A Functional View of Imperative Information Flow

Thomas H. Austin[1], Cormac Flanagan[1], and Martín Abadi[1,2]

[1] University of California, Santa Cruz
[2] Microsoft Research Silicon Valley

Abstract. We analyze dynamic information-flow control for imperative languages in terms of functional computation. Specifically, we translate an imperative language to a functional language, thus accounting for the main difficulties of information-flow control in the imperative language.

1 Introduction

Dynamic information-flow control appears to be increasingly popular, useful, but intricate. These characteristics partly stem from the treatment of realistic languages, in particular with imperative features. (Dynamic information-flow control for the pure lambda calculus seems considerably simpler and less practically relevant.) In this paper we aim to contribute to the understanding of dynamic information-flow control, by translating imperative features to a functional calculus. The translation accounts for difficulties of information-flow control in the source language; information-flow control in the target language is comparatively straightforward.

The rest of this introduction presents our goals and results in more detail, putting them in the context of a long line of prior work.

Information-flow security

In the classic information-flow model of security [6, 7, 9], data is associated with security levels, and then one aims to guarantee that information propagates consistently with those levels. For example, for confidentiality policies, the security levels may be "secret" and "public", and then one requires, for instance, that secret data is not disclosed on public output channels. For integrity policies, similarly, the security levels may be "tainted" and "trusted", and then trusted outputs should not be computed from tainted inputs.

More precisely, a frequent, principled requirement is *non-interference*, which basically means that events observable at certain security levels are not influenced by events at certain other security levels. The events often correspond to straightforward inputs and outputs. So, concretely, the non-interference property implies that secret inputs do not influence public outputs, and that tainted inputs do not influence trusted outputs. Note that the non-interference property precludes partial flows of information. For example, given a secret integer input,

non-interference forbids revealing not only the entire integer but also its sign on a public output channel.

The enforcement of information-flow rules can rely on a variety of mechanisms. Since the early days of the subject, special architectures have been considered for this purpose. More commonly, the enforcement may be done in software, particularly in operating systems and in programming languages.

Static and dynamic language-based information-flow control

Working within a programming language enables fine-grained tracking of information flows. In this context, the enforcement may be done statically (e.g., at compile-time) or dynamically (at run-time). Static and dynamic techniques may also be combined. Sabelfeld and Myers [18] review the research on language-based information-flow control as of 2003.

Although dynamic techniques are far from new, until recently they were somewhat neglected in the programming-language research literature, because of concerns that they could not be both sound and precise enough. In contrast, flexible type systems were developed for static enforcement [13, 22], and led to realistic programming languages such as Jif [14, 16] and FlowCaml [10, 17].

Nevertheless, various forms of dynamic information-flow control may be attractive in practice, for instance in the context of widespread languages such as Perl and JavaScript (which are light on static guarantees). Language-level dynamic techniques also connect with dynamic tracking in operating systems, where they have been prominent in recent research artifacts (such as Asbestos [15] and HiStar [24]) and in deployed commercial systems (in SELinux and in aspects of Windows, in particular).

Accordingly, we have seen renewed interest and substantial progress in research on dynamic information-flow control in recent years [3, 11, 19, 20]. This research has aimed to develop, then to use, flexible and efficient systems that satisfy non-interference properties. For instance, this research effectively supports promising uses of dynamic information-flow control in Web browsers [5, 8, 12, 21]. This research is often clever and intricate—so we do not attempt to give a complete description here, but we focus on some of the intricacies below.

