' reflecting updates and a value V. We show the runtime syntax in Figure 5-2, the evaluation rules in Figure 5-3, and the auxiliary functions for dereference and assignment in Figure 5-4.

Our language includes support for reference cells, which introduce additional complexities in handling implicit flows. The rule [F-REF] handles reference allocation (ref e). It evaluates an expression e, encoding any influences from the program counter pc to the value V, and adds it to the store Σ' at a fresh address a. Facets in V inconsistent with pc are set to 0. (Critically, to maintain non-interference, $\Sigma(a) = 0$ for all a not in the domain of Σ.)

The rule [F-DEREF] for dereferencing ($!e$) evaluates the expression e to a value V, which should either be an address or a faceted values where all of the "leaves" are addresses. The rule uses a helper function $deref(\Sigma', V, pc)$ (defined in Figure 5-4), which takes the addresses from V, retrieves the appropriate values from the store Σ', and combines them in the return value V'. As an optimization, addresses that are not compatible with pc are ignored.

The rule [F-ASSIGN] for assignment ($e_1 {:=} e_2$) is similar to [F-DEREF]. It evaluates e_1 to a possibly faceted value V_1 corresponding to an address and e_2 to a value V'. The helper function $assign(\Sigma_2, pc, V_1, V')$ defined in Figure 5-4 decomposes V_1 into separate addresses, storing the appropriate facets of V' into the returned store Σ'. The changes to the store may come from both V_1 and pc.

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

When a faceted expression $\langle k \; ? \; e_1 : e_2 \rangle$ is evaluated, both sub-expressions must be evaluated in sequence, as per the rule [F-SPLIT]. The influence of k is added to the program counter for the evaluation of e_1 to V_1 and $\overline{k}$ for the evaluation of e_2 to V_2, tracking the branch of code being taken. The results of both evaluations are joined together in the operation $\langle\langle \, k \; ? \; V_1 : V_2 \, \rangle\rangle$. As an optimization, only one expression is

evaluated if the program counter already contains either k or $\overline{k}$, as indicated by the rules [F-LEFT] and [F-RIGHT].

Function application (e_1 e_2) is somewhat complex in the presence of faceted values. The rule [FA-APP] evaluates e_1 to V_1, which should either be a lambda or a faceted value containing lambdas, and evaluates e_2 to the function argument V_2. It then delegates the application $(V_1\ V_2)$ to an auxiliary relation defined in Figure 5-5:

$$\Sigma, (V_1\ V_2) \Downarrow_{pc}^{\mathtt{app}} \Sigma', V'$$

This relation breaks apart faceted values and tracks the influences of the labels through the rules [FA-SPLIT], [FA-LEFT], and [FA-RIGHT] in a similar manner to the rules [F-SPLIT], [F-LEFT], and [F-RIGHT] discussed previously. The actual application is handled by the [FA-FUN] rule. The body of the lambda $(\lambda x.e)$ is evaluated with the variable x replaced by the argument V.

Conditional branches (if e_1 then e_2 else e_3) are Church-encoded as function calls for the sake of simplicity. However, Figure 5-6 shows direct rules for evaluating conditionals in the presence of faceted values. Under the rule [F-IF-SPLIT], If the condition e_1 evaluates to a faceted value $\langle k\ ?\ V_H : V_L \rangle$, the if statement is evaluated twice with V_H and V_L as the conditional tests.

While expressions handle most of the complexity of faceted values, statements in λ^{jeeves} illustrate how faceted values may be concretized when exporting data to an external party. The semantics for statements are defined via the big-step evaluation relation:

$$\Sigma, S \Downarrow V_p, f : R$$

The rules for statements are specified in Figure 5-5. The rule [F-LET] handles let expressions (let $x = e$ in S), evaluating an expression e to a value V, performing the proper substitution in statement S. The rule [F-PRINT] handles print statements (print $\{e_1\}$ e_2), where the result of evaluating e_2 is printed to the channel resulting from the evaluation of e_1. Both the channel V_f and the value to print V_c may be faceted values, and furthermore, we must select the facets that correspond with

Application Rules $\boxed{\Sigma, (V_1\ V_2) \Downarrow^{\mathsf{app}}_{pc} \Sigma', V'}$

[FA-FUN]

$$\frac{\Sigma, e[x := V] \Downarrow_{pc} \Sigma', V'}{\Sigma, ((\lambda x.e)\ V) \Downarrow^{\mathsf{app}}_{pc} \Sigma', V'}$$

[FA-LEFT]

$$\frac{k \in pc \qquad \Sigma, (V_{\mathrm{H}}\ V_2) \Downarrow^{\mathsf{app}}_{pc} \Sigma', V}{\Sigma, (\langle k\ ?\ V_{\mathrm{H}} : V_{\mathrm{L}} \rangle\ V_2) \Downarrow^{\mathsf{app}}_{pc} \Sigma', V}$$

[FA-SPLIT]

$$\frac{\begin{array}{c} k \notin pc \qquad \bar{k} \notin pc \\ \Sigma, (V_{\mathrm{H}}\ V_2) \Downarrow^{\mathsf{app}}_{pc \cup \{k\}} \Sigma_1, V'_{\mathrm{H}} \\ \Sigma_1, (V_{\mathrm{L}}\ V_2) \Downarrow^{\mathsf{app}}_{pc \cup \{\bar{k}\}} \Sigma', V'_{\mathrm{L}} \\ V' = \langle\langle\, k\ ?\ V'_{\mathrm{H}} : V'_{\mathrm{L}}\, \rangle\rangle \end{array}}{\Sigma, (\langle k\ ?\ V_{\mathrm{H}} : V_{\mathrm{L}} \rangle\ V_2) \Downarrow^{\mathsf{app}}_{pc} \Sigma', V'}$$

[FA-RIGHT]

$$\frac{\bar{k} \in pc \qquad \Sigma, (V_{\mathrm{L}}\ V_2) \Downarrow^{\mathsf{app}}_{pc} \Sigma', V}{\Sigma, (\langle k\ ?\ V_{\mathrm{H}} : V_{\mathrm{L}} \rangle\ V_2) \Downarrow^{\mathsf{app}}_{pc} \Sigma', V}$$

Statement Evaluation Rules $\boxed{\Sigma, S \Downarrow V_p, f : R}$

[F-LET]

$$\frac{\begin{array}{c} \Sigma, e \Downarrow_{\emptyset} \Sigma', V \\ \Sigma, S[x := V] \Downarrow V_p, f : R \end{array}}{\Sigma, \text{let } x = e \text{ in } S \Downarrow V_p, f : R}$$

[F-PRINT]

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

Figure 5-5: Faceted evaluation semantics for application and statements.

Semantics for Derived Encodings

[F-IF-TRUE]

$$\frac{\Sigma, e_1 \Downarrow_{pc} \Sigma_1, \mathit{true} \qquad \Sigma_1, e_2 \Downarrow_{pc} \Sigma', V}{\Sigma, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow_{pc} \Sigma', V}$$

[F-IF-FALSE]

$$\frac{\Sigma, e_1 \Downarrow_{pc} \Sigma_1, \mathit{false} \qquad \Sigma_1, e_3 \Downarrow_{pc} \Sigma', V}{\Sigma, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow_{pc} \Sigma', V}$$

[F-IF-SPLIT]

$$\frac{\begin{array}{c}\Sigma, e_1 \Downarrow_{pc} \Sigma_1, \langle k \mathrel{?} V_H : V_L \rangle \\ e_H = \text{if } V_H \text{ then } e_2 \text{ else } e_3 \\ e_L = \text{if } V_L \text{ then } e_2 \text{ else } e_3 \\ \Sigma_1, \langle k \mathrel{?} e_H : e_L \rangle \Downarrow_{pc} \Sigma', V\end{array}}{\Sigma, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow_{pc} \Sigma', V}$$

Auxiliary Functions

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

Figure 5-6: Semantics for derived encodings.

our specified policies. We determine the set of relevant labels through the *closeK* function, which is then used to construct e_p from the relevant policies in the store Σ_2. e_p is evaluated and applied to V_f, returning the policy check V_p that is a faceted value containing booleans. A program counter pc is chosen such that the policies are satisfied, which determines the channel f and the value to print R. Note that there exists a $pc' \in PC$ where all branches are set to fa<u>l</u>se, which may always be displayed, thereby ensuring that there is always at least one valid choice for pc.

This property allows garbage collection of policies and facets. Because the constraints are always consistent, the only set of policies relevant to an expression e to output are associated with the transitive closure of labels L_e appearing in e and the policies associated with L_e. Thus any policy associated with an out-of-scope variable may be garbaged-collected. In addition, once a policy has been set to the equivalent of $\lambda x.\mathit{false}$ for a label k, k-sensitive facets and policies cannot be used in a print statement.

5.2 Properties

We prove that a single execution with faceted values is equivalent to multiple different executions without faceted values. From this we know that if execution terminates on each facet of a sensitive value, then faceted execution terminates. We also prove that the system cannot leak sensitive information either via the output or by the choice of output channel.

5.2.1 Projection Theorem

A key property of faceted evaluation is that it simulates multiple executions. In other words, a single execution with faceted values *projects* to multiple different executions without faceted values.

$$pc : \mathit{Expr}\ \text{(with facets)} \to \mathit{Expr}\ \text{(with fewer facets)}$$

$$pc(\langle k\ ?\ e_1 : e_2 \rangle) = \begin{cases} pc(e_1) & \text{if } k \in pc \\ pc(e_2) & \text{if } \overline{k} \in pc \\ \langle k\ ?\ pc(e_1) : pc(e_2) \rangle & \text{otherwise} \end{cases}$$

$$pc(\langle k\ ?\ V_1 : V_2 \rangle) = \begin{cases} pc(V_1) & \text{if } k \in pc \\ pc(V_2) & \text{if } \overline{k} \in pc \\ pc(V_1) & \\ & \text{if } pc(V_1) = pc(V_2) \\ \langle k\ ?\ pc(V_1) : pc(V_2) \rangle & \text{otherwise} \end{cases}$$

$$pc(\dots) = \text{compatible closure}$$

We extend pc to project faceted stores $\Sigma \in \mathit{Store}$ into stores with fewer facets.

$$\begin{array}{rcl} pc : \mathit{Value} & \to & \mathit{Value} \\ pc(\Sigma) & = & \lambda a.\, pc(\Sigma(a))\ \cup\ \lambda k.\, pc(\Sigma(k)) \end{array}$$

Thus pc projection does not remove policies, it only removes some labels on

expressions or values. We say that pc_1 and pc_2 are *consistent* if

$$\neg\exists k.\ (k \in pc_1 \wedge \overline{k} \in pc_2)\ \vee\ (\overline{k} \in pc_1 \wedge k \in pc_2)$$

We note some key lemmas regarding projection.

