Multiple Facets for Dynamic Information Flow

Thomas H. Austin Cormac Flanagan
taustin@uscs.edu cormac@ucsc.edu
University of California, Santa Cruz

Abstract

JavaScript has become a central technology of the web, but it is also the source of many security problems, including cross-site scripting attacks and malicious advertising code. Central to these problems is the fact that code from untrusted sources runs with full privileges. We implement *information flow controls* in Firefox to help prevent violations of data confidentiality and integrity.

Most previous information flow techniques have primarily relied on either static type systems, which are a poor fit for JavaScript, or on dynamic analyses that sometimes get stuck due to problematic implicit flows, even in situations where the target web application correctly satisfies the desired security policy.

We introduce *faceted values*, a new mechanism for providing information flow security in a dynamic manner that overcomes these limitations. Taking inspiration from secure multi-execution, we use faceted values to simultaneously and efficiently simulate multiple executions for different security levels, thus providing non-interference with minimal overhead, and without the reliance on the stuck executions of prior dynamic approaches.

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

Keywords Information flow control, dynamic analysis, JavaScript, web security

1. Introduction

JavaScript has helped to usher in a new age of richly interactive web applications. Often times, developers build these sites by including JavaScript code from a number of different sources. 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.

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.

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 systematic solution to many of these security challenges, but to date it has not achieved its potential, largely because much research on static information flow type systems is an awkward fit for dynamically typed JavaScript code. Additionally, there has been a folklore that dynamic information flow analysis is not sound in the presence of implicit flows.

This folklore is not true, however, and proposed mechanisms for dealing with implicit flows include the *no-sensitive-upgrade* semantics [39, 5] and the *permissive-upgrade* semantics [6]. Both semantics guarantee the key correctness property of termination-insensitive non-interference (TINI), which states that private inputs do not influence public outputs. (Private information can influence termination, but this channel is limited to a brute force attack [1]).

Despite this correctness guarantee, neither semantics provides an ideal foundation for JavaScript security since both 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 a 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.

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 [9, 13]. This *multi-process execution* cleanly guarantees non-interference 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 combine the benefits of multi-process execution with the efficiency of single-process execution. 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, for most data, the two raw values in a faceted value are identical, in which case we collapse the two simulated executions on identical data into a single execution, drastically reducing the overhead. In the presence of multiple principals and a complex security lattice, a faceted value can contain many raw values, rather than just two. In this situation, the semantics of tracking information flow is a little more complex, costing some run-time performance overhead. However, our experimental results suggest that faceted

evaluation outperforms multi-process execution as the number of principals increases.

This paper includes a formal description of the faceted value approach to dynamic information flow analysis, and a proof that it achieves termination-insensitive non-interference. We also 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 have implemented this mechanism inside the Firefox browser (using the Zaphod plug-in [26]) in order to validate its utility in a web browsing context. Additionally, we have used this implementation to compare the performance of faceted values against multi-process execution. Finally, we discuss declassification and how it relates to faceted values, noting this feature as an additional point of distinction with multi-process execution.

1.1 Overview of Faceted Evaluation

To motivate the need for 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 this paper is that the correct value for `y` after this assignment depends on the authority of the observer. For example, suppose initially that $\texttt{x} = \texttt{true}$ and $\texttt{y} = \texttt{false}$, and that `x` is secret whereas `y` is public. Then after this assignment:

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

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

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

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

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 value representation naturally generalizes the traditional 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. 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 by earlier dynamic information flow analyses, these two security labels are insufficient to avoid stuck executions in the presence of implicit flows. As illustrated by the conditional assignment considered above, correct handling of implicit flows requires the introduction of more general notion of faceted values $\langle k \ ? \ V_{\text{H}} : V_{\text{L}} \rangle$, in which the public facet V_{L} is a real value and not simply $\bot$. In particular, the post-assignment value for `y` is cleanly represented as the faceted value $\langle k \ ? \ \texttt{true} : \texttt{false} \rangle$ that captures `y`'s appearance to both public and private observers.

Based on this faceted value representation, 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 is designed to avoid 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 exactly like $C[V_{\text{H}}]$ from the perspective of a private observer, and behaves exactly like $C[V_{\text{L}}]$ to a public observer (under a termination-insensitive notion of equivalence). This *projection property* means that a single faceted computation simulates multiple non-faceted computations, one for each element in the security lattice. This projection property also enables an elegant proof of termination-insensitive non-interference, shown in Section 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

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

evaluates to the nested faceted value

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

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

As a second example, the expression

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

evaluates to the result

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

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

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

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

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

1.2 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 [16]. Here, the function `f(x)` returns the value of its boolean 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 $\texttt{false}^H$ and $\texttt{true}^H$) 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

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

Figure 1: A JavaScript Function with Implicit Flows

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

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 $\langle k \ ? \ \texttt{false} : \bot \rangle$ to reflect that this value depends on private inputs. Unfortunately, this approach is not sound, since it loses the critical information that a public observer should still see $\texttt{y} = \texttt{true}$. 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* check [39, 5], 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.

