A

Multiple Facets for Dynamic Information Flow with Exceptions

Thomas H. Austin, San José State University
Tommy Schmitz, University of California, Santa Cruz
Cormac Flanagan, University of California, Santa Cruz

JavaScript is 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 full privileges. *Information flow controls* help prevent violations of data confidentiality and integrity.

This paper explores *faceted values*, a mechanism for providing information flow security in a dynamic manner that avoids the stuck executions of some prior approaches, such as the no-sensitive-upgrade technique. Faceted values simultaneously simulate multiple executions for different security levels in order to guarantee termination-insensitive noninterference. We also explore the interaction of faceted values with exceptions, declassification, and clearance.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features; D.4.6 [**Operating Systems**]: Security and Protection—*Information flow controls*

General Terms: Languages, Security

Additional Key Words and Phrases: Information flow control, dynamic analysis, JavaScript, web security

1. INTRODUCTION

JavaScript has helped to usher in a new age of richly interactive web applications. With minimal effort, a web developer can build an impressive site by composing code from multiple sources. Unfortunately, there are few restrictions on the included code, and it operates with the same authority as the web developer's own code. Advertising has been a particular source of malicious JavaScript. There are a wide array of security measures used to defend against these problems, but the bulk of them tend to rely on competent web developers. Given the mercurial nature of security challenges, even a conscientious web developer has difficulty keeping up with the latest trends and best practices. Another option is to bake security controls into the browser itself. This strategy has been part of browser design since nearly the beginning, but the controls have tended to be fairly minimal.

Information flow analysis offers the promise of a systemic solution to many of these security challenges. Dynamic information flow analysis is particularly challenging in the presence of implicit flows. Proposed mechanisms for dealing with implicit flows include the *no-sensitive-upgrade* semantics [Zdancewic 2002; Austin and Flanagan 2009] and the *permissive-upgrade* semantics [Austin and Flanagan 2010; Bichhawat et al. 2014]. Both semantics guarantee the key correctness property of termination-insensitive noninterference (TINI), which states that private inputs do not influence public outputs. (Private information can influence termination. In a sequential setting, this channel is limited to a brute

This work was supported by NSF grant CNS-0905650.
Author's addresses: T. H. Austin, Computer Science Department, San José State University; T. Schmitz and C. Flanagan, Computer Science Department, University of California at Santa Cruz.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 1539-9087/YYYY/01-ARTA $10.00
DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

force attack [Askarov et al. 2008], though it can be a high-bandwidth channel in concurrent systems [Moore et al. 2012]).

However, both of these semantics suffer from the same weakness: in the presence of subtle implicit flows that are hard to track, the semantics halts execution in order to avoid any (potential) information leak. Note that this fail-stop is not caused by the web application violating its security policy; instead it is a mechanism failure caused by the inability of the dynamic information flow analysis to track implicit flows. Thus, these dynamic analyses reject valid programs that conform to the security policy. Alternately, a dynamic monitor could ignore unsafe updates rather than terminating execution [Fenton 1974]; however, this approach might return results that are inconsistent with the standard semantics of the language.

An interesting solution to these mechanism failures is to simultaneously execute two copies of the target program: a high-confidentiality (H) process that has access to secret data, and a low-confidentiality (L) process that sees dummy default values instead of the actual secret data [Capizzi et al. 2008; Devriese and Piessens 2010]. This *multi-process execution* cleanly guarantees noninterference since no information flow is permitted between the two processes, and it also avoids mechanism failures. Unfortunately, for a web page with n principals (roughly, URL domains), we may require up to 2^n processes, one for each element in the powerset lattice for these principals.

In this paper, we take inspiration from multi-process execution to develop a single-process semantics that avoids stuck states. The key technical novelty is the introduction of a *faceted value*, which is a pair of two raw values that contain low and high confidentiality information, respectively. By appropriately manipulating these faceted values, a single process can *simulate* the two processes (L and H) of the multi-execution approach. The primary benefit of this approach is that when the two raw values in a faceted value are identical, we collapse the two simulated executions on identical data into a single execution to reduce overhead.

Faceted evaluation naturally extends to multiple principals and a complex security lattice, whereby a faceted value can contain many raw values, rather than just two. Note that in addition to avoiding stuck executions, multi-process execution also guarantees termination-sensitive noninterference (TSNI), in part because the H and L processes proceed independently. Under faceted evaluation, the H and L computations are coupled (and indeed often identical), so faceted evaluation only guarantees TINI rather than TSNI.

This paper includes a formal description of the faceted value approach to dynamic information flow analysis, and a proof that it achieves termination-insensitive noninterference. We present a *projection theorem* showing that a computation over faceted values simulates 2^n non-faceted computations, one for each element in the powerset security lattice. We also give a proof of *termination-insensitive semantics preservation*, showing that noninterfering programs exhibit equivalent behavior under faceted and standard semantics, modulo termination. In order to validate the utility of faceted evaluation in a web browsing context, we implement this mechanism inside the Firefox browser (using the Zaphod plug-in [Mozilla Labs Zaphod 2010]) and use this implementation to compare the performance of faceted values against multi-process execution.

This paper extends our original conference paper [Austin and Flanagan 2012] to support exceptions, which introduce additional subtleties. Additionally, we discuss *semantics preservation* (Section 3.3), *clearance* using faceted values (Section 6), the *introspection of faceted values* (Section 8), and a *failure-oblivious* information flow monitor (Section 4.3). We also provide more in-depth discussion of our implementation (Section 9). Finally, this paper simplifies our technical development, for example by eliminating the substitution argument in our evaluation relation.

1.1. Overview of Faceted Evaluation

To motivate the utility of faceted values in dynamic information flow, we start by considering 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 provide a means to represent 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:

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

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 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

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

where only private observers can see V, and where public or unauthorized observers instead see $\bot$ (roughly meaning undefined).

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 above, correct handling of implicit flows requires the introduction of the more general notion of faceted values $\langle k \ ? \ V_{\text{H}} : V_{\text{L}} \rangle$, in which the public facet V_{L} is a real value and not simply $\bot$. In particular, the post-assignment value for `y` is 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, this paper develops 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. In particular, if $C[\bullet]$ is any program context, then the computation $C[\langle k \ ? \ V_{\text{H}} : V_{\text{L}} \rangle]$ appears to behave like $C[V_{\text{H}}]$ from the perspective of a private observer, and behaves 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 a short proof of termination-insensitive noninterference, shown in Section 3.2.

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

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

Figure 1: A JavaScript Function with Implicit Flows

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

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 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.

1.2. Handling Implicit Flows

The key challenge in dynamic information flow analysis lies in handling implicit flows. To illustrate this difficulty, consider the code in the first column of Figure 1, which is adapted from an example by Fenton [1974]. Here, the function `f(x)` returns the value of its boolean

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

argument `x`, but it first attempts to "launder" this value by encoding it in the program counter.

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 more traditional $\texttt{false}^k$ and $\texttt{true}^k$) 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 (`$\bot$`) ...` 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 different dynamic information flow semantics. While the prior semantics that we discuss here have no notion of facets, explaining them in terms of faceted values is illuminating.

Naive. An intuitive strategy for handling the assignment `y=false` that is conditional on the private input `x` is to simply set `y` to a private false value, traditionally written as $\texttt{false}^k$, in order to reflect that this value depends on private inputs. In faceted notation, where $\texttt{false}^k$ is written as $\langle k\ ?\ \texttt{false} : \bot\rangle$, the problem with the approach becomes clear, as we lose the critical information that a public observer should still see $\texttt{y} = \texttt{true}$; instead a public observer now sees $\bot$. The next conditional branch on `y` exploits this confusion. Since `y` is $\langle k\ ?\ \texttt{false} : \bot\rangle$, the branch is not executed, so `z` remains `true`, and so $\texttt{f}(\langle k\ ?\ \texttt{true} : \bot\rangle)$ returns `true`, as illustrated in column 3. Thus, this naive strategy fails to ensure TINI, since the public output of `f` leaks the contents of its private input.

Various prior approaches attempt to close this information leak without introducing full faceted values, with mixed results.

No-Sensitive-Upgrade. With the *no-sensitive-upgrade* (NSU) check [Zdancewic 2002; Austin and Flanagan 2009], execution halts 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 1. This strategy guarantees TINI, but only at the expense of getting stuck on some implicit flows.

Permissive-Upgrade. A more flexible approach is to permit the implicit flow caused by the conditional assignment to `y`, but to record 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 the faceted value $\langle k\ ?\ \texttt{false} : *\rangle$, where "$*$" denotes that the public facet is an unknown, non-$\bot$ value.[2]

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 1, the execution gets stuck as shown in the PU column.

Failure-Oblivious Information Flow Monitor. An alternate approach provides noninterference by ignoring potentially unsafe updates rather than terminating execution [Fenton 1974]. We refer to this type of monitor as a *failure-oblivious information flow monitor* (FO), drawing a parallel to failure-oblivious computing [Rinard et al. 2004]. As shown in the FO column of Figure 1, the update to the public variable `y` is ignored since this update is conditional on the sensitive value of `x`. Therefore on the case where $\texttt{x}=\langle k\ ?\ \texttt{false} : \bot\rangle$, the

[2]The original paper [Austin and Flanagan 2010] used $\texttt{false}^P$ to represent $\langle k\ ?\ \texttt{false} : *\rangle$, where the superscript P denotes "partially leaked".

Figure 2: Trade-offs Between Information Flow Control Mechanisms

	Fail-Stop Monitor	Failure-Obliv. Monitor	Secure Multi-Execution	Faceted Evaluation
Runtime overhead	low	low	high	mid
Transparent controls	yes	no	yes*	yes*
Straightforward declassification	yes	yes	no	yes
Independent from language	no	no	yes	no
Easy to parallelize	no	no	yes	no
Protects termination channel	no	no	yes	no
Protects timing channels	no	no	yes+	no

*: Controls are transparent for authorized views/channels.
+: Defends timing channels with *sequential* SME for some views. See Kashyap et al. [2011] for a more detailed discussion.

result does not match the standard semantics. Note that this approach is related to Fenton's original approach, except that labels on a value may be modified in some circumstances.

Faceted Evaluation. As shown in the last column of Figure 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.

1.3. Comparison of Information Flow Control Mechanisms

Figure 2 highlights trade-offs between various mechanisms for information flow.

One important consideration for any security control is its *transparency*; that is, if an execution completes, is there any observable difference in its results compared to those of an execution without any security controls. The no-sensitive-upgrade and permissive-upgrade semantics (jointly listed as the fail-stop monitors) have this property, though the failure-oblivious monitor does not. Faceted evaluation and secure multi-execution (SME) both support transparency for authorized views or channels[3]. However, if sensitive information would leak to an unauthorized view or channel, then transparency is broken in order to guarantee noninterference without terminating execution. Transparency might also be violated if an observer can view multiple channels, since the observed outputs might not be in the expected order; Rafnsson and Sabelfeld [2013] and Zanarini et al. [2013] offer a more in depth discussion on this point, and discuss how SME can address this issue.

[3]Our *termination-insensitive semantics preservation theorem* in Section 3.3 states this quality more formally for faceted evaluation.

Secure multi-execution is an approach that naturally lends itself to parallelization. Since secure multi-execution uses separate processes, it also protects the termination channel (thus providing *termination-sensitive* noninterference) and it offers some defense against timing channels[4]. While these are strong advantages, they are not without a cost. *Declassification* becomes difficult; the two processes must be coordinated, which reintroduces both the termination channel and timing channels. In contrast, a faceted value captures multiple views of a single value. Therefore, declassification can be done by restructuring the faceted value to reflect the change in the different views. Likewise, declassification in the monitor based approaches is simply a matter of changing the security label.

Finally, ease of implementation is also an important concern. Unique among these mechanisms, secure multi-execution may be designed to be independent from the language implementation. De Groef et al. [2012] do an excellent job of discussing the trade-offs between using secure multi-execution on the whole web browser vs. applying it only on the web scripts used in the browser. They note that integrating the web scripts is more efficient and allows them to make more fine-grained decisions, though at the cost of implementation complexity. These same arguments seem to apply within the language implementation as well; controls integrated into the run-time environment make the language implementation more difficult to maintain, but may allow for more fine-grained decisions.

1.4. Dynamic Information Flow Control for Exceptions

As outlined above, faceted execution provides one approach for correctly handling side-effects and the associated implicit flows. A second challenging problem in dynamic information flow systems is how to handle control flow effects such as exceptions, since the standard propagation of exceptions up the evaluation stack can leak information [Stefan et al. 2011].