Lemma 1. *If $V = \langle\langle\, pc\ ?\ V_1 : V_2\, \rangle\rangle$ then $\forall q \in PC$*

$$q(V) \ = \ \begin{cases} \langle\langle\, pc \setminus q\ ?\ q(V_1) : q(V_2)\, \rangle\rangle & \textit{if } q \textit{ is consistent with } pc \\ q(V_2) & \textit{otherwise} \end{cases}$$

Lemma 2. *If $V' = deref(\Sigma, V, pc)$ then $\forall q \in PC$ where q is consistent with pc, $q(V') = deref(q(\Sigma), q(V), pc \setminus q)$.*

Lemma 3. *If $\Sigma' = assign(\Sigma, pc, V_1, V_2)$ then $\forall q \in PC$*

$$q(\Sigma') \ = \ \begin{cases} assign(q(\Sigma), pc \setminus q, q(V_1), q(V_2)) \\ \qquad\qquad\qquad\qquad \textit{if } q \textit{ consistent with } pc \\ q(\Sigma) \qquad\qquad\qquad\qquad \textit{otherwise} \end{cases}$$

Lemma 4. *Suppose pc and q are not consistent and that either*

$$\begin{array}{rl} & \Sigma, e \Downarrow_{pc} \Sigma', V \\ \textit{or} & \Sigma, (V_1 V_2) \Downarrow_{pc}^{\texttt{app}} \Sigma', V \end{array}$$

Then $q(\Sigma) = q(\Sigma')$.

The following projection theorem shows how a single faceted evaluation simulates (or projects) to multiple executions, each with fewer facets, or possibly with no facets at all (if for each label k in the program, either k or $\overline{k}$ is in q).

Theorem 1 (Projection Theorem)**.** *Suppose*

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

Then for any $q \in PC$ where pc and q are consistent

$$q(\Sigma), q(e) \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$$

This theorem significantly extends the projection property of Austin and Flanagan [9], in that it supports dynamic label allocation and flexible, dynamically specified policies, and is also more general in that it can either remove none, some, or all top-level labels in a program, depending on the choice of the projection $PC\ q$. A full proof of the projection theorem is available in Appendix A.1.

5.2.2 Termination-Insensitive Non-Interference

The projection property captures that data from one collection of executions, represented by the corresponding set of branches pc, does not leak into any incompatible views, thus enabling a straightforward proof of non-interference.

Two faceted values are *pc-equivalent* if they have identical values for the set of branches pc. This notion of pc-equivalence naturally extends to stores $(\Sigma_1 \sim_{pc} \Sigma_2)$ and expressions $(e_1 \sim_{pc} e_2)$:

$$\begin{array}{rcl}
(V_1 \sim_{pc} V_2) & \text{iff} & pc(V_1) = pc(V_2) \\
(\Sigma_1 \sim_{pc} \Sigma_2) & \text{iff} & pc(\Sigma_1) = pc(\Sigma_2) \\
(e_1 \sim_{pc} e_2) & \text{iff} & pc(e_1) = pc(e_2)
\end{array}$$

The notion of pc-equivalence and the projection theorem enable a concise statement and proof of termination-insensitive non-interference.

Theorem 2 (Termination-Insensitive Non-Interference)**.**
Let pc be any set of branches. Suppose $\Sigma_1 \sim_{pc} \Sigma_2$ and $e_1 \sim_{pc} e_2$, and that:

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

Then $\Sigma_1' \sim_{pc} \Sigma_2'$ and $V_1 \sim_{pc} V_2$.

Proof. By the Projection Theorem:

$$pc(\Sigma_1), pc(e_1) \Downarrow_{\emptyset} pc(\Sigma_1'), pc(V_1)$$
$$pc(\Sigma_2), pc(e_2) \Downarrow_{\emptyset} pc(\Sigma_2'), pc(V_2)$$

The pc-equivalence assumptions imply that $pc(\Sigma_1) = pc(\Sigma_2)$ and $pc(e_1) = pc(e_2)$. Hence $pc(\Sigma_1') = pc(\Sigma_2')$ and $pc(V_1) = pc(V_2)$ since the semantics is deterministic. □

5.2.3 Termination-Insensitive Policy Compliance

While we have shown non-interference for a set of labels, the labels do not directly correspond to the output revealed to a given observer. In this section we show how we can prove termination-insensitive *policy compliance*; data is revealed to an external observer only if it is allowed by the policy specified in the program. Thus if S_1 and S_2 are terminating programs that differ only in k-labeled components and the computed policy V_i for each program does not permit revealing k-sensitive data to the output channel, then the set of possible outputs from each program is identical. Here, an output $f : v$ combines both the output channel f and the value v, to ensure that sensitive information is not leaked either via the output value or by the choice of output channel.

Before we formally prove this property, we introduce the notion of *k-security*. A program S is *k-secure* if it terminates and its computed policy never permits revealing k-sensitive data, i.e.

$$\exists V, f, R \text{ such that } \emptyset, S \Downarrow V, f : R.$$

and

$$\begin{aligned} &\forall V, f, R.\ \text{ if } \emptyset, S \Downarrow V, f : R \\ &\text{then } \forall pc.\ pc(V) = \mathit{true} \Rightarrow \overline{k} \in pc \end{aligned}$$

Also, note that every label has a default policy of $\lambda x.\mathit{true}$. More formally:

$$\Sigma(k) = \lambda x.\mathit{true} \quad \forall k \notin \mathit{domain}(\Sigma)$$

Theorem 3. *Suppose for $i \in 1, 2$:*

$$S_i = print\ \{e\}\ C[\langle k\ ?\ e_i : e_l \rangle]$$

where each S_i is k-secure. Then
$\{\ f : R \mid \exists V.\ \emptyset, S_1 \Downarrow V, f : R\ \} = \{\ f : R \mid \exists V.\ \emptyset, S_2 \Downarrow V, f : R\ \}$.

Proof. We show left-to-right containment as follows. (The converse containment holds by a similar argument.) Let $e'_i = C[\langle k\ ?\ e_i : e_l \rangle]$. Suppose

$$\emptyset, S_1 \Downarrow V_{p1}, f : R$$

Then by the [F-PRINT] rule

$$\begin{aligned}
&\emptyset, e \Downarrow_{\emptyset} \Sigma_1, V_f \\
&\Sigma_1, e'_1 \Downarrow_{\emptyset} \Sigma_{21}, V_{c1} \\
&e_{p1} = \lambda x.true\ \wedge_f\ \Sigma_{21}(k_1)\ \wedge_f\ ...\ \wedge_f\ \Sigma_{21}(k_n) \\
&\{\ k_1\ ...\ k_n\ \} = closeK(labels(e)\ \cup\ labels(e'_1), \Sigma_{21}) \\
&\Sigma_{21}, e_{p1}\ V_f \Downarrow_{\emptyset} \Sigma_{31}, V_{p1} \\
&pc(V_f) = f,\ pc(V_{c1}) = R,\ pc(V_{p1}) = true.
\end{aligned}$$

Since S_1 is k-secure, we now have that $\bar{k} \in pc$.
Since S_2 terminates, there is also an [F-PRINT] run for S_2 that includes the antecedents

$$\begin{aligned}
&\emptyset, e \Downarrow_{\emptyset} \Sigma_1, V_f \\
&\Sigma_1, e'_2 \Downarrow_{\emptyset} \Sigma_{22}, V_{c2} \\
&e_{p2} = \lambda x.true\ \wedge_f\ \Sigma_{22}(k_1)\ \wedge_f\ ...\ \wedge_f\ \Sigma_{22}(k_n) \\
&\{\ k_1\ ...\ k_n\ \} = closeK(labels(e)\ \cup\ labels(e'_2), \Sigma_{22}) \\
&\Sigma_{22}, e_{p2}\ V_f \Downarrow_{\emptyset} \Sigma_{32}, V_{p2}
\end{aligned}$$

We assume that both rule instances have identical labels $\{k_1, ...k_n\}$. In general, of course, those labels may differ. For example, e_{p2} may include an additional conjunct $\Sigma_{22}(k')$ not $\{\bar{k}\}$- equivalent to a corresponding conjunct in e_{p2}, but in this case we

can add a semantically transparent corresponding conjunct $\lambda x.true$ to recover the equivalence $e_{p1} \sim_{\{\bar{k}\}} e_{p2}$.
Now $e'_1 \sim_{\{\bar{k}\}} e'_2$.
So by Theorem 5, $\Sigma_{21} \sim_{\{\bar{k}\}} \Sigma_{22}$, $V_{c1} \sim_{\{\bar{k}\}} V_{c2}$.

Also $e_{p1} \sim_{\{\bar{k}\}} e_{p2}$, so

$$\Sigma_{31} \sim_{\{\bar{k}\}} \Sigma_{32}$$
$$V_{p1} \sim_{\{\bar{k}\}} V_{p2}$$

We now continue the [F-PRINT] run on S_2 by choosing the same pc as from the run on S_1.
Clearly $pc(V_f) = f$.
Moreover, since $\bar{k} \in pc, pc(V_{c2}) = pc(V_{c1}) = R$.
Similarly, $pc(V_{p2}) = pc(V_{p1}) = true$.
Hence we can conclude $\emptyset, S_2 \Downarrow V_{p2}, f : R$ as required. □

Chapter 6

Faceted Execution for Database-Backed Applications

Interactions with databases 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 [104]. Indeed, the patch for the recent HotCRP bug involves policy checks across application code and database queries.