Imperative systems and the problem of sensitive upgrades

Whether static or dynamic, information-flow control accounts for a useful range of realistic language features and computational phenomena. Even straightforward imperative features can be challenging. The following classic example illustrates some of the difficulties:

Function `f x`	`x` = `true`^{H}	`x` = `false`^{H}
`y := true;`	`y` = `true`^{L}	`y` = `true`^{L}
`z := true;`	`z` = `true`^{L}	`z` = `true`^{L}
`if (x) y := false;`	`y` set to `false`^{H}	–
`if (!y) z := false;`	–	`z` set to `false`^{L}
`!z;`	returns `true`^{L}	returns `false`^{L}
Return value:	`true`^{L}	`false`^{L}

The superscript H marks private data; similarly, the superscript L marks public data. When ($\texttt{f true}^H$) is called, the value for the reference cell `y` must be updated. Since the update to `y` is conditional on the value of `x`, information about `x`'s private value leaks to `y`. If the value for `y` is set to $\texttt{false}^H$ to capture the influence of private data, an attacker can use `y` in a second conditional assignment statement to leak the value of `x`. Since `y` is $\texttt{false}^H$, the value for `z` remains $\texttt{true}^L$, and therefore ($\texttt{f true}^H$) = $\texttt{true}^L$. When ($\texttt{f false}^H$) is called, the value of `y` remains $\texttt{true}^L$. Therefore, `z` is set to $\texttt{false}^L$ (since `y` is public), and ($\texttt{f false}^H$) = $\texttt{false}^L$. Thus, if allowed to run to completion, this piece of code leaks a secret input.

One sensible approach to preventing this leak is to forbid sensitive upgrades [3, 23]. Specifically, programs are not allowed to write to locations that contain low-security data within high-security contexts (e.g., after branching on some secret). This approach can be realized either statically or dynamically:

- Statically, it seems fairly natural to require that each location has a fixed security level, much like (in virtually all programming languages) it has a fixed type, and to combine the security level with the type.
- Dynamically, on the other hand, the various realizations of this approach introduce interesting twists, typically to enhance efficiency or flexibility.

Simple functional systems

Remarkably, these difficulties and the related complications do not seem to arise in the context of pure functional languages.

In the pure lambda calculus, in particular, a traditional system of labels, with extremely simple rules, suffices for sound information-flow tracking [2]. In this system, each subexpression of a program may have a label that indicates its sensitivity. Formally, if e is an expression and H is a label, then $H : e$ is an expression.

A basic rule permits lifting labels in order to permit function application:

$$(H : e_1)(e_2) \;\rightarrow\; H : (e_1\ e_2)$$

Here, e_1 and e_2 are arbitrary expressions. When e_1 is a function of the form $(\lambda x.e)$, this rule enables the subsequent, straightforward application of the β rule (which returns the result of replacing x with e_2 in e, as usual), underneath the label H.

Analogously, for conditionals, labels may be lifted out of guards:

$$\texttt{if}\ (H : e_1)\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \;\rightarrow\; H : (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3)$$

If conditionals are primitive, then so is the rule. If conditionals are encoded as functions, then this rule is easily derived from the rule for application.

Functional vs. imperative systems

This contrast between functional and imperative systems suggests a number of questions:

- Are the difficulties really specific to imperative programming?
  If so, perhaps the lambda calculus is not an appropriate core language for understanding computational phenomena, at least not for security?
- On the other hand, since imperative programs can be translated to functional programs, by passing stores as values, could the difficulties be analyzed and resolved by translation? Perhaps more than one translation should be considered in order to account for some of the twists in information-flow control?

For static information-flow control, such questions seem relatively simple, and they have been at least partly resolved. In particular, one can translate various static information-flow systems for imperative languages to the Dependency Core Calculus (DCC) [1], which is a functional computational lambda calculus. The present paper aims to address such questions for dynamic information-flow control.

Contents of this paper

For this purpose, the paper defines a simple imperative language and a simple functional language. Both have primitive rules for tracking flows of information.

- In the imperative language, the rules are analogous to those from the literature, and may be judged reasonably permissive.
- In the functional language, the rules amount to a mostly unsurprising management of labels, of the kind that we have shown for the lambda calculus.

Both sets of rules for tracking flows of information are sound, in the sense that they yield non-interference properties. The paper shows how to account for the imperative rules in terms of the functional rules, by translation.