To illustrate this problem, consider the function `g(x)` shown in Figure 3. The figure also shows the behavior of this function on two secret input arguments, $\texttt{false}^k$ and $\texttt{true}^k$, under a naive (that is, non-faceted and incorrect) semantics for exceptions[5]. For the input $\texttt{x} = \texttt{false}^k$, the statement `if (x) throw ""` does not raise an exception, but then the following code binds `y` to `true`, which leaks information about the secret value of `x`. In particular, if `x` is $\texttt{true}^k$ (see right column), then an exception is thrown and caught, after which `y` is bound to $\texttt{false}^k$. Thus, the value of `y` (`true` or $\texttt{false}^k$) reflects the original argument `x` but the labels for `y` do not fully track this dependency. Repeating this code pattern a second time result in the variable `z` containing a fully declassified copy of the input argument `x`. Thus, under the naive semantics for exceptions, the function `g(x)` provides a way to remove the privacy label from its boolean argument.

Various semantics have been proposed for tracking information flow through exceptions. For example, in [Stefan et al. 2011], the `toLabeled` construct catches all exceptions and converts them to normal values. [Hritcu et al. 2013] propagates exceptions along data-flow paths (like `NaN`s) rather than on traditional control-flow paths. Both of these approaches involve changing the conventional semantics of exceptions, and so are not entirely satisfactory.

In contrast, faceted evaluation allows us to deal with the problems of both side-effects (illustrated in Figure 1) and control-effects (illustrated in Figure 3) in a consistent manner. In particular, faceted semantics propagates exceptions along control-flow paths, yet still provides the desired noninterference guarantees, as we describe in Section 5.

---

[4]To protect against timing channels, the low-confidentiality process must be run to completion before the high-confidentiality process begins execution. Note that *sibling* processes, where neither has strictly more access to confidential data, do not gain this benefit. Kashyap et al. [2011] discuss this property in more detail.

[5]This example uses a try-catch expression instead of the more conventional try-catch statement in order to illustrate the problem of control-effects in code without side-effects.

Figure 3: Naive Semantics for Exceptions

Function g(x)	x = false^k	x = true^k
`let y = try {`		
`if (x) throw "";`	then branch not executed	exception thrown with $pc = \{H\}$
`true;`	`y` bound to `true`	
`} catch(e) {`		exception caught, $pc = \{H\}$
`false;`		`y` bound to `false`^{k}
`}`		
`let z = try {`		
`if (y) throw "";`	exception raised	then branch not executed
`true;`		`true`
`} catch(e) {`	exception caught, $pc = \{\}$	
`false;`	`z` bound to `false`	
`}`		
`return z;`	Returns public `false`	Returns public `true`

2. A PROGRAMMING LANGUAGE WITH FACETS

We formalize faceted evaluation for dynamic information flow in terms of the idealized language λ^{facet} shown in Figure 4. This language extends the λ-calculus with mutable reference cells, reactive I/O, a special value $\bot$, and a mechanism for creating faceted values. Despite its intentional minimality, this language captures the essential complexities of dynamic information flow in more realistic languages, since it includes key challenges such as heap allocation, mutation, implicit flows, and higher-order function calls. In particular, conditional tests can be Church-encoded in the usual fashion.

Expressions in λ^{facet} 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 ($\mathtt{ref}\ e$), dereference ($!e$), and update ($e_1 \mathtt{:=} e_2$) a reference cell. To model JavaScript's interactive nature, λ^{facet} also supports reading from ($\mathtt{read}(f)$) and writing to ($\mathtt{write}(f, e)$) external resources such as files.

The expression $\langle k\ ?\ e_1 : e_2 \rangle$ creates a faceted value where the value of e_1 is considered secret to principal k; observers that cannot see k's private data will instead see the public facet produced by e_2. We initially use the terms *label* and *principal* as synonyms and focus primarily on confidentiality—Section 7 later introduces integrity labels in the context of declassification.

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 $\langle k\ ?\ V : \bot \rangle$, which denotes a value V that is private to principal k, with no corresponding public value.

2.1. 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 5. Each reference cell is allocated at an address a, and the store σ maps addresses to values. The store also maps 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, and use $\emptyset$ to denote both the empty store and the empty substitution.

Figure 4: The source language λ^{facet}

Syntax:

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

Standard encodings:

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

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 5, which are mostly straightforward. For example, the rule [S-APP] evaluates the body of the called function, where the notation $e[x := v]$ maps x to v wherever it occurs in the expression e.

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

Figure 5: Standard Semantics

Runtime Syntax

$$\begin{array}{llcl} a & \in & \mathit{Address} & \\ \sigma & \in & \mathit{store} & = & (\mathit{Address} \rightarrow_p \mathit{value} \ \cup \ \mathit{File} \rightarrow \mathit{value}^*) \\ v & \in & \mathit{value} & ::= & c \mid a \mid (\lambda x.e) \mid \bot \\ w & \in & \mathit{value}^* & \end{array}$$

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

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

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

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

$$\frac{\sigma(f) = v.w \quad \sigma' = \sigma[f := w]}{\sigma, \mathtt{read}(f) \downarrow \sigma', v} \quad [\text{S-READ}]$$

$$\frac{\sigma, e \downarrow \sigma_1, v \quad \sigma' = \sigma_1[f := \sigma_1(f).v)]}{\sigma, \mathtt{write}(f, e) \downarrow \sigma', v} \quad [\text{S-WRITE}]$$

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

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

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

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

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

3. FACETED EVALUATION

Having defined the standard semantics of the language, we now extend that semantics with faceted values that dynamically track information flow and provide noninterference guarantees.

Figure 6 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

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

which contain both a private facet V_{H} and a public facet V_{L}. For instance, the value $\langle k \ ? \ 42 : 0\rangle$ indicates that 42 is confidential to the principal k, and unauthorized viewers instead see

Figure 6: Faceted Evaluation Semantics

Runtime Syntax

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\Sigma, e \Downarrow_{pc} \Sigma', V \quad pc \text{ visible to } view(f) \quad L = view(f) \quad v = L(V)}{\Sigma, \texttt{write}(f, e) \Downarrow_{pc} \Sigma'[f := \Sigma'(f).v], V}\ \text{[F-WRITE1]}$$

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

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

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

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

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

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

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

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 $\bot$.

Figure 7: Faceted Evaluation Auxiliary Functions

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

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

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

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

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 $\langle\!\langle\, pc \ ? \ V_1 : V_2 \,\rangle\!\rangle$ 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.

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

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

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-sensitive 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 $\perp$) 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:

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

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 $\langle k\ ?\ V_{\mathrm{H}} : V_{\mathrm{L}} \rangle$, then the rule [FA-SPLIT] applies *both* V_{H} and V_{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 $\perp$ can be applied as a function and returns $\perp$ via [FA-$\perp$] (much like the earlier [S-APP-$\perp$] rule).

As an example, consider the function application (f 4) where f is a private function represented as $\langle k\ ?\ (\lambda x.e) : \perp \rangle$. The rules [F-APP] and [FA-SPLIT] decompose the application (f 4) into two separate applications: $((\lambda x.e)\ 4)$ and $(\perp\ 4)$. The first application evaluates normally via [FA-FUN] to a result, say V, and the second application evaluates to $\perp$ via [FA-$\perp$], so the result of the call is $\langle k\ ?\ V : \perp \rangle$, 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-DEREF] uses the helper function $\mathtt{deref}(\Sigma, V_a, pc)$, defined in Figure 7, to decompose V_a into appropriate addresses, 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.

In a similar manner, the rule [F-ASSIGN] uses the helper function $assign(\Sigma, pc, V_a, V)$ 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, to avoid information leaks via implicit flows. The *assign* function is also defined in Figure 7.

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 $view(f) = \{k_1, \ldots, k_n\}$ 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 $view(f)$ actually reads from the file via [F-READ1]; all other executions are no-ops via [F-READ2]. Note that this semantics allows only $view(f)$ (and not higher views) to read from f, and so the value v is converted into a faceted value $\langle\!\langle\, pc'\ ?\ v : \perp \,\rangle\!\rangle$, where pc' is a program counter label that denotes exactly $view(f)$ (and not higher views).[6]

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 $L = view(f)$ to project the faceted value V produced by e into a corresponding non-faceted value v written to the file.

For simplicity, we Church-encode conditional branches as function calls, and so 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 6. To provide helpful intuition, however, Figure 8 sketches alternative direct rules for evaluating a conditional test $\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3$. In particular, if e_1 evaluates to a faceted value $\langle k\ ?\ V_H : V_L \rangle$, the if

[6]Alternatively, we could also allow all views greater than $view(f)$ to read the file, as in SME [Devriese and Piessens 2010], by maintaining a per-view (that is, faceted) index into the contents of f.

Figure 8: Faceted Evaluation Semantics for Derived Encodings

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

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

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

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

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.)

3.1. The Projection Property

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_{H} in a value $\langle k\ ?\ V_{\text{H}} : V_{\text{L}} \rangle$ only when $k \in L$, and sees V_{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:

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

We extend L to also project faceted stores $\Sigma \in \mathit{Store}$ into non-faceted stores of the standard semantics. A file f is visible only to $\mathit{view}(f)$, and appears empty (ϵ) to all other views.

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

We also use a view L to operate on expressions, where this operation eliminates faceted expressions and also performs access control on I/O operations by eliminating accesses to

files that are not authorized under that view:

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

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.

Definition 3.1. 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:

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

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

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

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

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

Next, from the perspective of any view L, if 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 pc is not visible to L, then this operation has no observable effect.

LEMMA 3.4. *If* $\Sigma' = \mathit{assign}(\Sigma, pc, V_1, V_2)$ *then*

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

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

LEMMA 3.5. *Suppose* pc *is not visible to* L *and that*

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

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

PROOF. See Appendix. □

We now prove our central projection theorem, which shows that an evaluation under the faceted semantics of Figure 6 simulates many evaluations under the standard semantics, one for each possible view for which pc is visible.

THEOREM 3.6 (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. □

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.

3.2. Termination-Insensitive Noninterference

The projection property enables a simple proof of noninterference; 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 ($e_1 \sim_L e_2$):

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

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

THEOREM 3.7 (TERMINATION-INSENSITIVE NONINTERFERENCE). *Let L be any view.*

$$\begin{array}{ll} \textit{Suppose} & \\ & \Sigma_1 \sim_L \Sigma_2 \\ & \Sigma_1, e \Downarrow_{\emptyset} \Sigma_1', V_1 \\ & \Sigma_2, e \Downarrow_{\emptyset} \Sigma_2', V_2 \\ \textit{Then:} & \\ & \Sigma_1' \sim_L \Sigma_2' \\ & V_1 \sim_L V_2 \end{array}$$

PROOF. By the Projection Theorem:

$$\begin{array}{l} L(\Sigma_1), L(e_1) \downarrow L(\Sigma_1'), L(V_1) \\ L(\Sigma_2), L(e_2) \downarrow L(\Sigma_2'), L(V_2) \end{array}$$

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[7]. □

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

[7]Without loss of generality, we assume that any allocations of reference cells in the two executions use the same addresses. We refer the interested reader to [Banerjee and Naumann 2002] for an alternative proof technique that does not rely on this assumption, but which involves a more complicated compatibility relation on stores.

3.3. Termination Insensitive Semantics Preservation

We next study the issue of *semantics preservation*; that is, how the faceted relation $\Downarrow$ relates to the standard evaluation relation of $\downarrow$. For programs that leak information under standard evaluation, clearly faceted evaluation needs to perturb that behavior in order to guarantee noninterference. For programs are already noninterfering under standard evaluation, we show that (modulo termination) faceted evaluation is essentially equivalent to standard evaluation.

Intuitively, we say a non-faceted expression e is noninterfering if its evaluation never leaks information from a secret file to a public file, or more generally from one file f_1 to a second file f_2 where f_1 and f_2 are visible to different views, i.e. $\mathit{view}(f_1) \neq \mathit{view}(f_2)$.

To help formalize this idea, we note that since $\mathit{store} \subset \mathit{Store}$, the relation $\sim_L$ defined above is applicable to (non-faceted) stores, and identifies stores that differ only in files that are not visible to view L.

LEMMA 3.8. *$\sigma_1 \sim_L \sigma_2$ iff $\forall a.\ \sigma_1(a) = \sigma_2(a)$ and $\forall f.\ \mathit{view}(f) = L \Rightarrow \sigma_1(f) = \sigma_2(f)$.*

We define a (non-faceted) expression e to be *noninterfering* on a (non-faceted) store σ_1 if for any view L, whenever

$$\begin{array}{c} \sigma_1 \sim_L \sigma_2 \\ \sigma_1, e \downarrow \sigma_1', v_1 \\ \sigma_2, e \downarrow \sigma_2', v_2 \end{array}$$

we have that the resulting values are identical ($v_1 = v_2$), and moreover all L-visible files in the final stores agree, i.e. $\forall f$ with $\mathit{view}(f) = L.\ \sigma_1'(f) = \sigma_2'(f)$. That is, the L-invisible difference in the initial file system does not leak into the result or become L-visible in the final file system.