With Jacqueline [105], we demonstrate how to reduce the opportunity for error by automatically enforcing information flow policies in database-backed web applications. The core of our work on Jacqueline is a policy-agnostic object-relational mapping (ORM) framework. Standard ORM frameworks abstract over interactions with an underlying database to provide a uniform data object representation. The Jacqueline ORM framework additionally provides a uniform representation of sensitive values and policies in order to support policy-agnostic programming. 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.

In this chapter, we extend faceted execution to interoperate with relational op-

erators, describe how it maps onto an implementation strategy that works with unmodified relational databases, and prove that this extends the guarantees. Towards making policy-agnostic programming scale, we also present and formalize an *early pruning* optimization that reduces the scope of faceted execution in web applications.

6.1 Solution Overview

Austin *et al.*'s faceted semantics for Jeeves [10] provide strong guarantees, but they have the following problems for web applications. First, the guarantees only hold for programs that run entirely within a faceted Jeeves runtime, preventing Jeeves programs from interoperating with commodity databases. Second, 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 6.2.

Django

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

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

Jacqueline

```sql
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 6.1: SQL code and example tables, with and without policies.

EventGuest.objects.**filter**(guest__name="Alice")

Django Query

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

Jacqueline Query

```sql
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 6.2: Translated ORM queries in Django vs. Jacqueline.

6.1.1 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. A database not designed to work with policy-agnostic programming may be transformed into one that is simply by adding meta-data columns.