[2] The original paper [6] used the $\texttt{false}^P$ to represent $\langle k \ ? \ \texttt{false} : * \rangle$, where the superscript P denotes "partially leaked".

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 $\texttt{f}(\bot)$ 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.

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

The expression $\langle k \ ? \ e_1 : e_2 \rangle$ creates a faceted values 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 *principals* as synonyms and focus primarily on confidentiality—Section 5 later introduces integrity labels in the context of robust 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.

Figure 2: The source language λ^{facet}

Syntax:

$e ::=$		*Term*
	x	variable
	c	constant
	$\lambda x.e$	abstraction
	$e_1\ e_2$	application
	$\texttt{ref}\ e$	reference allocation
	$\texttt{!}e$	dereference
	$e\texttt{:=}e$	assignment
	$\texttt{read}(f)$	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:

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

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 3. A closure $(\lambda x.e, \theta)$ is a pair of a λ-expression and a substitution θ that maps variables to values. 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.

We formalize the standard semantics via a big-step relation

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

that evaluates an expression e in the context of a store σ and a substitution θ, and which returns the resulting value v and the (possibly modified) store σ'. This relation is defined via the evaluation rules shown in Figure 3, which are mostly straightforward. For example, the rule [S-APP] evaluates the body of the called function, where the notation $\theta[x := v]$ denotes the substitution that is identical to θ except that it maps x to v.

The only unusual aspect of this semantics concern 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 3: Standard Semantics

Runtime Syntax

$$\begin{array}{lllll}
a & \in & \mathit{Address} & & \\
\sigma & \in & \mathit{store} & = & (\mathit{Address} \to_p \mathit{value} \\
 & & & & \quad \cup\ \mathit{File} \to \mathit{value}^*) \\
\theta & \in & \mathit{subst} & = & \mathit{Var} \to_p \mathit{value} \\
v & \in & \mathit{value} & ::= & c \mid a \mid (\lambda x.e, \theta) \mid \bot \\
w & \in & \mathit{value}^* & &
\end{array}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{}{\sigma, \theta, \bot \downarrow \sigma, \bot} \quad \text{[S-BOT]}$$

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 which provide noninterference guarantees.

Figure 4 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 ($V \in \mathit{Value}$, $\Sigma \in \mathit{Store}$, $\Theta \in \mathit{Subst}$) from those of the standard semantics ($v \in \mathit{value}$, $\sigma \in \mathit{store}$, $\theta \in \mathit{subst}$).

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 the value 0. Often, the public facet is set to $\bot$ to denote that there is no intended publicly visible facet. Implicit flows introduce public facets other than $\bot$.

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

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

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

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

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

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

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

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

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

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 $\langle k \ ? \ V_{\text{H}} \ : \ V_{\text{L}} \rangle$, 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 $\bot$ can be applied as a function and returns $\bot$ via [FA-BOT] (much like the earlier [S-APP-BOT] rule).

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

The operand of a dereference operation $(\texttt{!}e)$ may also be a faceted value tree. In this case, the rule [F-REF] uses the helper function $\texttt{deref}(\Sigma, V_a, pc)$ 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 $\mathit{assign}(\Sigma, pc, V_a, V)$ to decompose V_a into appropriate addresses and to update the store Σ at those locations with V, while ensuring that each update is only visible to appropriate principals that are consistent with pc, to avoid information leaks via implicit flows.

The faceted semantics of I/O operations introduces some additional complexities since it involves communication with external, non-faceted files. Each file f has an associated view $\mathit{view}(f) = \{k_1, \ldots, k_n\}$ 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 $\texttt{read}(f)$ may be executed multiple times with different pc labels. Of these multiple executions, only the single execution where pc is visible to $\mathit{view}(f)$ actually reads from the file via [F-READ]; all other executions are no-ops via [F-READ-IGNORE]. The non-faceted value v read from the file is converted to a faceted value $\langle\!\langle \, pc' \ ? \ v \ : \ \bot \, \rangle\!\rangle$ that is only visible to $\mathit{view}(f)$, where pc' is the program counter representation of that view.

An output $\texttt{write}(f, e)$ behaves in a similar manner, so only one execution writes to the file via the rule [F-WRITE]. This rule uses the projection operation $v = L(V)$ where $L = \mathit{view}(f)$ to project the faceted value V produced by e into a corresponding non-faceted value v that is actually 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 4. To provide helpful intuition, however, Figure 5 sketches alternative direct rules for evaluating a conditional test $\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$. In particular, if e_1 evaluates to a faceted value $\langle k \ ? \ V_H \ : \ V_L \rangle$, the if statement is evaluated potentially twice, using facets V_H and V_L as the conditional test by the [F-IF-SPLIT] rule.