Using this definition, we now show that (modulo termination) faceted evaluation behaves the same as standard evaluation on noninterfering programs:

THEOREM 3.9 (TERMINATION-INSENSITIVE SEMANTICS PRESERVATION). *Let e be a non-faceted expression e that is noninterfering on σ and suppose*

$$\begin{array}{c} \sigma, e \downarrow \sigma', v \\ \sigma, e \Downarrow_{\emptyset} \Sigma', V \end{array}$$

Then $\forall f.\sigma'(f) = \Sigma'(f)$, and $\forall L.v = L(V')$.

PROOF. Pick any view L. By the projection theorem,

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

Clearly $L(e) = e$ as e is non-faceted. Also, L is idempotent, so $L(\sigma) = L(L(\sigma))$, and hence $\sigma \sim_L L(\sigma)$. Thus, by the noninterference of e on the two executions

$$\begin{array}{c} \sigma, e \downarrow \sigma', v' \\ L(\sigma), e \downarrow L(\Sigma'), L(V) \end{array}$$

we have that $v' = L(V)$. Moreover, for any f with $\mathit{view}(f) = L$, $\sigma'(f) = L(\Sigma')(f) = \Sigma'(f)$. Since the choice of L above was arbitrary, we have $\forall f.\sigma'(f) = \Sigma'(f)$, as required. $\square$

3.4. 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.

Figure 9: Efficient Construction of Faceted Values

$$
\begin{array}{lcl}
\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 \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 \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 \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 \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 \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 \text{ where } k < \mathit{head}(V_n) \\
 & & \qquad \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 \text{ where } k < \mathit{head}(V_n) \\
 & & \qquad \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 \text{ where } k < \mathit{head}(V_o) \\
 & & \qquad \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 \text{ where } k < \mathit{head}(V_n) \\
 & & \qquad \text{and } k < \mathit{head}(pc)
\end{array}
$$

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.

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

Figure 9 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 3.2.

4. COMPARISON TO PRIOR SEMANTICS

Prior work presented the *no-sensitive-upgrade* (NSU) semantics [Zdancewic 2002; Austin and Flanagan 2009] and the *permissive-upgrade* (PU) semantics [Austin and Flanagan 2010; Bichhawat et al. 2014] 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 and omit I/O operations, since these prior semantics were formalized under these assumptions. We also use the optimized facet-construction operation from Figure 9 in order to avoid reasoning about dead facets.

In order to better highlight the relation between information flow monitors and faceted evaluation, we also include a *failure-oblivious information flow monitor*. Inspired by the failure-oblivious computing work of Rinard et al. [2004], this monitor provides termination-

Figure 10: No Sensitive Upgrade Semantics

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

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

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

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

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

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

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

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

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

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

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

Figure 11: Permissive-Upgrade Semantics (extends Figure 10)

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

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

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

insensitive noninterference by ignoring unsafe operations. This strategy may produce inaccurate results, but avoids stuck executions.

4.1. 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 10. These rules are somewhat analogous to the faceted evaluation rules of Figure 6, but with some noticeable limitations and 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-sensitive 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:

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

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 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 4.1 (FACETED EVALUATION GENERALIZES NSU EVALUATION). *If* $\Sigma, e \Downarrow_{pc} \Sigma', V$ *then* $\Sigma, e \Downarrow\!\!\Downarrow_{pc} \Sigma', V$.

PROOF. See Appendix. □

4.2. 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 [Austin and Flanagan 2010]. Essentially, the PU semantics works by tracking *partially leaked* data, which we represent here as a faceted value $\langle k \ ? \ R : * \rangle$.[8]

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

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. 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 [Austin and Flanagan 2010].

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 4.2 (FACETED EVALUATION GENERALIZES PU EVALUATION). *If* $\Sigma, e \Downarrow_{pc} \Sigma', V$, *then* $\Sigma, e \Downarrow\!\!\Downarrow_{pc} \Sigma', V$.

PROOF. See Appendix. □

Again, the converse to this theorem does not hold, since Figure 1 shows an execution that gets stuck under the permissive-upgrade semantics but not under the faceted semantics.

4.3. Failure-Oblivious Information Flow Monitor

The monitor-based approaches discussed in this section so far use a *fail-stop* approach; that is, they terminate execution in order to prevent the loss of sensitive information. However, Rinard et al. [2004] have shown that applications often work quite well with minor errors. *Failure-oblivious computing* exploits this quality to allow applications with memory errors to continue operation. In a similar spirit, we develop a *failure-oblivious information flow monitor* that does not terminate the application to guarantee termination-insensitive noninterference.

Fenton [1974] makes use of a similar strategy, though labels[9] are fixed to be either public or private and may not change.

The central idea of this approach is to ignore operations that might lead to a loss of sensitive information. In that vein, we update the no-sensitive-upgrade semantics with the following additional rules for assignment.

---

[8] In [Austin and Flanagan 2010], these partially leaked values were represented as R^P, with a superscript P denoting partially leaked.

[9] "Data marks" in Fenton's terminology.

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

In contrast with the other monitor-based approaches, the failure-oblivious strategy does not get stuck on the example from Figure 1, and yet it still maintains noninterference. However, for the case where $x = \langle k \ ? \ \texttt{true} : \bot \rangle$, this approach returns a value of `false`. Therefore, this approach satisfies noninterference but does not guarantee transparency, and so provides inaccurate results to both authorized viewers and unauthorized viewers.

In contrast, secure multi-execution and faceted evaluation only alter the semantics of executions that violate noninterference. Observers with the proper permissions see results that match the standard semantics, thereby guaranteeing transparency for those views.

The failure-oblivious information flow monitor strategy presents an additional option in the design space of information flow controls. When the secure multi-execution and faceted evaluation strategies are too heavyweight, and some inaccuracy is acceptable in trade for increased availability, the failure-oblivious approach may be a good solution.

5. FACETED EVALUATION WITH EXCEPTIONS

We next extend the faceted semantics to support throwing and catching exceptions.

Languages such as JavaScript provide exceptions to facilitate error handling and non-local control flow. Exceptions introduce additional complexities for our analysis, since some projections of a faceted execution could terminate normally while others throw exceptions, and we need to ensure that each view only sees the appropriate behavior.

We extend the syntax of λ^{facet} as follows:

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

For the sake of simplicity, we do not add information to exceptions.

Figure 12 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 is 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 5. The other rules from Figure 5 remain unchanged, and therefore are not repeated.

5.1. Faceted Exceptions

Figures 13 and 14 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* ($\langle k \ ? \ B_1 : B_2 \rangle$). We extend the operator $\langle\!\langle\, k \ ? \ V_1 : V_2 \,\rangle\!\rangle$ to handle behaviors in a straightforward manner:

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

Figure 12: Standard Semantics

Runtime Syntax:

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

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

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

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

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

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

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

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

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

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

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

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

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

Handling exceptions under faceted evaluation requires some care, since in an application $(e_1\ e_2)$, if e_1 evaluates to `raise` for some view, then e_2 should not be evaluated for that view. Similarly, exception handling code should only be executed 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 14.

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 $L(B) \neq \texttt{raise}$: see Figure 14.

With this relation, e is evaluated normally if B is a value, as specified by [FB-NORMAL]. 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.

Figure 13: Core Rules for Faceted Evaluation with Exception Handling

Runtime Syntax

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 14: Faceted Evaluation Rules for Application and Exceptions

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

$$\frac{\Sigma, e[x := V] \Downarrow_{pc} \Sigma', B'}{\Sigma, ((\lambda x.e)\ V) \Downarrow_{pc}^{\texttt{app}} \Sigma', B'} \quad [\text{FA-FUN1}]$$

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

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

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

$$\frac{\begin{array}{c} k \notin pc, \overline{k} \notin pc \\ \Sigma, (B_{\text{H}}\ B_2) \Downarrow_{pc \cup \{k\}}^{\texttt{app}} \Sigma_1, B'_{\text{H}} \\ \Sigma_1, (B_{\text{L}}\ B_2) \Downarrow_{pc \cup \{\overline{k}\}}^{\texttt{app}} \Sigma', B'_{\text{L}} \\ B = \langle\!\langle\ k\ ?\ B'_{\text{H}} : B'_{\text{L}}\ \rangle\!\rangle \end{array}}{\Sigma, (\langle k\ ?\ B_{\text{H}} : B_{\text{L}} \rangle\ B_2) \Downarrow_{pc}^{\texttt{app}} \Sigma', B} \quad [\text{FA-SPLIT1}]$$

$$\frac{k \in pc \qquad \Sigma, (B_{\text{H}}\ B_2) \Downarrow_{pc}^{\texttt{app}} \Sigma', B}{\Sigma, (\langle k\ ?\ B_{\text{H}} : B_{\text{L}} \rangle\ B_2) \Downarrow_{pc}^{\texttt{app}} \Sigma', B} \quad [\text{FA-LEFT1}]$$

$$\frac{\overline{k} \in pc \qquad \Sigma, (B_{\text{L}}\ B_2) \Downarrow_{pc}^{\texttt{app}} \Sigma', B}{\Sigma, (\langle k\ ?\ B_{\text{H}} : B_{\text{L}} \rangle\ B_2) \Downarrow_{pc}^{\texttt{app}} \Sigma', B} \quad [\text{FA-RIGHT1}]$$

$$\frac{\begin{array}{c} \texttt{raise} \in \langle k\ ?\ B_{\text{H}} : B_{\text{L}} \rangle \\ k \notin pc, \overline{k} \notin pc \\ \Sigma, (R\ B_{\text{H}}) \Downarrow_{pc \cup \{k\}}^{\texttt{app}} \Sigma_1, B'_{\text{H}} \\ \Sigma_1, (R\ B_{\text{L}}) \Downarrow_{pc \cup \{\overline{k}\}}^{\texttt{app}} \Sigma', B'_{\text{L}} \\ B = \langle\!\langle\ k\ ?\ B'_{\text{H}} : B'_{\text{L}}\ \rangle\!\rangle \end{array}}{\Sigma, (R\ \langle k\ ?\ B_{\text{H}} : B_{\text{L}} \rangle) \Downarrow_{pc}^{\texttt{app}} \Sigma', B} \quad [\text{FA-SPLIT2}]$$

$$\frac{\begin{array}{c} \texttt{raise} \in \langle k\ ?\ B_{\text{H}} : B_{\text{L}} \rangle \\ k \in pc \qquad \Sigma, (R\ B_{\text{H}}) \Downarrow_{pc}^{\texttt{app}} \Sigma', B \end{array}}{\Sigma, (R\ \langle k\ ?\ B_{\text{H}} : B_{\text{L}} \rangle) \Downarrow_{pc}^{\texttt{app}} \Sigma', B} \quad [\text{FA-LEFT2}]$$

$$\frac{\begin{array}{c} \texttt{raise} \in \langle k\ ?\ B_{\text{H}} : B_{\text{L}} \rangle \\ \overline{k} \in pc \qquad \Sigma, (R\ B_{\text{L}}) \Downarrow_{pc}^{\texttt{app}} \Sigma', B \end{array}}{\Sigma, (R\ \langle k\ ?\ B_{\text{H}} : B_{\text{L}} \rangle) \Downarrow_{pc}^{\texttt{app}} \Sigma', B} \quad [\text{FA-RIGHT2}]$$

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

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

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

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

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

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

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

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

Figure 15: Faceted Evaluation Auxiliary Functions with Exceptions

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

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

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

LEMMA 5.1. *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 5.2. *If* $\Sigma, (B_1 \ B_2) \Downarrow_{pc}^{\mathsf{app}} \Sigma', B$ *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 5.3. *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 15, 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`, $\perp$ will be entered into the resulting store for address a. Similarly, storing a faceted behavior containing `raise` will result in $\perp$ being stored in place of all `raise` facets[10]. 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 new rules [FE-ASSIGN] and [FE-DEREF] rule use new versions of the *assign* and *deref* functions, defined in Figure 15, that account for the possibility of `raise`.

The [FA-SPLIT1], [FA-LEFT1], [FA-RIGHT1] rules replace [FA-SPLIT], [FA-LEFT], and [FA-RIGHT]. They differ only in that they handle faceted behaviors instead of faceted values. The [FA-SPLIT2], [FA-LEFT2], and [FA-RIGHT2] rules handle faceted behaviors on the right hand side of the application. The [FE-VAL] and [FA-$\perp$] rules remains unchanged from the equivalent rules in Figure 6. The [FE-SPLIT], [FE-LEFT], [FE-RIGHT], [FA-FUN1], [FE-READ1], [FE-READ2], and [FE-WRITE1] rules are modified only in that they return behaviors instead of values.

5.2. 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.

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

We extend Lemmas 3.2, 3.3, 3.4, and 3.5 to handle faceted behaviors and to account for the presence of `raise`.