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 6.1, 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"⟩
```

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 following query on the rows from Figure 6.1: **SELECT** * **from** Event **WHERE** location = "Schloss Dagstuhl". 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 6.2, 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.

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

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

values ⟨a ? "Charlie" : "∗∗∗"⟩, ⟨b ? "Bob" : "∗∗∗"⟩, and ⟨c ? "Alice" : "∗∗∗"⟩. 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	f	jid	jvars
"Charlie"	0	"a=True"	"∗∗∗"	0	"a=False"
"∗∗∗"	0	"a=False"	"∗∗∗"	1	"b=False"
"Bob"	1	"b=True"	"∗∗∗"	2	"c=False"
"∗∗∗"	1	"b=False"	"Alice"	2	"c=True"
"Alice"	2	"c=True"	"Bob"	1	"b=True"
"∗∗∗"	2	"c=False"	"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 $\{a, \overline{b}, c\}$ would see ["∗∗∗", "Alice", "Charlie"].

While the Jacqueline ORM can use SQL queries for selects, joins, and sorts, there is no equivalent aggregate function, for instance **COUNT** or **SUM**. Using aggregate queries in the database could leak information. While selects, joins, and sorts preserve non-interference by preserving the representation of faceted values, aggregate queries would combine values across facets. For instance, using a SQL query to sum across all rows of some description would sum across the secret *and* public facets. For this reason, Jacqueline performs these operations in memory using the Jeeves runtime.

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.

6.1.2 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, 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. Second, the most common information-leaking computation sinks involve

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.

6.2 Formal Semantics and Policy Compliance

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

6.2.1 Syntax and Formal Semantics

The language λ^{JDB} extends the language λ^{jeeves} [10] with support for databases, which we model as relational tables. Figure 6-1 summarizes the λ^{JDB} syntax, with the constructs from λ^{jeeves} marked in gray. The λ^{jeeves} language, in turn, extends the standard imperative λ-calculus with constructs for declaring new labels (label k in e), for imperatively attaching policies to labels (restrict(k, e)), and for creating faceted values ($\langle k \; ? \; e_H : e_L \rangle$). This last expression behaves like e_H from the perspective of any principal authorized to see data with label k. For all other principals, the faceted expression behaves exactly like e_L.

The language λ^{JDB} extends λ^{jeeves} with support for databases, which we model as relational tables, where each table is a (possibly empty) sequence of rows and each

$e ::=$	*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 6-1: λ^{JDB} syntax.

Runtime Syntax

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

Contexts Rules

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

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

Figure 6-2: Runtime syntax and contexts rules for faceted evaluation of λ^{JDB}.

row is a sequence of strings. We require that all rows in a table have the same size. To manipulate tables, λ^{JDB} includes the usual operators of the relational calculus: *selection* ($\sigma_{i=j}\ e$), which selects the rows in a table where fields i and j are identical, *projection* ($\pi_{\bar{i}}\ e$), 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" $\forall \overline{A}, B.(B \to \overline{A} \to B) \to B \to \text{table } \overline{A} \to B$.

6.2.2 Formal Semantics

We formalize the big-step semantics as the relation $\Sigma, e \Downarrow_{pc} \Sigma', V$, denoting that expression e and store Σ evaluate to V, producing a new store Σ'. The program

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

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

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

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

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

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

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

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

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

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

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

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

Figure 6-3: Rules for evaluation λ_J subset of λ^{JDB}.

Evaluation with Relational Operations

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

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

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

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

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

$$\frac{}{\Sigma, \text{fold } V_f \; V_p \; (\text{table } \epsilon) \Downarrow_{pc} \Sigma, V_p} \quad [\textsc{f-fold-empty}]$$

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

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

Figure 6-4: Rules for evaluation with relational operations.

counter pc is a set of *branches*. Each branch is either a label k or a negated label $\overline{k}$. Association with k means the computation is visible only to principals authorized to see k. Association with $\overline{k}$ means the computation is visible only to principals *not* authorized to see k.

We could represent faceted tables as $\langle k \mathbin{?} \text{table } T_1 : \text{table } T_2 \rangle$, but this approach would incur significant space overhead, as it requires storing two copies of possibly large database tables, possibly with only small differences between the two tables. Instead, we use the more efficient approach of *faceted rows*, where each row $(B, \overline{s})$ in the database includes a set of branches B describing who can see that row. For example, the expression $\langle k \mathbin{?} \text{row "Alice" "Smith"} : \text{row "Bob" "Jones"} \rangle$ evaluates to the following table [2]:

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

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 $\langle\!\langle \cdot \mathbin{?} \cdot : \cdot \rangle\!\rangle$ to create either a new faceted value or a table with internal branches on rows:

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

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

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

T_H and T_L creates a new table T containing the rows from T_H and T_L, annotated with k and $\overline{k}$ respectively. We extend this operator to sets of branches:

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

We show the runtime syntax for faceted evaluation rules in Figure 6-2, the evaluation expression rules for the λ_{J} subset of λ^{JDB} in Figure 6-3, and the evaluation rules for relational operations in Figure 6-4. 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 $\overline{k}$ 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 $\overline{k}$.

Our rules use contexts (Figure 6-2) 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 fold are more interesting. If a row $(B, \overline{s})$ 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 $\overline{s}$ and the accumulator V', producing a new accumulator V''. The result of that fold step is $\langle\!\langle B ? V'' : V' \rangle\!\rangle$, a faceted expression that appears like V'' to principals that can see the B-labeled row and like V' to other principals.

The faceted execution semantics describe the propagation of labels and facets for the purpose of complying with policies at computation sinks. λ^{JDB} expressions do not perform I/O, while λ^{JDB} statements include the effectful construct print $\{e_v\}$ e_r that prints expression e_r under the policies and viewing context e_v. The λ^{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 λ^{jeeves} rules for declaring new labels and attaching policies to labels are in Appendix B.1.

6.2.3 End-to-End Policy Compliance

Austin *et al.* have proven policy compliance guarantees for λ^{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 λ^{JDB}, yielding guarantees of end-to-end policy compliance for database-backed applications.

The proof of policy compliance involves extending the *projection* property of λ^{jeeves}. A key property of λ^{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 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

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

$$\forall \bar{k} \in B . k \notin L$$

We extend views to values:

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

We extend views to expressions:

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

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

We then prove the Projection Theorem. The full proof is in Appendix B.5. Proofs of the key lemmas are in Appendices B.2 and B.3.

Theorem 4 (Projection)**.** *Suppose* $\Sigma, e \Downarrow_{pc} \Sigma', V$*. Then for any view* L *for which* pc *is visible,*

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

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

Theorem 5 (Termination-Insensitive Non-Interference)**.**

Let L be any view. Suppose $\Sigma_1 \sim_L \Sigma_2$ and $e_1 \sim_L e_2$, and that:

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

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

The Termination-Insensitive Non-Interference Theorem allows us to extend the termination-insensitive *policy compliance* theorem of λ^{jeeves} [10]: data is revealed to an external observer only if it is allowed by the policy specified in the program.

6.2.4 Early Pruning

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

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

We prove the Projection Theorem holds with this extension.

Part III

Executing Policy-Agnostic Programs

Chapter 7

Implementing a Policy-Agnostic Web Framework

An important part of demonstrating the feasibility of policy-agnostic programming involves building a realistic prototype that we can use to demonstrate advantages over implementations with manually implemented information flow policies. In this chapter, we describe the implementation of Jeeves as an embedding in Python, a dynamically typed language that is typically interpreted. We also describe the implementation of Jacqueline as an extension of Django, a popular Python web framework. Finally, we discuss the tradeoffs of using Python, as opposed to Scala.

7.1 Python Embedding of the Jeeves Runtime

We embedded Jeeves in a subset of Python, as a library[1]. Python's flexibility facilitates embedding through overloading and dynamic source transformation. We implemented Jeeves as an embedding that allows the programmer to use a subset of Python with policy-agnostic programming constructs. Programmers can write Jeeves programs simply by importing our library and annotating classes and functions with the `@jeeves` decorator. The library exports functions for creating labels, creating sensitive values, attaching policies, and producing non-faceted values based

[1] The code is publicly available at `https://github.com/jeanqasaur/jeeves`.

on policies. Our implementation supports a subset of Python's syntax that includes if-statements, for-loops, and return statements.

7.1.1 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.

7.1.2 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 [72] to find a satisfying assignment.

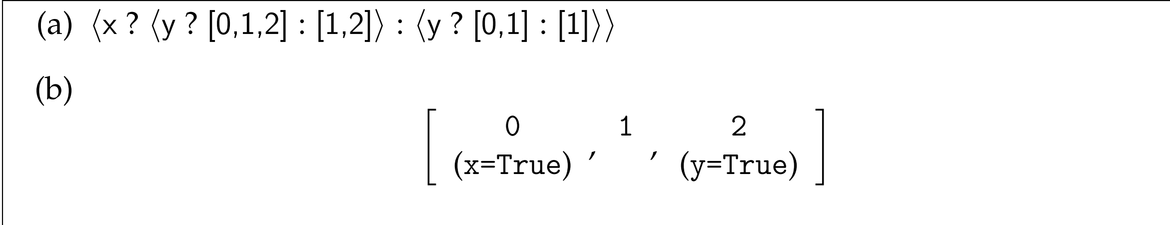


Figure 7-1: Representing faceted lists: (a) is the naive representation and (b) is the optimized representation.

7.1.3 Garbage-Collecting Labels and Policies

We designed our environments to support long-running programs. For a given expression, Jeeves policies have the property that only policies associated with labels involved in transitive closure of the expression and associated policies may affect the concretization. This allows us to garbage-collect irrelevant policies. The policy environment is a map between labels and policies that stores weak references to labels, allowing the Python runtime to garbage-collect labels and policies once the labels are no longer in use. The implementation uses a new solver instance for each concretization because Z3 does not yet support garbage collection.