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

Figure 4: Faceted Evaluation Semantics

Runtime Syntax

$$
\begin{array}{rcll}
\Sigma & \in & \mathit{Store} & = & (\mathit{Address} \rightarrow_p \mathit{Value}) \cup (\mathit{File} \rightarrow \mathit{Value}^*) \\
\Theta & \in & \mathit{Subst} & = & \mathit{Var} \rightarrow_p \mathit{Value} \\
R & \in & \mathit{RawValue} & ::= & c \mid a \mid (\lambda x.e, \Theta) \mid \bot \\
V & \in & \mathit{Value} & ::= & R \mid \langle k \ ?\ V_1 : V_2 \rangle \\
h & \in & \mathit{Branch} & ::= & k \mid \overline{k} \\
pc & \in & PC & = & 2^{\mathit{Branch}}
\end{array}
$$

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

$$\frac{}{\Sigma, \Theta, c \Downarrow_{pc} \Sigma, c} \quad \text{[F-CONST]}$$

$$\frac{}{\Sigma, \Theta, x \Downarrow_{pc} \Sigma, \Theta(x)} \quad \text{[F-VAR]}$$

$$\frac{}{\Sigma, \Theta, (\lambda x.e) \Downarrow_{pc} \Sigma, (\lambda x.e, \Theta)} \quad \text{[F-FUN]}$$

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

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

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

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

$$\frac{}{\Sigma, \Theta, \bot \Downarrow_{pc} \Sigma, \bot} \quad \text{[F-BOT]}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

Auxiliary Functions

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

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

Figure 5: Faceted Evaluation Semantics for Derived Encodings

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

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

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

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

of the standard semantics:

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

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

$$\begin{array}{rcl} L : \mathit{Subst} & \to & \mathit{subst} \\ L(\Theta) & = & \lambda x.\ L(\Theta(x)) \end{array}$$

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

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:

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

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.

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

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

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

Lemma 1. *If* $V = \langle\!\langle\, pc \ ? \ V_1 : V_2 \,\rangle\!\rangle$ *then*

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

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

Lemma 2. *If* $V' = \mathit{deref}(\Sigma, V, pc)$ *then* $L(V') = L(\Sigma)(L(V))$.

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

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

A consequence of Lemma 3 is that evaluation with a pc that is not visible to a view L produces no observable change in the store.

Lemma 4. *Suppose* pc *is not visible to* L *and that*

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

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

Proof. In the auxiliary material for this paper.

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

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

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

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

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

Proof. In the auxiliary material for this paper.

Consequently, if pc is initially empty, then faceted evaluation simulates 2^n standard evaluations, where n is the number of principals.

3.2 Termination-Insensitive Non-Interference

The projection property enables a very simple proof of non-interference; it already captures the idea that information from one view does not leak into an incompatible view, since the projected computations are independent. To formalize this argument, we start by defining two faceted values to be L-*equivalent* if they have identical standard values for view L. This notion of L-equivalence naturally extends to substitutions ($\Theta_1 \sim_L \Theta_2$) and stores ($\Sigma_1 \sim_L \Sigma_2$):

$$\begin{array}{lll} (V_1 \sim_L V_2) & \text{iff} & L(V_1) = L(V_2) \\ (\Theta_1 \sim_L \Theta_2) & \text{iff} & L(\Theta_1) = L(\Theta_2) \\ (\Sigma_1 \sim_L \Sigma_2) & \text{iff} & L(\Sigma_1) = L(\Sigma_2) \end{array}$$

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

Figure 6: Efficient Construction of Faceted Values

$$\langle\!\langle \bullet \ ? \bullet : \bullet \rangle\!\rangle : PC \times Value \times Value \to Value$$

$$\langle\!\langle \ \emptyset \ ? \ V_n \ : \ V_o \ \rangle\!\rangle = V_n$$

$$\langle\!\langle \ \{k\} \cup rest \ ? \ \langle k \ ? \ V_a : V_b \rangle \ : \ \langle k \ ? \ V_c : V_d \rangle \ \rangle\!\rangle = \langle \ k \ ? \ \langle\!\langle \ rest \ ? \ V_a : V_c \ \rangle\!\rangle \ : \ V_d \ \rangle$$
$$\langle\!\langle \ \{\overline{k}\} \cup rest \ ? \ \langle k \ ? \ V_a : V_b \rangle \ : \ \langle k \ ? \ V_c : V_d \rangle \ \rangle\!\rangle = \langle \ k \ ? \ V_c \ : \ \langle\!\langle \ rest \ ? \ V_b : V_d \ \rangle\!\rangle \ \rangle$$