LEMMA 5.4. *If* $B = \langle\!\langle\, pc \ ?\ B_1 : B_2 \,\rangle\!\rangle$ *then*

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

LEMMA 5.5. *If* $B' = \mathit{deref}(\Sigma, B, pc)$ *then* $\forall L$ *consistent with* pc

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

LEMMA 5.6. *If* $\Sigma' = \mathit{assign}(\Sigma, pc, B_1, B_2)$ *then*

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

---

[10] `raise` $\in B$ indicates that either $B =$ `raise`, or that $B = \langle k \ ?\ B_{\mathrm{H}} : B_{\mathrm{L}} \rangle$, where either `raise` $\in B_{\mathrm{H}}$ or `raise` $\in B_{\mathrm{L}}$.

LEMMA 5.7. *Suppose pc is not visible to L and that*

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

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

PROOF. See Appendix. □

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 5.8 formalizes this property.

LEMMA 5.8. *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 5.9 (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. □

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

THEOREM 5.10 (TERMINATION-INSENSITIVE NONINTERFERENCE WITH EXCEPTIONS). *Let L be any view.*

Suppose

$$\Sigma_1 \sim_L \Sigma_2$$
$$\Sigma_1, e \Downarrow_{\emptyset} \Sigma_1', B_1$$
$$\Sigma_2, e \Downarrow_{\emptyset} \Sigma_2', B_2$$

Then

$$\Sigma_1' \sim_L \Sigma_2'$$
$$B_1 \sim_L B_2$$

PROOF. Holds by a similar argument as in the proof for Theorem 3.7. □

6. FACET EVALUATION WITH CLEARANCE

One limitation of dynamic information flow is that untrusted code operating on private data could use timing or termination effects to leak some information about that data, as in the following code, which diverges when `secret` is true:

```
while (secret) {};
```

To limit this problem, Stefan et al. [2011] introduce the promising notion of *clearance*, where low clearance code is not even permitted to access private data. Thus, the following code snippet would fail at the read of `secret` in a low-clearance context, independent of the value of that secret data.

```
function leak() {
  while (secret) {};
}
lowerClearance(L);
leak();
```

One limitation of this notion of clearance is that it only applies to code executed within a particular dynamic scope. Consequently, in a language with unrestricted side-effects like JavaScript, adversarial code can escape clearance constraints by registering callbacks, as in

```
function leak() {
  while (secret) {};
}
lowerClearance(L);
setTimeout(leak, 0);
```

The lower clearance applies when `setTimeout` is called, but not when `leak` is called by JavaScript's event loop, so this code can still use divergence to leak information about `secret`.

We now show how faceted evaluation naturally supports a somewhat richer notion of clearance. We use the idiom $\langle k \ ? \ \texttt{skip} : \texttt{leak()} \rangle$ to execute the function `leak` with low clearance $pc = \{\overline{k}\}$; this idiom is analogous to the `lowerClearance(L)` call above. Thus the function `leak` is executed with $pc = \{k\}$, and so that computation only sees the public `false` facet of `secret`.

```
let secret = <k?true:false>;
function leak() {
  while (secret) {};
}
<k?skip:leak()>;
```

Moreover, if a computation stores a callback somewhere, then that callback is tainted with label $\{\overline{k}\}$, and if the callback is later called from an unconstrained context (with $pc = \{\}$), the tainted callback is still unable to access k-sensitive data. Thus, in the following example, faceted evaluation still prevents low-clearance code from leaking the secrete value `true` via termination; instead the low-clearance code still always sees the public facet `false` of `secret`.

```
let secret = <k?true:false>;
let callback = function (){};
function leak() {
  callback = function() {
    while (secret) {};
  }
}
<k?skip:leak()>;
callback();
```

7. FACET DECLASSIFICATION

For many real systems, noninterference is too restrictive. 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 [Devriese and Piessens 2010], declassification is rather challenging. The L and H processes must be coordinated in a careful manner, with the attendant problems involved in sharing data between multiple processes. Additionally, declassification may re-introduce the termination channel, losing some benefits of the multi-execution approach. In contrast, faceted evaluation makes declassification fairly straightforward. The public and confidential facets are tied together in a single faceted

Figure 16: Declassification of Faceted Values

Declassification Rule

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

Downgrade Function

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

value, so declassification simply requires restructuring the faceted value to migrate information from one facet to another.

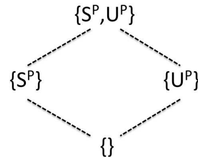
Providing a declassification operation with no restrictions invalidates most security guarantees. For instance, an attacker could declassify a user's password, or overwrite data that would be declassified later by legitimate code. In this manner, valid code intending to declassify the result of a password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

To provide more reliable security guarantees in the presence of declassification with faceted values, we pattern our approach after *robust declassification* [Zdancewic 2003; Myers et al. 2004], which guarantees that an active attacker, able to introduce code, is no more powerful than a passive attacker, who can only observe the results. An important topic for future work is to explore extending declassification to other policies, such as stateful declassification [Vanhoef et al. 2014].

This technique 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[11].

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

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



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

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.

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 16 performs the appropriate 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 16 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 7.1. *For any value V and view L:*

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

PROOF. See Appendix. □

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:

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

THEOREM 7.2 (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. □

As a result, noninterference also holds for these same views.

THEOREM 7.3 (TERMINATION INSENSITIVE NONINTERFERENCE WITH DECLASSIFICATION). *Suppose $L_P \neq \{\}$ for each P used in a declassification operation and*

$$\begin{array}{ll} & \Sigma_1 \sim_L \Sigma_2 \\ & \Sigma_1, e \Downarrow_\emptyset \Sigma_1', V_1 \\ & \Sigma_2, e \Downarrow_\emptyset \Sigma_2', V_2 \\ \textit{Then:} & \\ & \Sigma_1' \sim_L \Sigma_2' \\ & V_1 \sim_L V_2 \end{array}$$

Proof. Follows from Theorem 7.2 via a proof similar to Theorem 3.7.

8. INTROSPECTION OF FACETED VALUES

Determining whether a value is faceted might be a useful tool for a developer troubleshooting an application. One might consider introducing an `isFaceted` primitive in order to determine whether a value `v` is a faceted or a raw value.

```
isFaceted <k?42:0>; // evaluates to true
isFaceted 17        // evaluates to false
```

Correctly implementing introspection of faceted values is surprisingly complex and rife with pitfalls. Therefore, in this section we review a proposed solution, as well as some obvious approaches that do *not* work.

The rule for introspection of raw values is straightforward, as shown in the following rule [F-INTROSPECT-RAW]:

$$\frac{\Sigma, e \Downarrow_{pc} \Sigma_1, R}{\Sigma, \texttt{isFaceted}\ e \Downarrow_{pc} \Sigma', \texttt{false}} \quad [\textsc{f-introspect-raw}]$$

However, much more subtle issues arise when the result is faceted. A logical approach is to return `true`, as demonstrated in the following (broken) rule:

$$\frac{\Sigma, e \Downarrow_{pc} \Sigma_1, \langle k\ ?\ V_1 : V_2 \rangle}{\Sigma, \texttt{isFaceted}\ e \Downarrow_{pc} \Sigma', \texttt{true}} \quad [\textsc{f-introspect-broken1}]$$

While this approach seems reasonable, noninterference is not guaranteed. Though $\langle k\ ?\ \texttt{true} : \texttt{false} \rangle$ should be indistinguishable from `false` to an unauthorized viewer of k-sensitive data, `isFaceted` $\langle k\ ?\ \texttt{true} : \texttt{false} \rangle$ is distinguishable from `isFaceted false`.

Another (broken) approach would return `true` only to authorized viewers of k-sensitive data, as illustrated in the following rule:

$$\frac{\Sigma, e \Downarrow_{pc} \Sigma_1, \langle k\ ?\ V_1 : V_2 \rangle}{\Sigma, \texttt{isFaceted}\ e \Downarrow_{pc} \Sigma', \langle k\ ?\ \texttt{true} : \texttt{false} \rangle} \quad [\textsc{f-introspect-broken2}]$$

This rule works when only a single principal is involved, but fails for more complex lattices. With this strategy, `isFaceted` $\langle k\ ?\ 1 : \langle l\ ?\ 2 : 3 \rangle \rangle$ evaluates to $\langle k\ ?\ \texttt{true} : \texttt{false} \rangle$. However `isFaceted` $\langle l\ ?\ 2 : 3 \rangle$ evaluates to $\langle l\ ?\ \texttt{true} : \texttt{false} \rangle$, permitting an l-authorized viewer to distinguish between the two values and thereby learn k-sensitive information.

The correct approach must permit an observer to learn if a value is faceted on *any* of the principals the observer can see. The rule [F-INTROSPECT-FACETED] gives the proper solution.

$$\frac{\begin{array}{c}\Sigma, e \Downarrow_{pc} \Sigma_1, \langle k \ ?\ V_1 : V_2 \rangle \\ \Sigma_1, \texttt{isFaceted}\ V_2 \Downarrow_{pc} \Sigma', V'\end{array}}{\Sigma, \texttt{isFaceted}\ e \Downarrow_{pc} \Sigma', \langle k \ ?\ \texttt{true} : V' \rangle} \quad \text{[F-INTROSPECT-FACETED]}$$

Following this approach, isFaceted $\langle k\ ?\ 1 : \langle l\ ?\ 2 : 3\rangle\rangle$ evaluates to $\langle k\ ?\ \texttt{true} : \langle l\ ?\ \texttt{true} : \texttt{false}\rangle\rangle$ and isFaceted $\langle l\ ?\ 2 : 3\rangle$ evaluates to $\langle l\ ?\ \texttt{true} : \texttt{false}\rangle$, preventing k-sensitive information from leaking to l-authorized observers.

9. JAVASCRIPT IMPLEMENTATION IN FIREFOX

We incorporate our ideas for faceted evaluation into Firefox through the Narcissus JavaScript engine [Eich 2004] and the Zaphod Firefox plugin [Mozilla Labs Zaphod 2010]. The ZaphodFacets implementation [Austin 2011] extends the faceted semantics to handle the additional complexities of JavaScript. Our implementation is available online with some examples [Austin 2011], including the code from Figure 1. For ease of implementation, we do not include support for exception handling in this implementation. Instead, we halt execution if an exception might leak data.

We added two new primitives to the language. The makeFacetedValue(k,v1,v2) creates a new faceted value for the principal represented by the string k where v1 is the facet for authorized viewers of k and v2 is the facet observable to unauthorized viewers. For example, the following code sets x to a faceted value of $\langle k\ ?\ 42 : 0\rangle$.

```
var x = makeFacetedValue("k", 42, 0);
```

The second primitive is a getPublic function that extracts the public value of its input. For example, with the above code defining x, getPublic(x) returns 0. We use these two functions on all input/output boundaries of the system in order to appropriately label data as it comes in and to appropriately monitor data as it goes out. Section 9.1 details how we identify private data and Section 9.2 discusses which scripts we identify as untrusted.

9.1. 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.

9.2. Identifying Untrusted Scripts

An advantage of information flow techniques is that, while they are generally focused on enforcing confidentiality, they can also enforce integrity. In this section, we show how we use faceted values to track the influences of untrusted scripts; in this way, we can allow them to run but ensure that they will not modify sensitive fields.

Our policy treats only scripts coming from the same origin as the page to be trusted. This policy is not fool-proof; if the hosting 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, as we discuss in Section 9.4.

Any scripts loaded from a different origin are treated as untrusted, which we represent with a label of `"Untrusted"`. Effectively, each script is transformed into the following code:

```
var untrustedCode = makeFacetedValue("Untrusted",
  function() { /* original code here */ },
  function(){});
untrustedCode();
```

The untrusted code is wrapped in a function that is made into a faceted value. 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.

9.3. Faceted Values and the Document Object Model

The Document Object Model (DOM) presents interesting challenges for information flow analysis in general, and for faceted evaluation in particular.

The DOM is typically implemented in C or C++ code. As a result, the JavaScript engine may lose its ability to track a sensitive value once it has been written to the DOM without additional facilities in place. To illustrate the challenge, consider the following code:

```
var secret = makeFacetedValue("Confidential", 42, 0);
var title1 = document.getElementById("title1");
title1.setAttribute("class", secret);
var newVal = document.getElementsByTagName("h1")[0]
    .getAttribute("class");
```

Assuming no other private information has been written to the DOM, is it safe to release `newVal` to an external third party? It is not possible to tell without knowledge of the underlying structure of the webpage. If the first `h1` element on the page has an `id` attribute of `title1`, then `newVal = secret` and the value must be protected; otherwise the data may be released safely.