7.1.4 Representing lists

When data values can encapsulate multiple views, it is possible to significantly improve performance if algorithms over data structures take the multiple views into account [103]. The naive representation of a faceted list is as a facet tree (Figure 7-1(a)), but this is exponential in the number of labels. We can do better by leveraging common subsequences to represent it as a list of values and label conditions (Figure 7-1(b)). Concretizing the list returns the subsequence of values that correspond to conditions that evaluate to True. In addition to being more compact for the general case, this representation enables us to perform list comprehensions in linear time.

7.1.5 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.

7.1.6 Decision to Use Python

We considered implementing Jacqueline in Scala, especially since there existed a Scala implementation of Jeeves [10,106]. We chose to implement Jacqueline in Python for three reasons: 1) the popularity of dynamically typed for web programming, 2) the ease of use of dynamically typed languages for prototyping, and 3) the flexibility of Python as a target for embedding a domain-specific language.

A major reason we switched from Scala to Python was our desire to demonstrate feasibility in the web application domain. Dynamically typed languages are increasingly popular for web programming. Of the candidate dynamic languages, we chose Python for the target source language because of its popularity: according to the TIOBE index [98], Python is the fifth most popular programming language as of August 2015, behind Java, C, C++, and C#.

Another reason we switched was because adoption was important to us and we hypothesized that users of dynamically typed languages are more likely to adopt policy-agnostic programming. Policy-agnostic programming provides runtime support for automating the enforcement of information flow policies. With this approach, the programmer trades ease of programming for runtime overhead: the programmer no longer needs to explicitly implement policies as repeated checks and filters, with the expectation that the runtime will do more work to customize program behavior. This strategy is a better fit for the rapid development that programmers like to do in dynamically typed languages [110].

We found Python to be a more flexible language for embedding an experimental domain-specific language. Our Scala code was verbose, as Scala requires us to explic-

itly annotate the different evaluation cases for different types. We also spent a fair amount of the implementation time providing type-checking and type-conversion hints to the type checker. The trade-off is that Scala's static types provide more guarantees, making it more crucial to thoroughly test the Python implementation. Both Python and Scala allow for most, but not all, operators to be overloaded. With Scala, a work-around for supporting a fully sugared embedded language is to use compiler extensions like Scala-Virtualized [70] 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 [5] for implementing macros provides support. Python's dynamic typing also made it easy to "monkey-patch" the Django web framework by overloading Django functions with replacement Jeeves functions. Building our own ORM would have been substantially more labor-intensive in Scala because of the types.

7.2 Limitations

We embed into a subset of Python that corresponds with our extension of the imperative lambda calculus. While incorporating more Python constructs is mostly a matter of engineering, constructs such as **eval** and those used for reflection cannot be supported using our embedding strategy. This is because this implementation embeds Jeeves on top of the Python interpreter and thus relies on runtime invariants that are at risk of being violated by malicious users. For instance, we assume that only the runtime can examinine the facets of sensitive values. For this reason, the guarantees hold only if the program remains within the permitted subset of Python. This should not be a problem, as 1) it is possible to statically verify that the programmer uses the permitted subset and 2) we assume the programmer is cooperative rather than adversarial. Note that we do not have these limitations if, rather than embedding faceted execution into a non-faceted language, we used an interpreter that supports faceted execution.

Chapter 8

Jacqueline in Practice

In this chapter, we demonstrate the practical feasibility of policy-agnostic in terms of both expressiveness and performance. Using 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 the Workshop on Programming Languages Technology for Massive Open Online Courses (PLOOC) 2014.

- **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.

8.1 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) [11,79]. 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.

8.2 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)

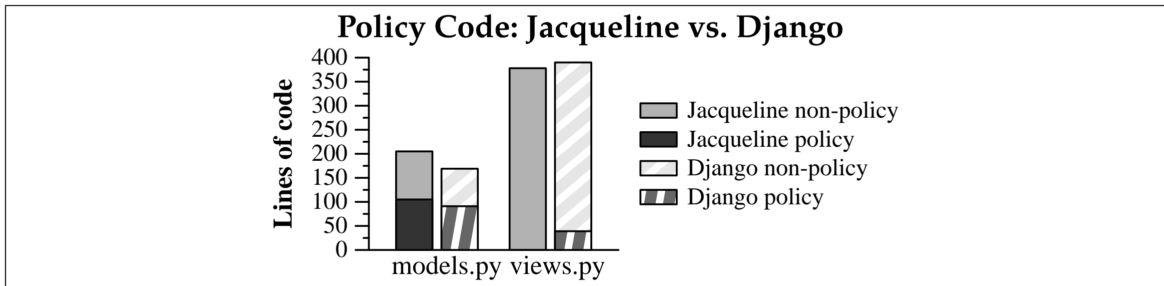


Figure 8-1: Distribution of policy code with Jacqueline and Django conference management systems.

separating policies and other functionality decreases policy code size.

8.2.1 Django Conference Management System

We compare the lines of code in the Jacqueline and Django conference management systems in Figure 8-1. 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%.

There are two other code-related arguments for why Jacqueline provides advantages over Django. The first has to do with policy spaghetti. The policies are

concentrated in the data schema file models.py for Jacqueline, whereas the Django policies are additionally distributed across the code. In our small conference management system, there are at least ten distinct locations where the program implements a policy check in the controller file *views.py*. This brings us to the second argument, which has to do with ease of reasoning and maintainability. In Jacqueline, the programmer can trust that policies will be implemented without needing to reason about other functionality. In Django, the programmer needs to understand the code in order to implement policies and understand the policies in order to implement new functionality. Otherwise, a single misplaced check can leak information.

8.2.2 HotCRP

Policies and functionality are intertwined across the HotCRP conference management system [53], 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].

8.3 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 non-trivial 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

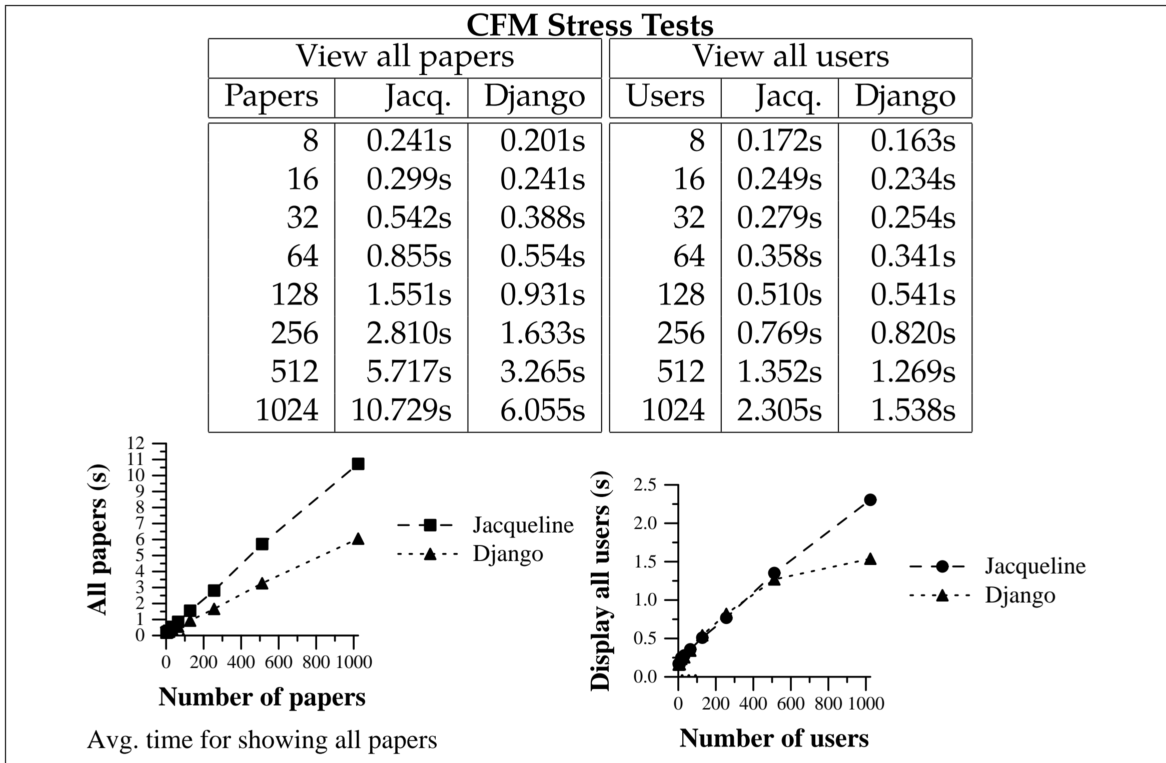
CFM Representative Actions					
View single paper			View single user		
Papers	Jacq.	Django	Users	Jacq.	Django
8	0.160s	0.177s	8	0.164s	0.158s
16	0.165s	0.175s	16	0.164s	0.159s
32	0.160s	0.177s	32	0.164s	0.159s
64	0.159s	0.173s	64	0.164s	0.159s
128	0.160s	0.173s	128	0.167s	0.158s
256	0.159s	0.173s	256	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 8-2: Times to view profiles for a single paper and single user, in Jacqueline and Django.

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.

8.3.1 Representative Actions

We measured the time it takes for our system to 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 8-2. 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 performs better than the Django implementation because in a few places, the implementation needs iterate over collections of data rows a second time in order to apply policy checks. In the Jacqueline implementation, the programmer can simply rely on the ORM to attach the policies. Note that roundtrips to the database dominate the Django baseline performance. This is a known performance bottleneck with object-relational mappings that Cheung *et al.* [27] address.



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 8-3: Times to view list of summary information for all papers and all users, in Jacqueline and Django.

8.3.2 Stress Tests

In Figure 8-3 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 resolves different policies for each paper and user field. The graphs demonstrate that with both Jacqueline and Django, the time to load data scales linearly with respect to the underlying algorithms. In these results, Jacqueline has a 1.75x overhead for showing all papers that comes from fetching both versions of data from the database before resolving the policies. Integrating policies more deeply with the database could reduce this overhead. There is no solver overhead, as there are no mutual dependencies between sensitive values and policies.

Results for the other case studies show similar promise for Jacqueline's ability to scale. In Figure 8-4 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

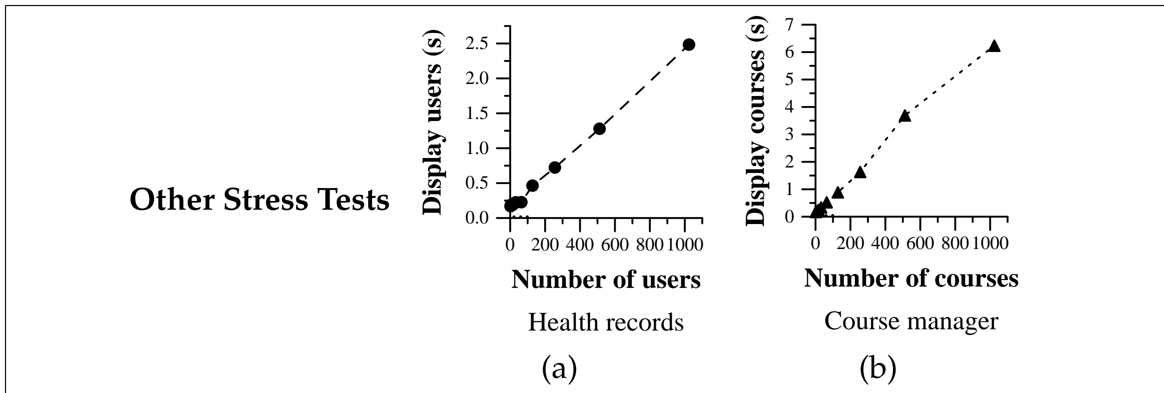


Figure 8-4: Jacqueline stress tests for other case studies.

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 8-5: The course manager stress test performs well with the Early Pruning optimization and times out otherwise.

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.

8.3.3 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 8-5 how for just 8 randomly generated courses and instructors, the system begins to hit memory limits. Early Pruning makes it possible to execute such programs. As long as the computation to determine a viewer is simple, Early Pruning can simplify other computations after the viewer is known.

Chapter 9

Conclusions

As users share more sensitive data and as more code computes using this data, information leaks are becoming more prevalent. A standard way of protecting sensitive data is by implementing checks and filters across the program. When programs become more complex, it is increasingly difficult to reason about the interaction of these checks with each other and with the rest of the program. There are many opportunities for the programmer to inadvertently leak information.

This thesis addresses the problem of information leaks resulting from programmer error. The work focuses on *information flow*, the flow of sensitive values through computations. The fundamental idea behind the approach is a new programming model called *policy-agnostic programming* that allows the programmer to factor out information flow policies from the rest of the program. Using this approach, the programmer is trusted only with correctly specifying the policies, rather than correctly implementing the policies across the program. Program correctness with respect to information flow policies now relies on the correct specification of the policies and the correctness of the language runtime implementation. We show that this reduces the amount of trusted code with reasonable performance tradeoffs.

9.1 Summary of Contributions

We present Jacqueline, a policy-agnostic web framework that automatically enforces information flow policies. Jacqueline allows the programmer to implement information flow policies once, alongside data schemas, rather than as repeated checks and filters across the program. Rather than simply preventing programs from leaking information, Jacqueline customizes program behavior based on the policies. The runtime executes alternate behaviors when flows are disallowed, instead of raising an error or producing a default value. The result is that the programmer can factor information flow policies out of the rest of the program. Jacqueline manages policy interactions and policy resolution so that the programmer does not need to.

Underlying Jacqueline is Jeeves [10, 106], a kernel language for policy-agnostic programming. Jeeves allows the programmer to associate sensitive values with policies and write the remainder of the program in a policy-agnostic manner. In Jeeves, sensitive values encapsulate multiple views. Policies describe rules under which each view is visible to a given viewer. The rest of the program needs to account for the fact that different views, perhaps corresponding to different granularities of sensitive values, are flowing through computations, but the program does not need to be aware of the policies guarding the views.