$$\langle\!\langle \ pc \ ? \ \langle k \ ? \ V_a : V_b \rangle \ : \ \langle k \ ? \ V_c : V_d \rangle \ \rangle\!\rangle = \langle \ k \ ? \ \langle\!\langle \ pc \ ? \ V_a : V_c \ \rangle\!\rangle \ : \ \langle\!\langle \ pc \ ? \ V_b : V_d \ \rangle\!\rangle \ \rangle \quad \text{where } k < head(pc)$$
$$\langle\!\langle \ \{k\} \cup rest \ ? \ \langle k \ ? \ V_a : V_b \rangle \ : \ V_o \ \rangle\!\rangle = \langle \ k \ ? \ \langle\!\langle \ rest \ ? \ V_a : V_o \ \rangle\!\rangle \ : \ V_o \ \rangle \quad \text{where } k < head(V_o)$$
$$\langle\!\langle \ \{\overline{k}\} \cup rest \ ? \ \langle k \ ? \ V_a : V_b \rangle \ : \ V_o \ \rangle\!\rangle = \langle \ k \ ? \ V_o \ : \ \langle\!\langle \ rest \ ? \ V_b : V_o \ \rangle\!\rangle \ \rangle \quad \text{where } k < head(V_o)$$
$$\langle\!\langle \ \{k\} \cup rest \ ? \ V_n \ : \ \langle k \ ? \ V_a : V_b \rangle \ \rangle\!\rangle = \langle \ k \ ? \ \langle\!\langle \ rest \ ? \ V_n : V_a \ \rangle\!\rangle \ : \ V_b \ \rangle \quad \text{where } k < head(V_n)$$
$$\langle\!\langle \ \{\overline{k}\} \cup rest \ ? \ V_n \ : \ \langle k \ ? \ V_a : V_b \rangle \ \rangle\!\rangle = \langle \ k \ ? \ V_a \ : \ \langle\!\langle \ rest \ ? \ V_n : V_b \ \rangle\!\rangle \ \rangle \quad \text{where } k < head(V_n)$$

$$\langle\!\langle \ \{k\} \cup rest \ ? \ V_n \ : \ V_o \ \rangle\!\rangle = \langle \ k \ ? \ \langle\!\langle \ rest \ ? \ V_n : V_o \ \rangle\!\rangle \ : \ V_o \ \rangle \quad \text{where } k < head(V_n) \text{ and } k < head(V_o)$$
$$\langle\!\langle \ \{\overline{k}\} \cup rest \ ? \ V_n \ : \ V_o \ \rangle\!\rangle = \langle \ k \ ? \ V_o \ : \ \langle\!\langle \ rest \ ? \ V_n : V_o \ \rangle\!\rangle \ \rangle \quad \text{where } k < head(V_n) \text{ and } k < head(V_o)$$
$$\langle\!\langle \ pc \ ? \ \langle k \ ? \ V_a : V_b \rangle \ : \ V_o \ \rangle\!\rangle = \langle \ k \ ? \ \langle\!\langle \ pc \ ? \ V_a : V_o \ \rangle\!\rangle \ : \ \langle\!\langle \ pc \ ? \ V_b : V_o \ \rangle\!\rangle \ \rangle \quad \text{where } k < head(V_o) \text{ and } k < head(pc)$$
$$\langle\!\langle \ pc \ ? \ V_n \ : \ \langle k \ ? \ V_a : V_b \rangle \ \rangle\!\rangle = \langle \ k \ ? \ \langle\!\langle \ pc \ ? \ V_n : V_a \ \rangle\!\rangle \ : \ \langle\!\langle \ pc \ ? \ V_n : V_b \ \rangle\!\rangle \ \rangle \quad \text{where } k < head(V_n) \text{ and } k < head(pc)$$

Theorem 2 (Termination-Insensitive Non-Interference)**.** *Let L be any view. Suppose*

$$\Sigma_1 \sim_L \Sigma_2 \qquad \Sigma_1, \Theta_1, e \Downarrow_\emptyset \Sigma_1', V_1$$
$$\Theta_1 \sim_L \Theta_2 \qquad \Sigma_2, \Theta_2, e \Downarrow_\emptyset \Sigma_2', V_2$$

Then:

$$\Sigma_1' \sim_L \Sigma_2' \qquad V_1 \sim_L V_2$$

Proof. By the Projection Theorem:

$$L(\Sigma_1), L(\Theta_1), L(e) \downarrow L(\Sigma_1'), L(V_1)$$
$$L(\Sigma_2), L(\Theta_2), L(e) \downarrow L(\Sigma_2'), L(V_2)$$

The L-equivalence assumptions imply that $L(\Theta_1) = L(\Theta_2)$ and $L(\Sigma_1) = L(\Sigma_2)$. Hence $L(\Sigma_1') = L(\Sigma_2')$ and $L(V_1) = L(V_2)$ since the standard semantics is deterministic. □