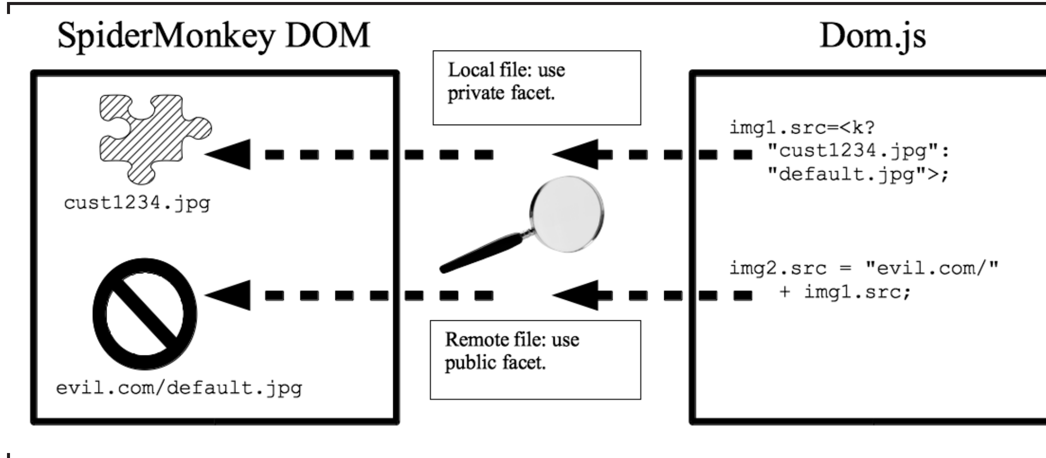
Without a more fine-grained knowledge of the DOM, monitor-based information flow systems must resort to more restrictive solutions such as forbidding any private data from being written to the DOM.

To resolve this issue, our implementation uses `dom.js` [Gal et al. 2011], an implementation of the DOM written in JavaScript. Because `dom.js` is written in JavaScript, we are able to persist faceted values in the DOM and correctly track the flow of private information. The underlying copy of the DOM is kept in sync with `dom.js` through the use of listeners and special hooks built into `dom.js`.

However, only a single facet will be rendered for the user to view. When a faceted value is written to the DOM, we must choose which facet to render. Our policy assumes that the user has the right to access all private data; therefore, the private facet is rendered to the DOM unless doing so would trigger network activity that could leak sensitive information. For instance, setting the `src` attribute of an `img` tag to `www.evil.com/4111111111111111.jpg` might leak the value of a user's credit card to `evil.com`'s server. ZaphodFacets uses the following policy:

(1) If a request is made for a resource from the hosting server, the private facet is used. (A more sophisticated policy might allow for certain origins to be whitelisted, and therefore permitted to see private facets as well.)

Figure 17: Illustration of Faceted Values and the DOM



(2) Otherwise, if the program counter is high, then the request for an image is suppressed.
(3) Finally, if the program counter is low, but the location of the resource is a faceted value, the low facet is selected.

Figure 17 shows the process in more detail. In this example, following a practice used by some banks to defend against phishing attacks, a user selected picture is displayed. An image is requested from a location determined by the faceted value $\langle confidential$? ”`cust1234.jpg`” : ”`default.jpg`”$\rangle$. Since the image is locally hosted, the image `cust1234.jpg` is loaded. However, malicious advertising code attempts to steal the image by requesting the same image from `evil.com`, which results in an image request for the faceted value $\langle confidential$? ”`www.evil.com/cust1234.jpg`” : ”`www.evil.com/default.jpg`”$\rangle$. But since the private facet refers to a different origin than the hosting website, the default image is rendered, and the attacker learns no information about the user's security picture.

Mashups present some interesting challenges, since we may wish to limit different libraries to different portions of the DOM. The policy we outline here does not currently address this issue; scripts loaded from different origins may modify the whole of the DOM. However, we note that the barrier between dom.js and the SpiderMonkey DOM provide us with an excellent place to apply *policy-agnostic programming* techniques [Yang et al. 2012], which would allow a developer to specify separate policy code to define these policies. Preliminary work [Austin et al. 2013] shows that faceted values combine well with policy-agnostic programming.

9.4. Cross-Site Scripting (XSS) Example

To illustrate how our controls can be useful for enforcing practical defenses, we consider an example of a webpage with an XSS vulnerability. Our controls do not prevent XSS attacks. Rather, they provide an additional layer of defense, reducing an attack's power.

In our 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 a server `evil.com`, controlled by the attacker.

The injected code is given below:

```
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);
}
```

The attack attempts to leak the password by loading an image from `evil.com`, encoding the password into the name of the requested image. However, incorporating the evasion technique mentioned in Section 9.3, it first writes the password to the `class` attribute of the `title1` element and then rereads it from the first `h1` element.

By our policy, the value of all password elements should be treated as confidential. Furthermore, any attempts to load files from a different origin should use the public facet; the server hosting the website, however, should see the private facet. Since we persist the different facets of `secret` to the DOM using dom.js, no security information is lost, and we do not lost track of the private code.

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.

While our example policy is far from complete, we use it to illustrate how our mechanism can enforce different information flow policies. A richer policy could specify a variety of fields and potential output channels. Furthermore, we imagine that browsers would wish to allow web developers to specify application-specific sensitive fields, such as credit card numbers, and allow users to protect information that they considered confidential (for instance, restricting the release of geolocation information).

9.5. Performance Results

Our approach is similar to prior work on secure multi-execution [Devriese and Piessens 2010]. To understand the performance tradeoff between these two approaches, we implemented 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 10.11.6. The machine had a 2.7 GHz Intel Core i7 processor with 4 cores and 16 GB of memory. For our benchmarks, we selected test cases from the SunSpider [Webkit.org 2011] benchmark suite. In all test cases, the default value used for SME is the same as the value used for the public facet in faceted evaluation.

— The **string-tagcloud** test case involves parsing JavaScript Object Notation (JSON). We modified this test to create 8 separate tag clouds from JSON-formatted strings. Our test cases involve 0 through 8 principals. In each case, every principal marks one element as confidential; additional inputs are public. For example, test 1 generates a tag cloud from 1 confidential input and 7 public inputs. Test 8 takes 8 confidential inputs, each marked as confidential by a distinct principal, and has no public inputs. The public facet of confidential data is set to a JSON string representing an empty array.
— The **md5-sparse** test case uses the crypto-md5 benchmark test from SunSpider. We modified this program to include 8 hashing operations with some inputs marked as con-

Figure 18: Faceted Evaluation vs. Secure Multi-Execution

Test case	# principals	Times in seconds		
		Secure multi-execution		Faceted execution
		sequential	concurrent	
string-tagcloud	0	43	43	56
	1	80	41	56
	2	149	45	57
	3	278	77	56
	4	512	143	58
	5	934	255	56
	6	1,694	444	56
	7	2,999	934	55
	8	*	2,097	58
md5-sparse	0	88	86	113
	1	166	86	117
	2	311	100	120
	3	566	151	126
	4	985	274	130
	5	1,806	508	134
	6	3,385	959	139
	7	*	1,883	142
	8	*	*	147
md5-interwoven	0	2	2	3
	1	4	2	9
	2	9	2	22
	3	18	4	52
	4	36	9	117
	5	72	17	257
	6	145	36	562
	7	289	79	1,214
	8	574	184	2,591

A result of "*" indicates a test that ran for more than one hour.

fidential. As with the string-tagcloud test, 0 to 8 of the inputs are marked as confidential to a different principal. Each hash input is the 15,824 character length sting used in the SunSpider test. When the data is marked as confidential, the public facet is set to an empty string.

— The **md5-interwoven** test case shows how the different evaluation strategies compare when the principals interact more heavily. In this case, the password 'secret' is hashed 50 times, and all hash inputs are marked as confidential to all principals. The public facets are set to '[redacted]', so that equal work is required for both the public and private facets. As in the previous tests, we increase the number of principals from 0 to 8.

The results in Figure 18 highlight the tradeoffs between the different approaches. The sequential variant of secure multi-execution has 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 in all of our test cases.

In the string-tagcloud test case, faceted evaluation's performance remains essentially flat as the number of principals increases. In part, this quality reflects our choice of public facets, which tend to require few computations. For instance, parsing a JSON string representing an empty array is trivial compared to parsing a large JSON string representing a tag cloud. While this setup favors faceted evaluation, we believe that public defaults are likely to be simple and hence easy to process.

The md5 cases illustrate the trade-offs between concurrent secure multi-execution and faceted evaluation. In both sets of tests, concurrent secure multi-execution outperforms faceted evaluation when the number of principals is small. However, in the md5-sparse case, SME must re-execute the same hash operations for different sets of principals, even if the operation involves no sensitive data. As a result, as the number of principals increases, faceted evaluation begins to outperform concurrent SME.

In contrast, the md5-interwoven set of tests represents the worst case for faceted evaluation. Faceted evaluation is not able to avoid any redundant work, since all hashes involve all principals. It has a more complex implementation and it is not able to run processes concurrently, so it is outperformed by both concurrent and sequential SME, regardless of the number of principals.

Our results suggest that faceted evaluation performs best when there is a more complex lattice of principals and when significant portions of the code are not security critical. Concurrent SME seems to be the better choice when the number of principals is small and when there is more interaction between different security principals.

10. RELATED WORK

Several researchers have discussed performing multiple executions to guarantee security properties. Capizzi et al. [2008] develop *shadow executions*, an approach similar to faceted values for use in securing information for desktop applications. They run both a public and a private copy of the application. 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. Devriese and Piessens [2010] extend this idea to JavaScript code with their *secure multi-execution* strategy, using a high and a low process to protect confidentiality in a similar manner. Our approach is similar in spirit, though we avoid overhead when code does not depend on confidential data.

Zanarini et al. [2013] note that secure multi-execution alters the behavior of programs violating noninterference, potentially introducing bugs that are difficult to analyze. Furthermore, the multiple processes might produce outputs to different channels in a different order than expected. They address these challenges through a *multi-execution monitor*. In essence, their approach executes the original program without modification and compares its results to the results of the SME processes; if output of secure multi-execution differs from the original at any point, a warning can be raised to note that the semantics have been altered.

De Groef et al. [2012] implement secure multi-execution in FlowFox, a fork of the Firefox web browser. They report a relatively modest performance overhead of 20% for their controls for a simple high/low lattice, and show the effectiveness of secure multi-execution in preventing many practical attacks. In addition, they apply their browser to the Alexa top-500 sites, showing that the secure multi-execution approach runs effectively on many major websites. Their work also does an excellent job of highlighting subtle implementation challenges involved in bringing secure multi-execution to the web browser. Rafnsson and Sabelfeld [2013] highlight some of the interesting challenges in making secure multi-execution a practical solution. Furthermore, they show how to handle declassification and to guarantee transparency.

Our semantics are closely related to work by Pottier and Simonet [2003]. While they prove noninterference statically for *Core ML*, their proof approach involves a Core ML^2 language that has expression pairs and value pairs, analogous to our faceted expressions and faceted values. Their work does provide evaluation rules for Core ML^2; while they are not intended as a dynamic enforcement mechanism, there is no reason that they could not be. Their evaluation rules are formatted in a small step semantics, which can supplement understanding of how faceted evaluation works.

Faceted evaluation is somewhat similar to symbolic execution [King 1976]. 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. Traditionally, symbolic execution works over several runs of a program, but there are some exceptions [Yang et al. 2012; Kolbitsch et al. 2011].

Despite the similarities, there are subtle differences between faceted evaluation and symbolic execution. 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 true branch will be executed twice, and the false branch will never be executed.

Yang et al. [2012] use symbolic execution as a runtime mechanism to guarantee noninterference (among other properties) in a manner similar to faceted evaluation. When symbolic values leave the system, they are *concretized* into a 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. This strategy has been combined with faceted values [Austin et al. 2013], allowing for more sophisticated policies to be specified without relying on a declassification primitive.

Kolbitsch et al. [2011] 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. The principal difference, besides the application, is that faceted values represent a lattice of different views of data, while Rozzle's symbolic heap values represent a range of possible values for different environments.

Several publications have explored exceptions in the context of information flow. In their study of information flow analysis for JavaScript, Hedin and Sabelfeld [2012] discuss safe exception handling for a dynamic information flow monitor. Their approach introduces a dynamic security label for exceptions. If this label is public, they forbid throwing exceptions in a secret context. In order to permit exceptions within a certain context (usually defined by a try-catch block), they extend the language to allow developers to raise the security label for exceptions.

Stefan et al. [2011] use a *labeled IO* (LIO) monad to guarantee information flow analysis. LIO tracks the current label of the execution, which serves as an upper bound on the labels of all data in lexical scope. It combines this notion with the concept of a *current clearance* that limits the maximum privileges allowed for an execution, thereby eliminating the termination channel. When an exception is thrown, LIO labels it with the label at the

point where the current exception is thrown. These exceptions can be caught provided that the label of the exception does not exceed the current clearance of the execution; otherwise the exception is re-thrown. Interestingly, the authors use this mechanism to recover from mechanism failures of their information flow monitor.