This thesis presents the design, semantics, and implementation of Jeeves and Jacqueline. We show how the Jeeves's runtime semantics and Jacqueline's object-relational mapping framework enforce policy compliance by construction. Through application case studies built using Jacqueline, we demonstrate that policy-agnostic is practically feasible. The key contributions are as follows:

- **Design of Jeeves, a language for policy-agnostic programming.** Factoring out information flow from the rest of the program is challenging because information flow is so intertwined with the rest of the program. We present a language, Jeeves, that allows the programmer to specify multiple views of sensitive values and associate policies specifying when secret views are accessible to a given viewer. In Jeeves, policies define a set of declarative constraints

over label values guarding sensitive values. Scalability is often a challenge with declarative languages because the declarative constraints define large, often infinite, spaces of possible behaviors. Jeeves's constraints allow the runtime to manage policy dependencies, including mutual dependencies between sensitive values and policy computations, while exploring a finite space. We present a formalization of Jeeves and proofs of termination-insensitive non-interference and policy compliance.

- **Design and implementation of Jacqueline, a web framework for policy-agnostic programming.** There is a gap between the guarantees that Jeeves provides and what we need to use policy-agnostic programming for database-backed web applications. It is important for Jeeves to interoperate with databases for web applications, but interacting with any external database can subvert the guarantees. To address this issue, we extend the policy-agnostic model for database-backed applications. The key to the solution is to use an object-relational mapping framework to map the Jeeves execution model onto a relational database. We present a formalization of this mapping and provide proofs that it extends Jeeves's policy compliance guarantees. We present Jacqueline, a web framework based on these semantics that is implemented as an extension of the Django Python web framework and works with unmodified SQL databases.

- **Optimization strategies for making policy-agnostic programming practical.** 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. We observe that web frameworks do need to assume the viewer is unknown until output because it is common for web frameworks to track the viewing context. We present an optimization strategy for web applications that allows the runtime to prune alternate execution branches based on the viewing context. 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.

- **Demonstration of practical feasibility.** One reason programmers continue to use error-prone languages and tools is because they facilitate implements of scalable applications. For this reason, it is important to demonstrate that Jacqueline has reasonable overheads compared to manually implemented policies. We demonstrate the expressiveness and performance of Jacqueline through several application 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.

9.2 Conclusions

As programmers create more programs that compute using sensitive information, it becomes increasingly attractive to decouple security and privacy concerns from the other functionality. It has previously been difficult to separate these concerns because they are often deeply intertwined with the rest of the program. To help programmers manage sensitive data, we present an approach that factors out the specification information flow policies from the rest of the program and automates their implementation. In this thesis, we show that policy-agnostic programming provides strong theoretical guarantees, centralizes the policy code, and exhibits reasonable execution overheads in our web application case studies.

Towards making policy-agnostic programming more appealing to programmers, one direction of future work involves creating tools for helping programmers check they wrote the policies they intended to write. With the policy-agnostic approach, we do not need to verify the correctness of the implementation of information flow policies, as the runtime automatically enforces them. The question that remains is whether the policies that the programmer specified are the policies the programmer

intended. A standard way to verify the correctness of code is to check it against an additional specification—meta-policies—from the programmer. Because these policies are already automatically enforced, however, an interesting question is what additional specifications should look like. Because we are executing specifications, it may be useful for the programmer to express sanity-checking specifications in the form of test cases.

Towards making policy-agnostic programming more general-purpose, another direction of future research involves extending the policy language to handle more stateful policies. Information flow is often too strong a requirement for the properties that programmers want. For instance, information flow policies cannot permit an average salary to be revealed while protecting individual salaries. To support such policies, we can expand policy-agnostic programming to enforce policies based on aggregate values. One way to do this is to incorporate differential privacy, which allows an aggregate value to be revealed if the likelihood of inferring individual values is low. Another potential solution is to support policies on computation histories, for instance allowing values to be revealed if certain operators have been applied to derive the result. Runtime support for this would be a natural extension of Jeeves's dynamic enforcement approach.

Looking forward, there is no reason to limit the enforcement of information flow policies to security and privacy. Software is increasingly operating over data from many users, for the consumption of many users, where each user may have a customized experience with respect to the data and application. Customization may have to do with access permissions, but it may also has to do with all sorts of other preferences of the data producers and consumers. Using existing programming paradigms, managing these policies across the program becomes both a programmer bottleneck and a source of error. Policy-agnostic programming promises to remove this bottleneck and help programmers focus on the novel and semantically interesting parts of software applications.

Bibliography

[1] Toward provenance-based security for configuration languages. In *Presented as part of the 4th USENIX Workshop on the Theory and Practice of Provenance*, Berkeley, CA, 2012. USENIX.

[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] Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. A core calculus for provenance. *Journal of Computer Security*, 2014. In press.

[7] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Hippocratic databases. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 143–154. VLDB Endowment, 2002.

[8] Owen Arden, Michael D. George, Jed Liu, K. Vikram, Aslan Askarov, and Andrew C. Myers. Sharing mobile code securely with information flow control. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 191–205, Washington, DC, USA, 2012. IEEE Computer Society.

[9] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 165–178, New York, NY, USA, 2012. ACM.

[10] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. Faceted execution of policy-agnostic programs. *Proceedings of the Eighth ACM SIGPLAN workshop on Programming languages and analysis for security - PLAS '13*, page 15, 2013.

[11] Adam Barth, Anupam Datta, John C. Mitchell, and Helen Nissenbaum. Privacy and contextual integrity: Framework and applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 184–198, Washington, DC, USA, 2006. IEEE Computer Society.

[12] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. Secpal: Design and semantics of a decentralized authorization language. *J. Comput. Secur.*, 18(4), December 2010.

[13] Moritz Y. Becker and Sebastian Nanz. The role of abduction in declarative authorization policies. In Paul Hudak and David Scott Warren, editors, *PADL*, Lecture Notes in Computer Science, 2008.

[14] Moritz Y. Becker and Sebastian Nanz. A logic for state-modifying authorization policies. *ACM Trans. Inf. Syst. Secur.*, 13(3), 2010.

[15] Moritz Y. Becker, Alessandra Russo, and Nik Sultana. Foundations of logic-based trust management. In *Proceedings - IEEE Symposium on Security and Privacy*. IEEE, 2012.

[16] Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. Capabilities for information flow. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, PLAS '11, pages 5:1–5:15, New York, NY, USA, 2011. ACM.

[17] Aaron Blankstein and Michael J. Freedman. Automating isolation and least privilege in web services. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, Washington, DC, USA, 2014. IEEE Computer Society.

[18] Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. Programming with angelic nondeterminism. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, New York, NY, USA, 2010. ACM.

[19] Dan Boneh, Amit Sahai, and Brent Waters. Functional encryption: Definitions and challenges. In *Proceedings of the 8th Conference on Theory of Cryptography*, TCC'11. Springer-Verlag, 2011.

[20] N. Broberg and David Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *European Symposium on Programming (ESOP)*, 2006.

[21] Niklas Broberg and David Sands. Flow locks: Towards a core calculus for dynamic flow policies. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3924 LNCS(March):180–196, 2006.

[22] Niklas Broberg and David Sands. Paralocks: Role-based information flow control and beyond. 45(1), 2010.

[23] Pablo Buiras, Deian Stefan, and Alejandro Russo. On dynamic flow-sensitive floating-label systems. *CoRR*, abs/1507.06189, 2015.

[24] Roberto Capizzi, Antonio Longo, V. N. Venkatakrishnan, and A. Prasad Sistla. Preventing information leaks through shadow executions. In *Proceedings - Annual Computer Security Applications Conference, ACSAC*, pages 322–331, 2008.

[25] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate pro-

grams. In *PLDI: Programming Languages Design and Implementation*, volume 47, pages 169–180, 2012.

[26] James Cheney. A formal framework for provenance security. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, pages 281–293, Washington, DC, USA, 2011. IEEE Computer Society.

[27] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 931–942, New York, NY, USA, 2014. ACM.

[28] Adam Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 105–118. USENIX Association, 2010.

[29] S. Chong and A.C. Myers. Language-Based Information Erasure. *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 241–254.

[30] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM conference on Computer and communications security - CCS '04*, page 198, New York, New York, USA, 2004. ACM Press.

[31] Stephen Chong, K. Vikram, and Andrew C. Myers. SIF: Enforcing confidentiality and integrity in web applications. In *Proceedings of the 16th USENIX Security Symposium*, pages 1–16, August 2007.

[32] Brian J. Corcoran, Nikhil Swamy, and Michael Hicks. Cross-tier, label-based security enforcement for web applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 269–282, June 2009.

[33] B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, 2005.

[34] Dorothy E. Denning. A lattice model of secure information flow, May 1976.

[35] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[36] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Proceedings - IEEE Symposium on Security and Privacy*, pages 109–124, 2010.

[37] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *Proceedings of the Third International Joint Conference on Automated Reasoning*, IJCAR'06. Springer-Verlag, 2006.