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

3.3 Efficient Construction of Faceted Values

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

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

$$\begin{aligned} head : Value &\to Label \cup \{\infty\} \\ head(\langle k \ ? \ V_1 : V_2 \rangle) &= k \\ head(R) &= \infty \end{aligned}$$

$$\begin{aligned} head : PC &\to Label \cup \{\infty\} \\ head(\{k\} \cup rest) &= k \quad \text{if } \forall k' \text{ or } \overline{k'} \in rest.\ k < k' \\ head(\{\overline{k}\} \cup rest) &= k \quad \text{if } \forall k' \text{ or } \overline{k'} \in rest.\ k < k' \\ head(\{\}) &= \infty \end{aligned}$$

Figure 6 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 1.

4. Comparison to Prior Semantics

Prior work presented the *no-sensitive-upgrade* (NSU) semantics [39, 5] and the *permissive-upgrade* (PU) semantics [6] for dynamic information flow. In this section, we adapt both of these semantics to our notation to illustrate how faceted evaluation extends both of these prior techniques. For clarity, in this section we assume that there is only a single principal k and omit I/O operations, since the two prior semantics were formalized under these assumptions. Finally, we use the optimized facet-construction operation from Figure 6 in order to avoid reasoning about dead facets.

4.1 Comparison to No-Sensitive-Upgrade Semantics

We formalize the NSU semantics via the evaluation relation

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

defined by the [NSU-*] rules in Figure 7. These rules are somewhat analogous to the faceted evaluation rules of Figure 4, but with some noticeable limitations and restrictions. In particular, the NSU semantics marks each raw value R as being either public or private:

$$\begin{aligned} V \quad ::= \quad & R && \text{public values} \\ \mid \quad & \langle k \ ? \ R : \bot \rangle && \text{private values} \end{aligned}$$

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 $\langle k \ ? \ (\lambda x.e, \Theta') : \bot \rangle$ extends the program counter pc with the label k during the call, reflecting that this computation is dependent on k-private data. Thus, under the NSU semantics, the program counter label is simply a set of principals, and never contains negated principals $\overline{k}$.

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

After the callee returns a result V, the following operation $\langle k \rangle^{pc} V$ creates a faceted value semantically equivalent to $\langle k \ ? \ V : \bot \rangle$, with

Figure 7: No Sensitive Upgrade Semantics

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

$$\frac{}{\Sigma, \Theta, c \Downarrow_{pc} \Sigma, c} \quad \text{[NSU-CONST]}$$

$$\frac{}{\Sigma, \Theta, x \Downarrow_{pc} \Sigma, \Theta(x)} \quad \text{[NSU-VAR]}$$

$$\frac{}{\Sigma, \Theta, (\lambda x.e) \Downarrow_{pc} \Sigma, (\lambda x.e, \Theta)} \quad \text{[NSU-FUN]}$$

$$\frac{}{\Sigma, \Theta, \bot \Downarrow_{pc} \Sigma, \bot} \quad \text{[NSU-BOT]}$$

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

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

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

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

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

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

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

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

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

Figure 8: Permissive Upgrade Semantics (extends Figure 7)

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

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

$$\frac{\begin{array}{c}\Sigma, \theta, e_1 \Downarrow_{pc} \Sigma_1, \langle k \ ? \ a : \bot \rangle \\ \Sigma_1, \Theta, e_2 \Downarrow_{pc} \Sigma', V \\ V' = \langle\!\langle \, pc \ ? \ V : * \, \rangle\!\rangle\end{array}}{\Sigma, \Theta, e_1 \texttt{:=} e_2 \Downarrow_{pc} \Sigma'[a := V'], V} \quad \text{[PU-ASSIGN-K]}$$

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 the [FA-LEFT] and [FA-RIGHT] rules of the faceted semantics.)

In order to preserve the NSU restriction on values, the NSU semantics needs to carefully restrict assignment statements. Essentially, the NSU evaluation rules for assignment statements halt execution in exactly those situations where the faceted semantics would introduce a non-trivial public facet. These rules use the following function to extract the principals in a value:

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

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 3 (Faceted evaluation generalizes NSU evaluation)**.** *If* $\Sigma, \Theta, e \Downarrow_{pc} \Sigma', V$ *then* $\Sigma, \Theta, e \Downarrow\!\!\Downarrow_{pc} \Sigma', V$.

Proof. In the auxiliary material for this paper.

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 [6]. Essentially, the PU semantics works by tracking *partially leaked* data, which

Figure 9: Declassification of Faceted Values

Declassification Rule

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

Downgrade Function

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

we represent here as a faceted value $\langle k\ ?\ R : * \rangle$.[3]

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

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 $\Sigma, \Theta, e \Downarrow_{pc} \Sigma', V$ with the two additional rules shown in Figure 8. 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 [6].