Hritcu et al. [2013] introduce *delayed exceptions* as a way to safely handle exceptions without violating noninterference. Delayed exceptions can be rethrown in a manner similar to LIO, but an alternative approach explored by the paper uses special *not-a-values* (NaVs) that behave in a manner similar to JavaScript's not-a-numbers (NaNs); any operations performed on a NaV return the original NaV. The authors further develop a prototype implementation using the NaV approach.

From the static analysis perspective, Myers [1999] includes a discussion on exceptions that was the basis for Jif's design. Likewise, Pottier and Simonet [2003] show how to handle exceptions for ML; their work became the basis for FlowCaml's exception handling. King et al. [2008] study false alarms caused by implicit flows. They argue that the bulk of these result from exceptions, and that safe handling of exceptions is not worth the hassle to developers. Askarov and Sabelfeld [2009a] argue uncaught exceptions are safe, as long as they cannot be caught during any execution of the program. Gampe and von Ronne [2011] provide noninterference in the context of method-not-found errors. Their type system guarantees that method-not-found errors can only happen if the error will not reveal any details about private data.

Yip et al. [2009] develop Resin, a server-side language runtime designed to enforce sophisticated security policies. Resin includes *filter objects*, which define boundaries of the system, and *policy objects*, which specify security policies for specific data. Together, these mechanisms allow for a rich, flexible system for specifying security policies. Resin tracks explicit flows of information and protects data integrity. Without support for implicit flows, its ability to protect confidentiality is more limited.

Other research has previously studied information flow analysis for JavaScript. Vogt et al. [2007], 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. Just et al. [2011] improve on this approach to better handle implicit flows through a hybrid dynamic and static analysis. Russo and Sabelfeld [2009] study timeout mechanisms and the channels that they enable. Russo et al. [2009] discuss dynamic tree structures, with obvious applications to the DOM. Bohannon et al. [2009] consider noninterference in JavaScript's reactive environment. Chugh et al. [2009] 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 [2009] 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. [2010] give an excellent overview of how JavaScript is used to circumvent privacy defense. Hedin and Sabelfeld [2012] develop dynamic information flow controls for a core of JavaScript that includes objects, higher-order functions, exceptions, and dynamic code evaluation; Hedin et al. [2015] extend this approach with static checks to provide a hybrid information flow control system for JavaScript. Kerschbaumer et al. [2013b] address many of the practical issues in making information flow analysis practical for JavaScript in a web browser, including outlining a variety of attacks where information flow analysis would be useful, and reviewing how their implementation is optimized to minimize the overhead of their controls. Further work by the same authors [Kerschbaumer et al. 2013a] shows how information flow analysis interacts with a JIT compiler, and offers a thorough overview of the performance overhead of different information flow controls. Rajani et al. [2015] imple-

ment information flow controls in the WebKit JavaScript engine, with careful coverage of both the DOM and event handling.

Information flow analysis largely traces its roots back to Denning [1976]. After her work, static systems dominated for a while, and type systems in particular. Volpano et al. [1996] codify Denning's approach as a type system, and also offer a proof of its soundness. Heintze and Riecke [1998] 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. Sabelfeld and Myers [2003] offer an extensive survey of other research on information flow.

The principal benefit of static analyses is that they can (generally) be performed before run time. In addition, they are generally superior than dynamic analyses in reasoning about alternate paths of execution. Type-based approaches to information flow analysis have a noticeable benefit in both speed and familiarity to developers, and have become the dominant approach. Myers [1999] discusses JFlow, a variant of Java with security types to provide strong information flow guarantees. JFlow was the basis for Jif [Jif 2010], a production-worthy language with information flow controls. No proof of JFlow's soundness was offered, largely due to the complexity of the language. Birgisson et al. [2011] show how capabilities can guarantee information flow policies. Flow-sensitive information flow analysis attempts to improve the precision of static analysis. Hunt and Sands [2006] describe a flow-sensitive type system. Hammer and Snelting [2009] use program dependence graphs to analyze JVM bytecode and guarantee TINI. Russo and Sabelfeld [2010] discuss the trade-offs between static and dynamic analyses in some depth.

Le Guernic et al. [2006] examine code from branches not taken, increasing precision at the expense of run-time performance overhead. Shroff et al. [2007] use a purely-dynamic analysis to track variable dependencies and reject more insecure programs over time.

Declassification is an important area of research for information flow analysis. Zdancewic [2003] uses integrity labels to provide *robust declassification*, permitting only high-integrity declassification decisions. Askarov and Myers [2010] consider a similar approach for endorsement, arguing that *checked endorsements* are needed to prevent an attacker from endorsing an unauthorized declassification. Chong and Myers [2004] use a framework for application-specific declassification policies. Askarov and Sabelfeld [2009b] study a declassification framework specifying what and where data is released. Vaughan and Chong [2011] infer declassification policies for Java programs. Declassification for secure multi-execution has been a subject of particular interest, due to the challenges of coordinating the high and low processes, with some solutions provided Rafnsson and Sabelfeld [2013] and by Vanhoef et al. [2014].

The termination channel is another area of particular concern for information flow analysis. Askarov et al. [2008] highlight complications of *intermediary output channels*, which allow an attacker to observe the output of a program during its execution. Rafnnson et al. [2011] buffer output to reduce data lost from intermediary output channels and termination behavior.

11. DISCUSSION

Information flow noninterference is a tricky security property to enforce via dynamic monitoring, since it is a *two-safety property*: noninterference can be refuted only by observing two executions (cmp. Theorem 3.7). Conversely, a *one-safety property* can be refuted by observing a single execution, and so one-safety properties are more amenable to dynamic enforcement. From this perspective, various prior techniques [Austin and Flanagan 2009; 2010] dynamically enforce a one-safety property that conservatively approximates the desired two-safety property of noninterference, but this conservative approximation introduces false alarms (that is, stuck executions even on some noninterfering implicit flows.

In contrast, faceted execution satisfies (termination-insensitive) semantics preservation for noninterfering programs (Theorem 3.9, and it also yields a projection property (Theorem 3.6), which is a one-safety property that suffices to prove the two-safety property of noninterference (Theorem 3.7).

ACKNOWLEDGMENTS

We thank the anonymous TOPLAS reviewers for their constructive feedback on this paper. We would also like to thank Martin Abadi for his feedback on an earlier draft of this paper, and Brendan Eich, Andreas Gal, and Dave Herman for valuable discussions on information flow analysis. Finally, we would like to thank David Flanagan and Donovan Preston for their help working with the dom.js project. This work was supported by NSF grant CNS-0905650.

REFERENCES

Askarov, A., Hunt, S., Sabelfeld, A., and Sands, D. 2008. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS '08*. Springer-Verlag, 333–348.

Askarov, A. and Myers, A. 2010. A semantic framework for declassification and endorsement. In *ESOP*. 64–84.

Askarov, A. and Sabelfeld, A. 2009a. Catch me if you can: permissive yet secure error handling. In *PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*. ACM, New York, NY, USA, 45–57.

Askarov, A. and Sabelfeld, A. 2009b. Tight enforcement of information-release policies for dynamic languages. In *IEEE Computer Security Foundations Symposium*. IEEE Computer Society, Washington, DC, USA, 43–59.

Austin, T. H. 2011. ZaphodFacetes github page. `https://github.com/taustin/ZaphodFacets`.

Austin, T. H. and Flanagan, C. 2009. Efficient purely-dynamic information flow analysis. In *Workshop on Programming Languages and Analysis for Security*.

Austin, T. H. and Flanagan, C. 2010. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*. ACM, 1–12.

Austin, T. H. and Flanagan, C. 2012. Multiple facets for dynamic information flow. See Field and Hicks [2012], 165–178.

Austin, T. H., Yang, J., Flanagan, C., and Solar-Lezama, A. 2013. Faceted execution of policy-agnostic programs. In *Workshop on Programming Languages and Analysis for Security*.

Banerjee, A. and Naumann, D. A. 2002. Secure information flow and pointer confinement in a Java-like language. In *IEEE Computer Security Foundations Workshop*. 253–267.

Bichhawat, A., Rajani, V., Garg, D., and Hammer, C. 2014. Generalizing permissive-upgrade in dynamic information flow analysis. In *Workshop on Programming Languages and Analysis for Security*.

Birgisson, A., Russo, A., and Sabelfeld, A. 2011. Capabilities for information flow. In *PLAS '11: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*. ACM.

Bohannon, A., Pierce, B. C., Sjöberg, V., Weirich, S., and Zdancewic, S. 2009. Reactive noninterference. In *ACM Conference on Computer and Communications Security*. 79–90.

Capizzi, R., Longo, A., Venkatakrishnan, V., and Sistla, A. 2008. Preventing information leaks through shadow executions. In *ACSAC*. 322 –331.

Chong, S. and Myers, A. C. 2004. Security policies for downgrading. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*. ACM, New York, NY, USA, 198–209.

Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. 2009. Staged information flow for JavaScript. In *Conference on Programming Language Design and Implementation*.

De Groef, W., Devriese, D., Nikiforakis, N., and Piessens, F. 2012. Flowfox: a web browser with flexible and precise information flow control. See Yu et al. [2012], 748–759.

Denning, D. E. 1976. A lattice model of secure information flow. *Communications of the ACM 19,* 5, 236–243.

Devriese, D. and Piessens, F. 2010. Noninterference through secure multi-execution. *Security and Privacy, IEEE Symposium on 0*, 109–124.

Dhawan, M. and Ganapathy, V. 2009. Analyzing information flow in JavaScript-based browser extensions. In *Annual Computer Security Applications Conference*.

EICH, B. 2004. Narcissus–JS implemented in JS. Available on the web at https://github.com/mozilla/narcissus/.

FENTON, J. S. 1974. Memoryless subsystems. *The Computer Journal 17,* 2, 143–147.

FIELD, J. AND HICKS, M., Eds. 2012. *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. ACM.

GAL, A., FLANAGAN, D., AND PRESTON, D. 2011. dom.js github page. https://github.com/andreasgal/dom.js, accessed October 2011.

GAMPE, A. AND VON RONNE, J. 2011. information flow control with errors.

GUERNIC, G. L., BANERJEE, A., JENSEN, T. P., AND SCHMIDT, D. A. 2006. Automata-based confidentiality monitoring. In *Asian Computing Science Conference on Secure Software*.

HAMMER, C. AND SNELTING, G. 2009. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*.

HEDIN, D., BELLO, L., AND SABELFELD, A. 2015. Value-sensitive hybrid information flow control for a javascript-like language. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*. IEEE, 351–365.

HEDIN, D. AND SABELFELD, A. 2012. Information-flow security for a core of JavaScript. In *Computer Security Foundations Symposium*.

HEINTZE, N. AND RIECKE, J. G. 1998. The SLam calculus: Programming with secrecy and integrity. In *Symposium on Principles of Programming Languages*.

HRITCU, C., GREENBERG, M., KAREL, B., PIERCE, B. C., AND MORRISETT, G. 2013. All your ifcexception are belong to us. In *IEEE Symposium on Security and Privacy*. 3–17.

HUNT, S. AND SANDS, D. 2006. On flow-sensitive security types. In *POPL*. 79–90.

JANG, D., JHALA, R., LERNER, S., AND SHACHAM, H. 2010. An empirical study of privacy-violating information flows in javascript web applications. In *ACM Conference on Computer and Communications Security*. 270–283.

Jif 2010. Jif homepage. http://www.cs.cornell.edu/jif/, accessed October 2010.

JUST, S., CLEARY, A., SHIRLEY, B., AND HAMMER, C. 2011. Information flow analysis for javascript. In *Programming language and systems technologies for internet clients*. ACM, New York, NY, USA, 9–18.

KASHYAP, V., WIEDERMANN, B., AND HARDEKOPF, B. 2011. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *IEEE Security and Privacy*.

KERSCHBAUMER, C., HENNIGAN, E., LARSEN, P., BRUNTHALER, S., AND FRANZ, M. 2013a. Information flow tracking meets just-in-time compilation. under submission.

KERSCHBAUMER, C., HENNIGAN, E., LARSEN, P., BRUNTHALER, S., AND FRANZ, M. 2013b. Towards precise and efficient information flow control in web browsers. In *Trust and Trustworthy Computing*. 187–195.

KING, D., HICKS, B., HICKS, M., AND JAEGER, T. 2008. Implicit flows: Can't live with 'em, can't live without 'em. In *International Conference on Information Systems Security*. 56–70.

KING, J. C. 1976. Symbolic execution and program testing. *Commun. ACM 19,* 7, 385–394.

KOLBITSCH, C., LIVSHITS, B., ZORN, B., AND SEIFERT, C. 2011. Rozzle: De-cloaking internet malware. Tech. Rep. MSR-TR-2011-94, Microsoft Research Technical Report.

MOORE, S., ASKAROV, A., AND CHONG, S. 2012. Precise enforcement of progress-sensitive security. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. ACM, 881–893.