[38] Jeffrey Fischer, Rupak Majumdar, and Shahram Esmaeilsabzali. Engage: a deployment management system. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 263–274. ACM, 2012.

[39] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. ICSE '05, New York, NY, USA, 2005. ACM.

[40] Robert W. Floyd. Nondeterministic algorithms. *J. ACM*, 14, October 1967.

[41] Nate Foster, Arjun Guha, Mark Reitblatt, Alec Story, Michael J. Freedman, Naga Praveen Katta, Christopher Monsanto, Joshua Reich, Jennifer Rexford, Cole Schlesinger, David Walker, and Rob Harrison. Languages for software-defined networks. *IEEE Communications Magazine*, 51(2), 2013.

[42] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic. *Proceeding of the 16th ACM SIGPLAN international conference on Functional programming - ICFP '11*, 46(9):279, 2011.

[43] Deepak Garg and Frank Pfenning. Stateful authorization logic - proof theory and a case study. *Journal of Computer Security*, 20(4), 2012.

[44] Daniel B. Giffin, Amit Levy, and Deian Stefan. Hails: Protecting data privacy in untrusted web applications. In *10th Symposium on ...*, pages 47–60, 2012.

[45] M. Hanus. Improving control of logic programs by using functional logic languages. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*. Springer LNCS 631, 1992.

[46] Michael Hanus. Curry - An Integrated Functional Logic Language. Technical Report 5, 2006.

[47] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. 1998.

[48] Martin Hirt and Ueli M. Maurer. Complete characterization of adversaries tolerable in secure multi-party computation (extended abstract). In James E. Burns and Hagit Attiya, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, Santa Barbara, California, USA, August 21-24, 1997*. ACM, 1997.

[49] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Symposium on Principles of Programming Languages (POPL)*, 2006.

[50] Information is Beautiful. Codebases: Millions of lines of code. `http://www.informationisbeautiful.net/visualizations/million-lines-of-code/`, August 2015.

[51] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach. *2011 IEEE Symposium on Security and Privacy*, pages 413–428, 2011.

[52] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. *ACM Computing Surveys*, 28(June):220–242, 1997.

[53] Eddie Kohler. HotCRP. http://www.read.seas.harvard.edu/ kohler/hotcrp/.

[54] Eddie Kohler. Github commit: Don't expose paper acceptance by allowing final copy submission. `https://github.com/kohler/hotcrp/commit/ba826918adfbece55ba1c04252c371e2c86a5ffb`, July 2009.

[55] Eddie Kohler. hotcrp / news. `https://github.com/kohler/hotcrp/blob/9f9382f711e6c719bc28c25433a5d228988ba314/NEWS`, June 2015.

[56] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, New York, NY, USA, 2007. ACM.

[57] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis, 2010.

[58] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4435 LNCS, pages 75–89, 2007.

[59] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovac, Raghu Ramakrishnan, Yirong Xu, and David DeWitt. Limiting disclosure in hippocratic databases. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 108–119. VLDB Endowment, 2004.

[60] Ninghui Li, J.C. Mitchell, and W.H. Winsborough. Design of a role-based trust-management framework. In *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, pages 114–130, 2002.

[61] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, PADL '03. Springer-Verlag, 2003.

[62] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. *ACM SIGPLAN Notices*, 40(1):158–170, January 2005.

[63] Jed Liu, MD George, K Vikram, and X Qi. Fabric: A platform for secure distributed computation and storage. *Proceedings of the ACM ...*, 2009.

[64] J. W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 3, 1999.

[65] Fan Long and Martin Rinard. Prophet: Automatic patch generation via learning from successful patches. Technical Report MIT-CSAIL-TR-2015-027, Massachusetts Institute of Technology, Cambridge, Massachusetts, July 2015.

[66] Fan Long and Martin Rinard. Staged condition repair with condition synthesis. In *SIGSOFT FSE*, 2015.

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

[68] Aleksandar Milicevic, Daniel Jackson, Milos Gligoric, and Darko Marinov. Model-based, event-driven programming paradigm for interactive web applications. In *SPLASH 2013*. ACM, 2013.

[69] Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, and Daniel Jackson. Unifying execution of imperative and declarative code. *2011 33rd International Conference on Software Engineering (ICSE)*, pages 511–520, 2011.

[70] Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, PEPM '12, New York, NY, USA, 2012. ACM.

[71] Carroll Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, July 1988.

[72] Leonardo De Moura and Nikolaj Björner. Z3: An efficient SMT solver. In *In Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[73] Andy Mück and Thomas Streicher. A tiny constraint functional logic language and its continuation semantics. In *ESOP '94: Proceedings of the 5th European Symposium on Programming*, pages 439–453, London, UK, 1994. Springer-Verlag.

[74] Andrew C Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '99*, pages 228–241. ACM Press, 1999.

[75] Prasad Naldurg and K. R. Raghavendra. SEAL: a logic programming framework for specifying and verifying access control models. In Ruth Breu, Jason Crampton, and Jorge Lobo, editors, *SACMAT 2011, 16th ACM Symposium on Access Control Models and Technologies, Innsbruck, Austria, June 15-17, 2011, Proceedings*. ACM, 2011.

[76] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *IEEE Symposium on Security and Privacy*, 2011.

[77] Joseph P. Near and Daniel Jackson. Rubicon: bounded verification of web applications. In *SIGSOFT FSE*, 2012.

[78] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical report, Citeseer, 2004.

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

[80] Lars E. Olson, Carl A. Gunter, and P. Madhusudan. A formal framework for reflective database access control policies. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, New York, NY, USA, 2008. ACM.

[81] Roly Perera, Umut A. Acar, James Cheney, and Paul Blain Levy. Functional programs that explain their work. ICFP '12, New York, NY, USA, 2012. ACM.

[82] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. Python: the full monty. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 217–232. ACM, 2013.

[83] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02)*, Portland, Oregon, 2002. Superseded by [84].

[84] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.

[85] Martin Rinard. Acceptability-oriented computing, 2003.

[86] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. Enhancing server availability and security through

failure-oblivious computing. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04. USENIX Association, 2004.

[87] Shariq Rizvi, Alberto Mendelzon, S. Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, New York, NY, USA, 2004. ACM.

[88] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings - IEEE Computer Security Foundations Symposium*, pages 186–199, 2010.

[89] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[90] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In *Proceedings of the 24th Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'05. Springer-Verlag, 2005.

[91] Hesam Samimi, Ei Darli Aung, and Todd Millstein. Falling back on executable specifications. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6183 LNCS, pages 552–576, 2010.

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

[93] David Schultz and Barbara Liskov. IFDB: Decentralized information flow control for databases. In *EuroSys*, 2013.

[94] Paritosh Shroff, Scott F. Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *Proceedings - IEEE Computer Security Foundations Symposium*, pages 203–217, 2007.

[95] Douglas Smith. A generative approach to aspect-oriented programming. *Generative Programming and Component Engineering*, pages 483–535, 2004.

[96] Douglas R Smith. Aspects as Invariants. In Olivier Danvy, Fritz Henglein, Harry Mairson, and Alberto Pettorossi, editors, *Automatic Program Development*, pages 270–286. Springer Netherlands, 2008.

[97] Geoffrey Smith. Principles of secure information flow analysis. In *Malware Detection*, pages 291–307. 2007.

[98] TIOBE Software. Tiobe index for august 2015. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, August 2015.

[99] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing Policy Updates in Security-Typed Languages. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 202–216. Ieee.

[100] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types, 2011.

[101] Michael Carl Tschantz and Shriram Krishnamurthi. Towards reasonability properties for access-control policy languages. In *Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies*, SACMAT '06, New York, NY, USA, 2006. ACM.

[102] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. a Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, pages 1–20, 2009.

[103] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. Variational data structures: Exploring tradeoffs in computing with variability. In *Onward!*, 2014.

[104] Ashford 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].

[105] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. End-to-end policy-agnostic security for database-backed applications. *arXiv.org*, cs.PL, 2015.

[106] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies, 2012.

[107] Andrew C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82. IEEE Computer Society, 1982.

[108] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. *ACM Symposium on Operating Systems Principles*, page 13, 2009.

[109] Steve Zdancewic. A type system for robust declassification. In *Proceedings of the Nineteenth Conference on the Mathematical Foundations of Programming Semantics. Electronic Notes in Theoretical Computer Science*, pages 1–16. Citeseer, 2003.

[110] Haiping Zhao. Hiphop for PHP: Move fast. `http://developers.facebook.com/blog/post/358/`, February 2010.

[111] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *Int. J. Inf. Sec.*, 6(2-3):67–84, 2007.

Appendix A

Proofs for Faceted Execution of λ^{jeeves}

A.1 Proof of Projection

Theorem 1. *Suppose*

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

Then for any $q \in PC$ where pc and q are consistent

$$q(\Sigma), q(e) \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$$

Proof. We prove a stronger inductive hypothesis, namely that for any $q \in PC$ where $\neg \exists k.(k \in pc \wedge \overline{k} \in q) \vee (\overline{k} \in pc \wedge k \in q)$

1. If $\Sigma, e \Downarrow_{pc} \Sigma', V$ then $q(\Sigma), q(e) \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$.

2. If $\Sigma, (V_1\ V_2) \Downarrow_{pc}^{\mathsf{app}} \Sigma', V$ then $q(\Sigma), (q(V_1)\ q(V_2)) \Downarrow_{pc \setminus q}^{\mathsf{app}} q(\Sigma'), q(V)$.

The proof is by induction on the derivation of $\Sigma, e \Downarrow_{pc} \Sigma', V$ and the derivation of $\Sigma, (V_1\ V_2) \Downarrow_{pc}^{\mathsf{app}} \Sigma', V$, and by case analysis on the final rule used in that derivation.

- For case [F-LABEL], $e = \text{label } k \text{ in } e'$.

By the antecedents of this rule:

$$k' \text{ fresh}$$
$$\Sigma[k' := \lambda x.true], e'[k := k'] \Downarrow_{pc} \Sigma', V$$

By induction

$$q(\Sigma[k' := \lambda x.true]), q(e'[k := k']) \Downarrow_{pc\setminus q} q(\Sigma'), q(V)$$

Since $k' \notin \Sigma$, we know that $k' \notin q(\Sigma)$.

Therefore, $q(\Sigma)[k' := \lambda x.true] = q(\Sigma[k' := \lambda x.true])$.

By α-renaming, we assume $k \notin q$, $\overline{k} \notin q$, $k' \notin q$, and $\overline{k}' \notin q$.

Therefore $q(e')[k := k'] = q(e'[k := k'])$.

- For case [F-RESTRICT], $e = \text{restrict}(k, e')$. By the antecedents of this rule:

$$\Sigma, e' \Downarrow_{pc} \Sigma_1, V$$
$$\Sigma' = \Sigma_1[k := \Sigma_1(k) \;\wedge_f\; \langle\!\langle\, pc \cup \{k\} \;?\; V : \lambda x.true \,\rangle\!\rangle]$$