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 (Faceted evaluation generalizes PU evaluation)**.** *If* $\Sigma, \Theta, e \Downarrow_{pc} \Sigma', V$*, then* $\Sigma, \Theta, e \Downarrow_{pc} \Sigma', V$.

Proof. In the auxiliary material for this paper.

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.

5. Facet Declassification

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

[3] In [6], these partially leaked values were represented as R^P, with a superscript P denoting partially leaked.

In the context of multi-process execution [13], declassification is rather challenging. The L and H processes must be coordinated in a careful manner, with all of the attendant problems involved in sharing data between multiple processes. Additionally, allowing declassification may re-introduce timing channels and the termination channel, losing major benefits of the multi-execution approach. In contrast, faceted evaluation makes declassification fairly straightforward. The public and confidential facets are tied together in a single faceted value during execution, so declassification simply requires restructuring the faceted value to migrate information from one facet to another.

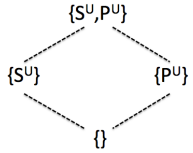
Providing a declassification operation with no restrictions invalidates most security guarantees. For instance, an attacker could declassify a user's password, or overwrite data that would be declassified later by legitimate code. In this manner, valid code intending to declassify the result of a password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

To provide more reliable security guarantees in the presence of declassification with faceted values, we show how to perform *robust declassification* [40], which guarantees that an active attacker, able to introduce code, is no more powerful than a passive attacker, who can only observe the results. (We use robust declassification as an illustrative example, but faceted values could also support other approaches to declassification.)

Robust declassification depends on a notion of *integrity*, which in turn requires that we distinguish between the terms *label* and *principal*. In particular, we introduce a separate notion of principals (P) into our formalism. A label k then marks data as being secret ($\mathtt{S}^P$) or as being low-integrity or untrusted ($\mathtt{U}^P$), both from the perspective of a particular principal P [4].

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

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



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

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 9 performs the appropriate robust declassification. Declassification cannot be performed by arbitrary unauthorized code, or else attackers could declassify all confidential data. Moreover, it is insufficient to allow code "owned" by P to perform declassification, since attackers could leverage that code to declassify data on their behalf. Hence, the rule [F-DECLASSIFY] checks that the control path to this declassification operation has not been influenced by untrusted data, via the check $\mathtt{U}^P \notin pc$.

Robust declassification allows data to move from the $\{\mathtt{S}^P\}$ view to the $\{\}$ view, but never from the $\{\mathtt{S}^P, \mathtt{U}^P\}$ view to the $\{\mathtt{U}^P\}$ view.

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

That is, secret data can be declassified only if it is trusted. The $\texttt{downgrade}_P$ function shown in Figure 9 performs the appropriate manipulation to declassify values. The following lemma clarifies that this function migrates values from the trusted secret view $\{\texttt{S}^P\}$ to the trusted public view $\{\}$, but not into any other view.

Lemma 5. *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. In the auxiliary material for this paper.

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:

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

Theorem 5 (Projection Theorem with Declassification)**.** *Suppose*

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

For any view L for which pc is visible, and where $L_P \neq \{\}$ for each P used in a declassification operation, we have:

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

Proof. In the auxiliary material for this paper.

As a result, non-interference also holds for these same views.

Theorem 6 (Termination Insensitive Non-Interference with Declassification)**.** *Suppose $L_P \neq \{\}$ for each P used in a declassification operation and*

$$\Sigma_1 \sim_L \Sigma_2 \qquad \Sigma_1, \Theta_1, e \Downarrow_\emptyset \Sigma_1', V_1$$
$$\Theta_1 \sim_L \Theta_2 \qquad \Sigma_2, \Theta_2, e \Downarrow_\emptyset \Sigma_2', V_2$$

Then:

$$\Sigma_1' \sim_L \Sigma_2' \qquad V_1 \sim_L V_2$$

Proof. Follows from Theorem 5 via a proof similar to Theorem 2.

6. JavaScript Implementation in Firefox

We incorporate our ideas for faceted evaluation into Firefox through the Narcissus [15] JavaScript engine and the Zaphod [26] Firefox plugin. The ZaphodFacets implementation [4] extends the faceted semantics to handle the additional complexities of JavaScript. Exceptions are particularly tricky, and we halt execution if an exception may leak information.