Mozilla Labs Zaphod 2010. Mozilla labs: Zaphod add-on for the firefox browser. http://mozillalabs.com/zaphod, accessed October 2010.

MYERS, A. C. 1999. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*.

MYERS, A. C., SABELFELD, A., AND ZDANCEWIC, S. 2004. Enforcing robust declassification. In *IEEE Computer Security Foundations Workshop*. 172–186.

POTTIER, F. AND SIMONET, V. 2003. Information flow inference for ML. *Transactions on Programming Languages and Systems 25,* 1, 117–158.

RAFNSSON, W. AND SABELFELD, A. 2011. Limiting information leakage in event-based communication. In *PLAS '11: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*. ACM.

RAFNSSON, W. AND SABELFELD, A. 2013. Secure multi-execution: Fine-grained, declassification-aware, and transparent. In *IEEE Computer Security Foundations Symposium*. IEEE Computer Society.

Rajani, V., Bichhawat, A., Garg, D., and Hammer, C. 2015. Information flow control for event handling and the dom in web browsers. In *Computer Security Foundations Symposium (CSF), 2015 IEEE 28th*. 366–379.

Rinard, M., Cadar, C., Dumitran, D., Roy, D. M., Leu, T., and Beebee, W. S. 2004. Enhancing server availability and security through failure-oblivious computing. In *Symposium on Operating Systems Design and Implementation (OSDI)*. 303–316.

Russo, A. and Sabelfeld, A. 2009. Securing timeout instructions in web applications. In *IEEE Computer Security Foundations Symposium*.

Russo, A. and Sabelfeld, A. 2010. Dynamic vs. static flow-sensitive security analysis. In *IEEE Computer Security Foundations Symposium*. IEEE Computer Society.

Russo, A., Sabelfeld, A., and Chudnov, A. 2009. Tracking information flow in dynamic tree structures. In *ESORICS*. 86–103.

Sabelfeld, A. and Myers, A. C. 2003. Language-based information-flow security. *Journal on Selected Areas in Communications 21,* 1, 5–19.

Shroff, P., Smith, S. F., and Thober, M. 2007. Dynamic dependency monitoring to secure information flow. In *Computer Security Foundations Symposium*.

Stefan, D., Russo, A., Mitchell, J. C., and Mazières, D. 2011. Flexible dynamic information flow control in haskell. In *Proceedings of the 4th ACM Symposium on Haskell*. Haskell '11. ACM, New York, NY, USA, 95–106.

Vanhoef, M., De Groef, W., Devriese, D., Piessens, F., and Rezk, T. 2014. Stateful declassification policies for event-driven programs. In *Computer Security Foundations Symposium (CSF), 2014 IEEE 27th*. 293–307.

Vaughan, J. and Chong, S. 2011. Inference of expressive declassification policies. In *IEEE Security and Privacy*.

Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Krügel, C., and Vigna, G. 2007. Cross-site scripting prevention with dynamic data tainting and static analysis.

Volpano, D., Irvine, C., and Smith, G. 1996. A sound type system for secure flow analysis. *Journal of Computer Security 4,* 2-3, 167–187.

Webkit.org. 2011. SunSpider JavaScript benchmark. `http://www.webkit.org/perf/sunspider/sunspider.html`, accessed October 2011.

Yang, J., Yessenov, K., and Solar-Lezama, A. 2012. A language for automatically enforcing privacy policies. See Field and Hicks [2012], 85–96.

Yip, A., Wang, X., Zeldovich, N., and Kaashoek, M. F. 2009. Improving application security with data flow assertions. In *SOSP*, J. N. Matthews and T. E. Anderson, Eds. ACM, 291–304.

Yu, T., Danezis, G., and Gligor, V. D., Eds. 2012. *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. ACM.

Zanarini, D., Jaskelioff, M., and Russo, A. 2013. Precise enforcement of confidentiality for reactive systems. In *Computer Security Foundations Symposium*.

Zdancewic, S. 2003. A type system for robust declassification. In *19th Mathematical Foundations of Programming Semantics Conference*.

Zdancewic, S. A. 2002. Programming languages for information security. Ph.D. thesis, Cornell University.

APPENDIX

A. PROOFS

Lemma 3.5. *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) $\Sigma, (V_1\ V_2) \Downarrow_{pc}^{\mathsf{app}} \Sigma', V$

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

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

— For cases [F-VAL], [F-READ2], and [FA-⊥], $\Sigma = \Sigma'$. Therefore, $L(\Sigma) = L(\Sigma')$.
— Cases [F-DEREF], [F-APP], [F-LEFT], [F-RIGHT], [F-WRITE2], [FA-FUN], [FA-LEFT], and [FA-RIGHT] hold 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 3.2, $L(\Sigma'(a)) = \bot$. 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 3.4, $L(\Sigma) = L(\Sigma')$.
— For case [F-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 [F-WRITE1], $e = \texttt{write}(f, e')$. By the antecedents of this rule:

$$\begin{gathered} \Sigma, e' \Downarrow_{pc} \Sigma'', V \\ pc \text{ visible to } view(f) \\ L' = view(f) \\ v = L'(V) \\ \Sigma' = \Sigma''[f := \Sigma''(f).v] \end{gathered}$$

By induction, $L(\Sigma')(f) = L(\Sigma'')(f)$. Since pc is not visible to L, $L \neq L'$. Therefore, $L(\Sigma)(f) = L(\Sigma'')(f) = L(\Sigma')(f) = \epsilon$.

□

Theorem 3.6 (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. 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], $e = V$. Since $\Sigma, V \Downarrow_{pc} \Sigma, V$ and $L(\Sigma), L(V) \downarrow L(\Sigma), L(V)$, 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 3.2, $L(V'') = L(V')$. 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 3.3 $L(V) = L(\Sigma')(a)$. Therefore, by the [S-DEREF] rule, $L(\Sigma), L(\mathtt{!}e') \downarrow L(\Sigma'), L(V)$.
  — If $\bot = L(V')$, then by Lemma 3.3 $L(V) = \bot$. Therefore, by the [S-DEREF] rule, $L(\Sigma), L(\mathtt{!}e') \downarrow L(\Sigma'), L(V)$.
— For case [F-ASSIGN], $e = (e_a \mathtt{:=} e_b)$. 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 3.4, $\forall a' \neq a, L(\Sigma')(a') = L(\Sigma_2)(a')$. Also by Lemma 3.4 $L(\Sigma')(a) = L(V)$. Therefore, by the [S-ASSIGN] rule, $L(\Sigma), L(e_a \mathtt{:=} e_b) \downarrow L(\Sigma'), L(V)$.
  — If $\bot = L(V_1)$, then by Lemma 3.4 $L(\Sigma') = L(\Sigma_2)$. 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:

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

By induction

$$L(\Sigma), L(e_a) \downarrow L(\Sigma_1), L(V_1)$$
$$L(\Sigma_1), L(e_b) \downarrow L(\Sigma_2), L(V_2)$$

V_1 must be a function, the bottom value ($\perp$), or a faceted value where all the nodes are functions or $\perp$.

— If $(\lambda x.e') = L(V_1)$, then it holds by induction that $L(\Sigma_2), e'[x := V_2] \downarrow L(\Sigma'), L(V)$. Therefore, by the [S-APP] rule, $L(\Sigma), L(e_a\ e_b) \downarrow L(\Sigma'), L(V)$.

— Otherwise, $\perp = L(V_1)$. By Lemma 3.5 and the [F-APP-$\perp$] rule, it follows that $L(\Sigma') = L(\Sigma_2)$ and $L(V) = \perp$. Therefore $L(\Sigma_2), L(e_a\ e_b) \downarrow L(\Sigma'), L(V)$ by the [S-APP-$\perp$] 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 3.5 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 3.5 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,

$$\Sigma(f) = v.w$$
$$L' = view(f)$$
$$pc \text{ visible to } L'$$
$$pc' = L' \cup \{\overline{k} \mid k \notin L'\}$$
$$\Sigma' = \Sigma[f := w]$$
$$V = \langle\!\langle\, pc'\ ?\ v : \perp \,\rangle\!\rangle$$

— If $L = view(f)$, then $L(V) = v$. This case holds since $L(\Sigma), \texttt{read}(f) \downarrow L(\Sigma'), v$.

— Otherwise, $L \neq view(f)$. Therefore $L(\Sigma) = L(\Sigma')$ since $L(\Sigma(f)) = \epsilon$. Also, $L(e) = \perp$ and $L(V) = \perp$. Therefore, this case holds since $L(\Sigma), \perp \downarrow L(\Sigma), \perp$.

— For [F-READ2], $e = \texttt{read}(f)$. By the antecedent of this rule, pc not visible to $view(f)$. Therefore, $L(e) = \perp$. Since $\Sigma, \texttt{read}(f) \Downarrow_{pc} \Sigma, \perp$ and $L(\Sigma), \perp \downarrow L(\Sigma), \perp$, this case holds.

— For [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, $L(\Sigma), e' \downarrow L(\Sigma''), L(V)$.

— 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,

$$\Sigma, e' \Downarrow\!\!\Downarrow_{pc} \Sigma', V$$
$$pc \text{ not visible to } view(f)$$

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

$$k \notin pc, \overline{k} \notin pc$$
$$\Sigma, (V_a\ V_2) \Downarrow\!\!\Downarrow^{\texttt{app}}_{pc \cup \{k\}} \Sigma_1, V'_a$$
$$\Sigma_1, (V_b\ V_2) \Downarrow\!\!\Downarrow^{\texttt{app}}_{pc \cup \{\overline{k}\}} \Sigma', V'_b$$

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 3.5, $\mathrm{L}(\Sigma_1) = L(\Sigma')$.

— 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 3.5, $L(\Sigma) = L(\Sigma_1)$. 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.

□

Theorem 4.1 (Faceted Evaluation Generalizes NSU Evaluation).
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], $e = V$. This case then holds since $\Sigma, V \Downarrow_{pc} \Sigma, V$ and $\Sigma, V \Downarrow\!\!\Downarrow_{pc} \Sigma, V$.

— case [NSU-APP]. Then $e = (e_a\ e_b)$. By the antecedents of this rule:

$$\Sigma, e_a \Downarrow_{pc} \Sigma_1, (\lambda x.e')$$
$$\Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V'$$
$$\Sigma_2, e'[x := V'] \Downarrow_{pc} \Sigma', V$$

By induction:

$$\Sigma, e_a \Downarrow\!\!\Downarrow_{pc} \Sigma_1, (\lambda x.e')$$
$$\Sigma_1, e_b \Downarrow\!\!\Downarrow_{pc} \Sigma_2, V'$$

Therefore, by the [F-APP] rule it is sufficient to show that $\Sigma_2, ((\lambda x.e')\ V') \Downarrow\!\!\Downarrow^{\texttt{app}}_{pc} \Sigma', V$. 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 $\Sigma', (\bot\ V') \Downarrow^{\mathsf{app}}_{pc} \Sigma', \bot$, 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 $\Sigma_2, (\langle k\ ?\ (\lambda x.e')\ V' : \bot\rangle) \Downarrow^{\mathsf{app}}_{pc} \Sigma', V$.

— If $k \in pc$, then $\Sigma_2, (\langle k\ ?\ (\lambda x.e')\ V' : \bot\rangle) \Downarrow^{\mathsf{app}}_{pc} \Sigma'', V''$ 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:

$$\Sigma_2, ((\lambda x.e')\ V') \Downarrow^{\mathsf{app}}_{pc} \Sigma_3, V_3$$
$$\Sigma_3, (\bot\ V') \Downarrow^{\mathsf{app}}_{pc} \Sigma_4, V_4$$
$$V_5 = \langle\!\langle\, k\ ?\ V_3 : V_4 \,\rangle\!\rangle$$

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:

$$\Sigma, e' \Downarrow_{k\cup\{pc\}} \Sigma', V'$$
$$V = \langle k\rangle^{pc} V'$$

By induction, $\Sigma, e' \Downarrow_{pc\cup\{k\}} \Sigma', V'$.

— If $k \in pc$, then $pc \cup \{k\} = pc$ and $V = V'$. Therefore, by the [F-LEFT] rule, $\Sigma, \langle k\ ?\ e' : \bot\rangle \Downarrow_{pc} \Sigma', V$.

— Otherwise, $k \notin pc$ and $\overline{k} \notin pc$. Therefore $V = \langle k\ ?\ V' : \bot\rangle$. By the [F-VAL] rule, $\Sigma', \bot \Downarrow_{pc\cup\{\overline{k}\}} \Sigma', \bot$. Therefore, $\Sigma, \langle k\ ?\ e' : \bot\rangle \Downarrow_{pc\cup\{k\}} \Sigma', V$ by the [F-SPLIT] rule.