By induction, $q(\Sigma), q(e') \Downarrow_{pc\setminus q} q(\Sigma_1), q(V)$.

$$\begin{aligned}
q(\Sigma') &= q(\Sigma_1[k := \Sigma_1(k) \wedge_f \langle\!\langle\, pc \cup \{k\} \;?\; V : \lambda x.true \,\rangle\!\rangle]) \\
&= q(\Sigma_1)[k := q(\Sigma_1(k)) \wedge_f \\
&\qquad\qquad q(\langle\!\langle\, pc \cup \{k\} \;?\; V : \lambda x.true \,\rangle\!\rangle)] \\
&= q(\Sigma_1)[k := q(\Sigma_1(k)) \wedge_f \\
&\qquad\qquad \langle\!\langle\, pc \cup \{k\} \setminus q \;?\; q(V) : \lambda x.true \,\rangle\!\rangle)] \\
&\quad \text{by Lemma 1}
\end{aligned}$$

- For case [F-VAL], $e = V$.

Since $\Sigma, V \Downarrow_{pc} \Sigma, V$ and $q(\Sigma), q(V) \Downarrow_{pc\setminus q} q(\Sigma), q(V)$, this case holds.

- For case [F-REF], $e = \text{ref } e'$. Then by the antecedents of the [F-REF] rule:

$$\Sigma, e' \Downarrow_{pc} \Sigma'', V'$$
$$a \notin dom(\Sigma'')$$
$$V'' = \langle\!\langle\, pc \;?\; V' : 0 \,\rangle\!\rangle$$
$$\Sigma' = \Sigma''[a := V'']$$
$$V = a$$

  By induction, $q(\Sigma), q(e') \Downarrow_{pc \setminus q} q(\Sigma''), q(V')$.

  Since $a \notin dom(\Sigma'')$, $a \notin dom(q(\Sigma''))$.

  By Lemma 1, $q(V'') = \langle\!\langle\, pc \setminus q \;?\; q(V') : q(0) \,\rangle\!\rangle$.

  Since $\Sigma' = \Sigma''[a := V'']$, $q(\Sigma') = q(\Sigma'')[a := q(V'')]$.

  Therefore $q(\Sigma), \text{ref } q(e') \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$.

- For case [F-DEREF], $e = !e'$. Then by the antecedents of the [F-DEREF] rule:

$$\Sigma, e' \Downarrow_{pc} \Sigma', V'$$
$$V = deref(\Sigma', V', pc)$$

  By induction, $q(\Sigma), q(e') \Downarrow_{pc \setminus q} q(\Sigma'), q(V')$.

  By Lemma 2, $q(V) = deref(q(\Sigma'), q(V'), pc \setminus q)$.

  Therefore $q(\Sigma), q(!e') \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$.

- For case [F-ASSIGN], $e = (e_a := e_b)$.

  By the antecedents of the [F-ASSIGN] rule:

$$\Sigma, e_a \Downarrow_{pc} \Sigma_1, V_1$$
$$\Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V$$
$$\Sigma' = assign(\Sigma_2, pc, V_1, V)$$

  By induction

$$q(\Sigma), q(e_a) \Downarrow_{pc \setminus q} q(\Sigma_1), q(V_1)$$
$$q(\Sigma_1), q(e_b) \Downarrow_{pc \setminus q} q(\Sigma_2), q(V)$$

By Lemma 3, $q(\Sigma') = assign(q(\Sigma_2), pc \setminus q, q(V_1), q(V))$. Therefore $q(\Sigma), q(e_a{:=}\,e_b) \Downarrow_{pc\setminus q}$ $q(\Sigma'), q(V)$.

- For case [F-APP], $e = (e_a\ e_b)$. By the antecedents of the [F-APP] rule:

$$\Sigma, e_a \Downarrow_{pc} \Sigma_1, V_1$$
$$\Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V_2$$
$$\Sigma_2, (V_1\ V_2) \Downarrow_{pc}^{\mathsf{app}} \Sigma', V$$

By induction

$$q(\Sigma), q(e_a) \Downarrow_{pc\setminus q} q(\Sigma_1), q(V_1)$$
$$q(\Sigma_1), q(e_b) \Downarrow_{pc\setminus q} q(\Sigma_2), q(V_2)$$
$$q(\Sigma_2), (q(V_1)\ q(V_2)) \Downarrow_{pc\setminus q}^{\mathsf{app}} q(\Sigma'), q(V)$$

Therefore $q(\Sigma), q(e_a\ e_b) \Downarrow_{pc\setminus q} q(\Sigma'), q(V)$.

- For case [F-LEFT], $e = \langle k\ ?\ e_a : e_b\rangle$. By the antecedents of this rule

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

  - If $k \in q$, then $q(\langle k\ ?\ e_a : e_b\rangle) = q(e_a)$.
    By induction $q(\Sigma), q(e_a) \Downarrow_{pc\setminus q} q(\Sigma'), q(V)$.

  - Otherwise $k \notin q$ and $\bar{k} \notin q$.
    Therefore $q(\langle k\ ?\ e_a : e_b\rangle) = \langle k\ ?\ q(e_a) : q(e_b)\rangle$.
    Since $k \in pc \setminus q$, it holds by induction that

$$q(\Sigma), \langle k\ ?\ q(e_a) : q(e_b)\rangle \Downarrow_{pc\setminus q} q(\Sigma'), q(V)$$

- Case [F-RIGHT] holds by a similar argument as [F-LEFT].

- For case [F-SPLIT], $e = \langle k \mathbin{?} e_a : e_b \rangle$. By the antecedents of the [F-SPLIT] rule:

$$\Sigma, e_a \Downarrow_{pc \cup \{k\}} \Sigma_1, V_1$$
$$\Sigma_1, e_b \Downarrow_{pc \cup \{\bar{k}\}} \Sigma', V_2$$
$$V = \langle k \mathbin{?} V_1 : V_2 \rangle$$

  - Suppose $k \in q$. Then $q(e) = q(e_a)$ and $q(V_1) = q(V)$.
    By induction, $q(\Sigma), q(e_a) \Downarrow_{pc \cup \{k\} \setminus q} q(\Sigma_1), q(V_1)$.
    Lemma 4 implies $q(\Sigma_1) = q(\Sigma')$, so this case holds.
  - If $\bar{k} \in q$, Then $q(e) = q(e_b)$ and $q(V_2) = q(V)$.
    By Lemma 4 we know that $q(\Sigma) = q(\Sigma_1)$.
    By induction, $q(\Sigma_1), q(e_b) \Downarrow_{pc \cup \{k\} \setminus q} q(\Sigma'), q(V_2)$.
  - If $k \notin q$ and $\bar{k} \notin q$, then by induction

$$q(\Sigma), q(e_a) \Downarrow_{pc \cup \{k\} \setminus q} q(\Sigma_1), q(V_1)$$
$$q(\Sigma_1), q(e_b) \Downarrow_{pc \cup \{\bar{k}\} \setminus q} q(\Sigma'), q(V_2)$$

    By Lemma 1, $q(V) = \langle\!\langle\, pc \setminus q \mathbin{?} q(V_1) : q(V_2) \,\rangle\!\rangle$.

- For case [FA-FUN], $V_1 = \lambda x.e'$. By the antecedent of this rule

$$\Sigma, e'[x := V_2] \Downarrow_{pc} \Sigma', V$$

  We know that $q(\lambda x.e' \; V_2) = q(e'[x := V_2])$.
  By induction $q(\Sigma), q(e'[x := V_2]) \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$.

- Both cases [FA-LEFT] and [FA-RIGHT] hold by a similar argument as [F-LEFT].

- Case [FA-SPLIT] holds by a similar argument as [F-SPLIT].

□

Appendix B

Proofs for λ^{JDB}

B.1 Rules from λ^{jeeves}

These rules from λ^{jeeves} [9] describe how to declare labels and attach policies to labels. The rule [F-LABEL] dynamically allocates a label (label k in e), adding a fresh label to the store with the default policy of $\lambda x.true$. Any occurrences of k in e are α-renamed to k' and the expression is evaluated with the updated store. Policies may be further refined (restrict(k, e)) by the rule [F-RESTRICT], which evaluates e to a policy V that should be either a lambda or a faceted value comprised of lambdas. The additional policy check is restricted by pc, so that policy checks cannot themselves leak data. It is then joined with the existing policy for k, ensuring that policies can only become

more restrictive.

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

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

B.2 Proof of Lemma 5

Lemma 5 (A)**.**

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

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

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

- If $x = L(\langle k \;?\; F_1 : F_2 \rangle)$ for some non-table values F_1 and F_2, then this case holds since

- $x = L(F_1)$ if $k \in L$.
- $x = L(F_2)$ if $k \notin L$.

- If $x = L(\langle\langle\, k \,?\, \text{table } T_1 : \text{table } T_2 \,\rangle\rangle)$, then $x = L(\text{table } T)$ where
  $T = \{(B \cup \{k\}, \bar{s}) \mid (B, \bar{s}) \in T_1, \bar{k} \notin B\}$
  $\cup\ \{(B \cup \{\bar{k}\}, \bar{s}) \mid (B, \bar{s}) \in T_2, k \notin B\}$.
  And so
  $x = \{(\emptyset, \bar{s}) \mid (B, \bar{s}) \in T_1, \bar{k} \notin B, B \cup \{k\} \sim L\}$
  $\cup\ \{(\emptyset, \bar{s}) \mid (B, \bar{s}) \in T_2, k \notin B, B \cup \{\bar{k}\} \sim L\}$.

  - If $k \in L$, then $B \cup \{\bar{k}\} \not\sim L$ and
    $B \cup \{k\} \sim L => \bar{k} \notin B$, 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.

□

B.3 Proof of Lemma 6

Lemma 6 (B)**.**

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

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

- If $B = \emptyset$, then $B \sim L$, so $x = L(V_1)$ as required.
- Otherwise, $B = B' \cup \{k\}$.

- If $B \sim L$, then

  $x = L(\langle\!\langle\, k \;?\; \langle\!\langle\, B' \;?\; V_1 : V_2 \,\rangle\!\rangle : V_2 \,\rangle\!\rangle)$

  $= L(\langle\!\langle\, B' \;?\; V_1 : V_2 \,\rangle\!\rangle)$ by Lemma 5, 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 5.
  * 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.

□

B.4 Lemma 7

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 7 (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.

B.5 Proof of Theorem 4 (Projection)

For convenience, we restate Theorem 4.

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 8. *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 $T = \{B.B', \bar{s}.\overline{s'}) \mid (B, \bar{s}) \in T_1, (B', \overline{s'}) \in T_2\}$.
  $L(T) = \{(B.B', \bar{s}.\overline{s'}) \mid (B, \bar{s}) \in T_1, (B', \overline{s'}) \in T_2, B.B' \sim L\}$, so this case holds.

- For case [F-CTXT], $e = E[e']$. By the antecedents of this rule

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

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

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

  Therefore, $L(\Sigma), L(E[e]) \Downarrow_{\emptyset} L(\Sigma'), L(V)$, as required.

- For case [F-STRICT], $e = S[\langle k \ ? \ V_1 : V_2 \rangle]$. By the antecedents of this rule

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

  We now consider each possible case for the next step in the derivation.

  – For subcase [F-LEFT], we know that $k \in pc, k \in L$ and

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

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

  – Subcase [F-RIGHT] holds by a similar argument.

– For subcase [F-SPLIT], $k \notin pc, \overline{k} \notin pc$ and

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

* 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 7, 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], $e = \text{fold } V_f\ V_p\ (\text{table } (B, \overline{s}).T)$. 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 8, $B \not\sim L$.
Therefore, $L(\text{table } (B, \overline{s}).T) = L(\text{table } T)$.
By the [F-FOLD-EMPTY] rule,

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

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

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

– If $B \sim L$, then $pc \cup B \sim L$.

  By induction,

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

  By Lemma 6, $L(V) = L(\langle\!\langle \ B \ ? \ V_2 : V_1 \ \rangle\!\rangle)$, as required.

– Otherwise, $B \not\sim L$, and therefore $pc \cup B \not\sim L$. By Lemma 7, $L(\Sigma_1) = L(\Sigma')$.

  By induction, $L(\Sigma), L(\text{fold } V_f \ V_b \ (\text{table } T)) \Downarrow_{\emptyset} L(\Sigma_1), L(V_1)$.

  $L(\text{table } (B, \overline{s}).T) = L(\text{table } T)$.

  By Lemma 6, $L(V) = L(\langle\!\langle \ B \ ? \ V_2 : V_1 \ \rangle\!\rangle)$, as required.

- For case [F-LEFT], $e = \langle k \ ? \ e_1 : e_2 \rangle$.

  By the antecedents of this rule, we have

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

  Since $k \in pc$, $L(e) = L(e_1)$.

  By induction, $L(\Sigma), L(e_1) \Downarrow_{\emptyset} L(\Sigma'), L(V)$.

- Case [F-RIGHT] holds by a similar argument.

- For case [F-SPLIT], $e = \langle k \ ? \ e_1 : e_2 \rangle$.

By the antecedents of this rule, we have

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

– If $k \in L$, then by induction $L(\Sigma), L(e_1) \Downarrow_{\emptyset} L(\Sigma_1), L(V_1)$.
  $L(\Sigma_1) = L(\Sigma')$ by Lemma 7, and by Lemma 5
  $L(V) = L(\langle\!\langle\, k \,?\, V_1 : V_2 \,\rangle\!\rangle) = L(V_1)$, as required.

– Otherwise $\bar{k} \in L$, so $L(\Sigma) = L(\Sigma_1)$ by Lemma 7.
  By induction, $L(\Sigma_1), L(e_2) \Downarrow_{\emptyset} L(\Sigma'), L(V_2)$,
  and by Lemma 5 $L(V) = L(\langle\!\langle\, k \,?\, V_1 : V_2 \,\rangle\!\rangle) = L(V_2)$, as required.

- For case [F-APP], $e = (\lambda x.e' \; V')$. By the antecedents of this rule,

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

  We know that $L(e) = L(\lambda x.e' \; V') = L(e'[x := V'])$.
  By induction, $L(\Sigma), L(e'[x := V']) \Downarrow_{\emptyset} L(\Sigma'), L(V)$, as required.

- For case [F-REF], $e = \text{ref } V'$. By the antecedents of this rule

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

  Without loss of generality, we assume that both evaluations allocate the same address a. Since $a \notin dom(\Sigma), a \notin dom(L(\Sigma))$.
  Also, we know that $\forall a' \in dom(\Sigma), \Sigma(a') = \Sigma'(a')$, and therefore $L(\Sigma(a')) = L(\Sigma'(a'))$.
  Since $pc \sim L$, $L(\Sigma'(a)) = L(\langle\!\langle\, pc \,?\, V' : 0 \,\rangle\!\rangle) = L(V')$ by Lemma 6. 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 6. And since $L(\langle\langle\, \emptyset\ ?\ V : \Sigma(a)\,\rangle\rangle) = L(V)$, this case holds.

$\square$