We added two new primitives to the language. The `makePrivate` function turns a value into a faceted value with a public facet of `undefined` [5]. This approach allows developers to specify a different public value through the JavaScript idiom for specifying default values. The following code sets `x` to a faceted value of $\langle k\ ?\ 42 : 0 \rangle$. The high value of x is set to 42, since `(42 || 0) === 42`; the low value will be 0, since `(undefined || 0) === 0`.

```
var x = makePrivate(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)` would return `0`. Generally, the browser's security policy should use these two functions (or variants) 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.

To track information flow through the Document Object Model (DOM), our implementation uses the `dom.js` DOM implementation written in JavaScript [17], to preventing the attacker from sanitizing data by writing it to the DOM and later rereading it. Our implementation is available online with some examples [4], including the code from Figure 1.

6.1 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.

We specify a simple policy that 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 true value.

In our example, the web developer is making use of a library for hashing passwords on the client side. The library is benign, but an attacker uses an XSS vulnerability in the page to wrap the hashing library and export the password to `evil.com`, a site under the attacker's control. 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`, incorporating the password into the name of the requested image. However, in an attempt to evade our controls, it first writes the password to the `class` attribute of the `title1` element and then rereads it from the first `h1` element. Without knowledge of the DOM structure of the page, it is not possible to know whether this code leaks information. However, with dom.js we persist the different facets of `secret` to the DOM so that no security information is lost. While the page can only render a single facet, it is critical that we maintain other views of the document.

With this example, `evil.com` sees only the public facet of `secret`, not the true password. Trusted same-origin sources *do* see the true value, and therefore work correctly with the page.

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).

6.2 Performance Results

Our approach is similar to Devriese and Piessens's work on secure multi-execution [13]. To understand the performance tradeoff between these two approaches we also 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 10.6.8. The machine had a 2.3 GHz Intel Core i7 processor with 4 cores and 8 GB of memory. For our benchmark, we used

[5] A string specifying the principal can be given as the second argument if multiple principals are required.

Figure 10: Faceted Evaluation vs. Secure Multi-Execution

	Times in ms		
	Secure multi-execution		Faceted
# principals	sequential	concurrent	execution
0	273,774	283,450	310,561
1	513,561	283,503	348,725
2	961,357	332,303	387,121
3	1,783,609	597,595	421,566
4	3,324,480	1,093,951	461,543
5	*	1,981,927	503,364
6	*	*	540,618
7	*	*	575,100
8	*	*	614,150

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

the crypto-md5 test from the SunSpider [38] benchmark suite. We modified this program to include 8 hashing operations with some inputs marked as confidential. Our test cases involve 0 through 8 principals. In each case, every principal marks one element as private; additional hash inputs are public. For example, test 1 hashes 1 confidential input and 7 public inputs. Test 8 hashes 8 confidential inputs, each marked as confidential by a distinct principal, and has no public hash inputs. Our results are summarized in Figure 10.

Our results highlight the tradeoffs between the different approaches. The sequential variant of secure multi-execution had the most lightweight infrastructure of the three approaches, reflected in its good performance when there are 0 principals. However, it can neither take advantage of multiple processors nor avoid unnecessary work. As a result, once even a single principal was involved, it was the worst performer. The time required roughly doubles with each additional principal.

Concurrent secure multi-execution outperforms our faceted evaluation implementation when the number of principals is small. However, as the number of principals increases, faceted evaluation quickly becomes the more efficient approach, since under secure multi-execution the number of processes increases exponentially compared to the number of principals. With three principals, faceted evaluation outperforms concurrent secure multi-execution in our tests. Beyond this point execution time for concurrent secure multi-execution roughly doubles with each added principal, as the elements in the lattice now outnumber the available cores.

7. Related Work

A few publications have discussed performing multiple executions to guarantee security properties. Capizzi et al.'s *shadow executions* [9] develop 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 [13] 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. Kashyap et al. [23] clarify some properties of secure multi-execution.

Our semantics are closely related to work by Pottier and Simonet [28]. While they prove non-interference 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. Our work departs from theirs in that we evaluate labeled expressions and values to dynamically guarantee non-interference, rather than using them to make static guarantees.

Kolbitsch et al. [25] 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.

Other research has previously studied information-flow analysis for JavaScript. Vogt et al. [36] track information flow in Firefox to defend against XSS attacks. Russo and Sabelfeld [30] study timeout mechanisms. Russo et al. [32] discuss dynamic tree structures, with obvious applications to the DOM. Bohannon et al. [8] consider non-interference in JavaScript's reactive environment. Chugh et al. [11] create a framework for information flow analysis with "holes" for analyzing dynamically evaluated code. Dhawan and Ganapathy [14] discuss JavaScript-based browser extensions (JSEs). Jang et al. [21] give an excellent overview of how JavaScript is used to circumvent privacy defenses.

Information flow analysis largely traces its roots back to Denning [12]. Volpano et al. [37] codify Denning's approach as a type system, and also offer a proof of its soundness. Heintze and Riecke [19] design a type system for their purely functional SLam Calculus, which they extend to include mutable reference cells, concurrency, and integrity guarantees. Sabelfeld and Myers [33] offer an extensive survey of other research on information flow.