— case [NSU-REF]. Then $e = \texttt{ref}\ e'$. By the antecedents of this rule:

$$\Sigma, e' \Downarrow_{pc} \Sigma_1, V'$$
$$a \notin dom(\Sigma_1)$$
$$\Sigma' = \Sigma_1[a := \langle\!\langle\, pc\ ?\ V' : \bot \,\rangle\!\rangle]$$

By induction, $\Sigma, e' \Downarrow_{pc} \Sigma_1, a$. Without loss of generality, we assume that both executions allocate the same address a. Therefore, $\Sigma, \texttt{ref}\ e' \Downarrow_{pc} \Sigma', a$ by the [F-REF] rule.

— Case [NSU-DEREF]. Then $e = \texttt{!}e'$. By the antecedents of this rule:

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

By induction, $\Sigma, e' \Downarrow_{pc} \Sigma', a$. Therefore $\Sigma, \texttt{!}e' \Downarrow_{pc} \Sigma', V$ by the [DEREF] rule.

— case [NSU-ASSIGN]. Then $e = e_a \texttt{:=} e_b$. By the antecedents of this rule:

$$\begin{array}{c} \Sigma, e_a \Downarrow_{pc} \Sigma_1, a \\ \Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V \\ pc = \texttt{label}(\Sigma_2(a)) \\ V' = \langle\langle\, pc \ ?\ V : \bot \,\rangle\rangle \\ \Sigma' = \Sigma_2[a := V'] \end{array}$$

By induction:

$$\begin{array}{c} \Sigma, e_a \Downarrow\!\!\Downarrow_{pc} \Sigma_1, a \\ \Sigma_1, e_b \Downarrow\!\!\Downarrow_{pc} \Sigma_2, V \end{array}$$

— 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:

$$\begin{array}{c} \Sigma, e_a \Downarrow_{pc} \Sigma_1, \bot \\ \Sigma_1, e_b \Downarrow_{pc} \Sigma', V \end{array}$$

By induction:

$$\begin{array}{c} \Sigma, e_a \Downarrow\!\!\Downarrow_{pc} \Sigma_1, \bot \\ \Sigma_1, e_b \Downarrow\!\!\Downarrow_{pc} \Sigma', V \end{array}$$

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:

$$\begin{array}{c} \Sigma, e_a \Downarrow_{pc} \Sigma_1, \langle k \ ?\ a : \bot\rangle \\ \Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V \\ pc \cup \{k\} = \texttt{label}(\Sigma_2(a)) \\ V' = \langle\langle\, pc \cup \{k\} \ ?\ V : \bot \,\rangle\rangle \\ \Sigma' = \Sigma_2[a := V'] \end{array}$$

By induction:

$$\begin{array}{c} \Sigma, e_a \Downarrow\!\!\Downarrow_{pc} \Sigma_1, \langle k \ ?\ a : \bot\rangle \\ \Sigma_1, e_b \Downarrow\!\!\Downarrow_{pc} \Sigma_2, V \end{array}$$

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, $\Sigma, e_a \texttt{:=} e_b \Downarrow\!\!\Downarrow_{pc} \Sigma', V$ by the [F-ASSIGN] rule.

□

Theorem 4.2 (Faceted Evaluation Generalizes PU Evaluation).
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.

— Cases [NSU-VAL] [NSU-APP], [NSU-APP-⊥], [NSU-APP-K], [NSU-LABEL], [NSU-REF], and [NSU-DEREF], hold by the same argument as in the proof for Theorem 4.1.
— Case [PU-ASSIGN]. Then $e = e_a \texttt{:=} e_b$. By the antecedents of this rule:

$$\begin{array}{c} \Sigma, e_a \Downarrow_{pc} \Sigma_1, a \\ \Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V \\ V' = \langle\langle\, pc \ ?\ V : \Sigma_2(a) \,\rangle\rangle \\ \Sigma' = \Sigma_2[a := V'] \end{array}$$

By induction:

$$\Sigma, e_a \Downarrow_{pc} \Sigma_1, a$$
$$\Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V$$

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 $\Sigma, e_a \texttt{:=} e_b \Downarrow_{pc} \Sigma', V$ by the [F-ASSIGN] rule.

— case [PU-ASSIGN-K]. Then $e = e_a \texttt{:=} e_b$. 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:

$$\Sigma, e_a \Downarrow_{pc} \Sigma_1, \langle k \; ? \; a : \bot \rangle$$
$$\Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V$$

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 $\Sigma, e_a \texttt{:=} e_b \Downarrow_{pc} \Sigma', V$ by the [F-ASSIGN] rule.

□

Lemma 5.7. *Suppose pc is not visible to L and that*

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

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) $\Sigma, (B_1 \; B_2) \Downarrow_{pc}^{\texttt{app}} \Sigma', B$ or
(3) $\Sigma, e \Downarrow_{pc}^{B'} \Sigma', B$ or
(4) $\Sigma, B' \; \texttt{catch} \; e \Downarrow_{pc}^{\texttt{catch}} \Sigma', B$

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

The proof is by induction on the derivation of $\Sigma, e \downarrow_{pc} \Sigma', B$, the derivation of $\Sigma, (B_1 \; B_2) \downarrow\downarrow_{pc}^{\texttt{app}} \Sigma', B$, the derivation of $\Sigma, e \Downarrow_{pc}^{B'} \Sigma', B$, the derivation of $\Sigma, B' \; \texttt{catch} \; e \Downarrow_{pc}^{\texttt{catch}} \Sigma', B$, 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-LEFT1], [FA-LEFT2], [FA-RIGHT1], [FA-RIGHT2], [FE-DEREF], [FE-APP], [FE-TRY], [FE-WRITE2], [FB-NORMAL], and [FX-CATCH] hold by induction.
— For cases [FE-SPLIT], [FA-SPLIT1], [FA-SPLIT2], [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], $e = \texttt{ref} \; e'$. By the antecedents of this rule:

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

By induction, $L(\Sigma) = L(\Sigma'')$. Therefore, $\forall a'$ where $a' \neq a$, $L(\Sigma)(a') = L(\Sigma')(a')$. By Lemma 5.4, $L(\Sigma'(a)) = \bot$. Since $a \notin dom(\Sigma)$, $\Sigma(a) = \bot$. Therefore $L(\Sigma) = L(\Sigma')$.

— For case [FE-ASSIGN], $e = e_a \texttt{:=} e_b$. By the antecedents of the [FE-ASSIGN] rule:

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

By induction, $L(\Sigma) = L(\Sigma_1) = L(\Sigma_2)$. Therefore by Lemma 5.6, $L(\Sigma) = L(\Sigma')$.

— For case [FE-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 [FE-WRITE1], $e = \texttt{write}(f, e')$. By the antecedents of this rule:

$$\Sigma, e' \Downarrow_{pc} \Sigma'', B$$
$$pc \text{ visible to } view(f)$$
$$L' = view(f)$$
$$v = L'(B)$$
$$\Sigma' = \Sigma''[f := \Sigma''(f).v]$$

By induction, $L(\Sigma')(f) = L(\Sigma'')(f)$. Since pc is not visible to L, $L \neq L'$. Therefore, $L(\Sigma)(f) = L(\Sigma'')(f) = L(\Sigma')(f) = \epsilon$.

□

Theorem 5.9 (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. 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

$$\Sigma, (B_1\ B_2) \Downarrow_{pc}^{\mathsf{app}} \Sigma', B$$
$$L(B_1) = (\lambda x.e')$$
$$L(B_2) \neq \texttt{raise}$$

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 $\Sigma, B'\ \texttt{catch}\ e \Downarrow_{pc}^{\mathsf{catch}} \Sigma', B$ 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 $\Sigma, (V_1\ V_2) \Downarrow_{pc}^{\mathsf{app}} \Sigma', V$, 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-LEFT1], [FA-RIGHT1], [FA-FUN1], [FA-SPLIT1], and [FA-$\bot$] hold by similar arguments as in the proof for Theorem 3.6.

— Cases [FB-SPLIT], [FX-SPLIT], and [FA-SPLIT2] hold by a similar argument as the one used for [FA-SPLIT] in Theorem 3.6.

— 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 5.8 $L(B) = \bot$ and $L(V') = \texttt{raise}$. By Lemma 5.4, $L(V) = L(V')$. $\forall a'$ where $a' \neq a$, $L(\Sigma')(a') = L(\Sigma'')(a')$. By Lemma 5.4, $L(\Sigma'(a)) = \bot$. 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 5.4, $L(V) = L(V')$. 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 5.5 $L(B) = L(\Sigma')(a)$. Therefore, by the [S-DEREF] rule, $L(\Sigma), L(\texttt{!}e') \downarrow L(\Sigma'), L(B)$.

— If $\bot = L(V')$, then by Lemma 5.5 $L(V) = \bot$. Therefore, by the [S-DEREF] rule, $L(\Sigma), L(\texttt{!}e') \downarrow L(\Sigma'), L(B)$.

— If $\texttt{raise} = L(V')$, then by Lemma 5.5 $L(V) = \texttt{raise}$. 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, raise, or $\bot$.

— If $L(B_1)$ is an address and $L(B) \neq \texttt{raise}$, then $a = L(B_1)$. By Lemma 5.6, $\forall a' \neq a, L(\Sigma')(a') = L(\Sigma_2)(a')$. Also by Lemma 5.6, $L(\Sigma')(a) = L(B)$. 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 5.6 $L(\Sigma') = L(\Sigma_2)$. Therefore, this case holds by the [S-ASSIGN-EXN2] rule.

— If $L(B_1) = \bot$, then by Lemma 5.6 $L(\Sigma') = L(\Sigma_2)$. Therefore, this case holds by the [S-ASSIGN-$\bot$] rule.

— Otherwise $L(B_1) = \texttt{raise}$. By Lemma 5.1, $L(\Sigma_2) = L(\Sigma_1)$ and $B = \texttt{raise}$. By Lemma 5.6, $L(\Sigma') = L(\Sigma_2)$. 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:

$$\Sigma, e_a \Downarrow_{pc} \Sigma_1, B_1$$
$$\Sigma_1, e_b \Downarrow_{pc} \Sigma_2, B_2$$
$$\Sigma_2, (B_1\ B_2) \Downarrow_{pc}^{\texttt{app}} \Sigma', B$$

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:

$$L(\Sigma_1), L(e_b) \downarrow L(\Sigma_2), L(B_2)$$
$$L(\Sigma_2), e'[x := L(B_2)] \downarrow L(\Sigma'), L(B)$$

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 5.2, $L(\Sigma_2) = L(\Sigma')$ and $L(B) = \texttt{raise}$. Therefore this case holds by the [S-APP-EXN2] rule.

— If $L(B_1) = \texttt{raise}$, then by Lemma 5.1, $L(\Sigma_1) = L(\Sigma_2)$ and $L(B_2) = \texttt{raise}$. By Lemma 5.2, $L(\Sigma_2) = L(\Sigma')$ 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 5.7 and the [FE-APP-$\bot$] 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-$\bot$] rule.

— For case [FE-TRY], $e = e_1\ \texttt{catch}\ e_2$. By the antecedents of this rule

$$\Sigma, e_1 \Downarrow_{pc} \Sigma_1, B_1$$
$$\Sigma_1, B_1\ \texttt{catch}\ e_2 \Downarrow_{pc}^{\texttt{catch}} \Sigma', B$$

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 5.3, $L(\Sigma') = L(\Sigma_1)$ 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,

$$\Sigma, e' \Downarrow_{pc} \Sigma', B$$
$$L = \mathit{view}(f)$$
$$pc \text{ not visible to } L \text{ or } L(B) = \texttt{raise}$$

— 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], [FX-CATCH], [FA-LEFT2], and [FA-RIGHT2] hold by induction.
— Cases [FB-RAISE], [FA-RAISE1], [FA-RAISE2], and [FX-NOERR] are vacuosly true.

□

Lemma 7.1. *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 $\texttt{U}^P \in L$ and $\texttt{S}^P \notin L$, then $L(V) = L(V_2) = L(V')$.
  — Otherwise $\texttt{U}^P \notin L$ and $\texttt{S}^P \notin L$. Then $L'(V) = L(V_1) = L(V')$.
— Case $V = \langle \texttt{U}^P \ ?\ V_1 : V_2 \rangle$. Let $V' = \texttt{downgrade}_P(V) = \langle\!\langle\, \texttt{U}^P \ ?\ V_1 : \texttt{downgrade}_P(V_2) \,\rangle\!\rangle$.
  — If $\texttt{U}^P \in L$, 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.

□

Theorem 7.2 (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. 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 3.6.
— For case [DECLASSIFY], $e = \texttt{declassify}_P\ e'$. Then by the antecedents of this rule:

$$\Sigma, e' \Downarrow_{pc} \Sigma', V' \qquad \texttt{U}^P \notin pc \qquad V = \texttt{downgrade}_P(V')$$

  By induction:

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

  By Lemma 7.1, $L(\texttt{downgrade}_P(V')) = L(V') = L(V)$.

□