Myers [27] discusses JFlow, a variant of Java with security types to provide strong information flow guarantees. JFlow was the basis for Jif [22], a production-worthy language with information flow controls. Birgisson et al. [7] show how capabilities can guarantee information flow policies. Hunt and Sands [20] describe a flow-sensitive type system. Russo and Sabelfeld [31] discuss the tradeoffs between static and dynamic analyses in some depth. Le Guernic et al. [18] examine code from branches not taken, increasing precision at the expense of run-time performance overhead. Shroff et al. [34] use a purely-dynamic analysis to track variable dependencies and reject more insecure programs over time.

Zdancewic [40] uses integrity labels to provide *robust declassification*. Askarov and Myers [2] consider *checked endorsements*. Chong and Myers [10] use a framework for application-specific declassification policies. Askarov and Sabelfeld [3] study a declassification framework specifying what and where data is released. Vaughan and Chong [35] infer declassification policies for Java programs. Askarov et al. [1] highlight complications of *intermediary output channels*. Rafnnson et al. [29] buffer output to reduce data lost from intermediary output channels and termination behavior. King et al. [24] study false alarms caused by implicit flows.

8. Discussion

Information flow non-interference is a tricky security property to enforce via dynamic monitoring, since it is a *2-safety property*: non-interference can be refuted only by observing two executions (cmp. Theorem 2). Conversely, a *1-safety property* can be refuted by observing a single execution, and so 1-safety properties are more amenable to dynamic enforcement. From this perspective, various prior techniques dynamically enforce a 1-safety property that conservatively approximates the desired 2-safety property of non-interference, but this conservative approximation introduces false alarms on implicit flows. Interestingly, our projection property (Theorem 1) is a 1-safety property that suffices to prove non-interference (Theorem 2) without introducing false alarms.

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

References

[1] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS '08*, pages 333–348. Springer-Verlag, 2008.

[2] Aslan Askarov and Andrew Myers. A semantic framework for declassification and endorsement. In *ESOP*, pages 64–84, 2010.

[3] Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *IEEE Computer Security Foundations Symposium*, pages 43–59, Washington, DC, USA, 2009. IEEE Computer Society.

[4] Thomas H. Austin. ZaphodFacetes github page. `https://github.com/taustin/ZaphodFacets`, 2011.

[5] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS '09: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 113–124, New York, NY, USA, 2009. ACM.

[6] Thomas H. Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 1–12. ACM, 2010.

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

[8] Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. Reactive noninterference. In *ACM Conference on Computer and Communications Security*, pages 79–90, 2009.

[9] R. Capizzi, A. Longo, V.N. Venkatakrishnan, and A.P. Sistla. Preventing information leaks through shadow executions. In *ACSAC*, pages 322 –331, dec 2008.

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

[11] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *PLDI*, pages 50–62, 2009.

[12] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[13] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. *Security and Privacy, IEEE Symposium on*, 0:109–124, 2010.

[14] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript-based browser extensions. In *ACSAC*, pages 382–391, 2009.

[15] Brendan Eich. Narcissus–JS implemented in JS. Available on the web at `https://github.com/mozilla/narcissus/`.

[16] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974.

[17] Andreas Gal, David Flanagan, and Donovon Preston. dom.js github page. `https://github.com/andreasgal/dom.js`, accessed October 2011, 2011.

[18] Gurvan Le Guernic, Anindya Banerjee, Thomas P. Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. In *ASIAN*, pages 75–89, 2006.

[19] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Symposium on Principles of Programming Languages*, pages 365–377, 1998.

[20] Sebastian Hunt and David Sands. On flow-sensitive security types. In *POPL*, pages 79–90, 2006.

[21] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in javascript web applications. In *ACM Conference on Computer and Communications Security*, pages 270–283, 2010.

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

[23] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *IEEE Security and Privacy*, 2011.

[24] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *International Conference on Information Systems Security*, pages 56–70, 2008.

[25] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-cloaking internet malware. Technical Report MSR-TR-2011-94, Microsoft Research Technical Report, 20011.

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

[27] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

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

[29] Willard Rafnsson and Andrei Sabelfeld. 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, 2011.

[30] Alejandro Russo and Andrei Sabelfeld. Securing timeout instructions in web applications. In *IEEE Computer Security Foundations Symposium*, 2009.

[31] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *IEEE Computer Security Foundations Symposium*. IEEE Computer Society, 2010.

[32] Alejandro Russo, Andrei Sabelfeld, and Andrey Chudnov. Tracking information flow in dynamic tree structures. In *ESORICS*, pages 86–103, 2009.

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

[34] Paritosh Shroff, Scott F. Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *CSF*, pages 203–217, 2007.

[35] Jeffrey Vaughan and Stephen Chong. Inference of expressive declassification policies. In *IEEE Security and Privacy*, 2011.

[36] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, 2007.

[37] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.

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

[39] Stephan Arthur Zdancewic. *Programming languages for information security*. PhD thesis, Cornell University, 2002.

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