There is a straightforward, naive translation from the imperative language to the functional language (by passing stores, as indicated above). Although sound, this translation is overly conservative. For instance, the translation of

$$\texttt{if}\ H\!:\!x\ \texttt{then}\ y\texttt{:=}0\ \texttt{else}\ y\texttt{:=}0$$

is roughly the function

$$(\lambda\sigma.\ \texttt{if}\ H\!:\!x\ \texttt{then}\ \langle 0, \sigma[y := 0]\rangle\ \texttt{else}\ \langle 0, \sigma[y := 0]\rangle)$$

which, when applied to a store σ_0 yields

$$H\!:\!\langle 0, \sigma_0[y := 0]\rangle$$

where $\sigma_0[y := 0]$ is the store that agrees with σ_0 except that it maps y to 0. Note that the entire resulting store is under the label H, so it is all tainted. Reading

the contents of some unrelated location z after executing this program will yield a result of the form $H : v$, always with a label H.

This paper defines a more sophisticated translation that addresses these difficulties. This translation includes a refined management of the store as a value. It dynamically separates and recombines the parts of a store that are updated or left unchanged in high-security contexts, without tainting the unchanged parts in the process, and without requiring static information for distinguishing those two parts of the store. Via this translation, the tracking for the imperative language can be replicated by the tracking in the functional language.

Both translations can be written in monadic style, if one wishes, and arguably the monadic style has certain benefits. However, we do not believe that monads are the answer: the straightforward use of monads, by itself, does little to resolve the problems that we tackle.

Variants and further work

Our languages have certain specific characteristics (often limitations), and considering other languages might be attractive in further work.

- For simplicity, our source language does not include dynamic allocation.
- Both our source language and our target language permit dynamic tests on labels. Such tests are attractive because they make it easy to write certain programs, and they preserve the main security properties of interest. On the other hand, their inclusion means that our target language is not the minimal labelled lambda calculus.
- We consider only two security levels, rather than a general lattice of security levels. Although the case of two levels is both instructive and common in practice, it seems interesting to generalize our results to arbitrary lattices.
- Finally, one may wonder about other language features, such as exceptions.

Our results are purely theoretical. However, analogous results might be of practical value. In scenarios where one language is compiled to another, refined translations such as ours may enable dynamic information-flow control to happen, soundly and flexibly, in the target language.

2 An Imperative Language with Information-Flow Control

We consider an imperative language ImpFlow that supports information-flow control. Its definition, which we give in this section, is along the lines of previous work [3]. The language (see Figure 1) extends the lambda calculus with mutable reference cells, a way of designating private expressions, and a way of testing for private labels. Terms include variables (x), functions ($\lambda x.m$), and function applications ($m_1\ m_2$). To support imperative updates and illustrate the challenges of implicit flows, ImpFlow includes terms for assignment ($i := m$) and dereferencing ($!i$). Here, i denotes an address in the range of $1, ..., n$. The term $H : m$

marks the result of evaluating m as private. The term $H?\ m$ tests if the result of evaluating m is private.

The semantics for ImpFlow is defined in Figure 1. A value $(\lambda x.m)^k$ combines an abstraction $(\lambda x.m)$ with a label k, where k may be either L for public data or H for private data. A program counter pc tracks the influences on the current execution. A store (Σ) maps addresses (i) to values (w).

We define the operational semantics of ImpFlow via the big-step relation:

$$m, \Sigma \Downarrow_{pc} w, \Sigma'$$

This relation evaluates an expression m in the context of a store Σ and the current label pc of the program counter, and it returns the resulting value w and the (possibly modified) store Σ'.

When a function is evaluated via the [M-FUN] rule, the program counter is attached as the label of the function. The evaluation of a value w is similar, via rule [M-VAL], and relies on an auxiliary operation to join a label onto a value:

$$(\lambda x.m)^k \sqcup pc \quad \stackrel{\text{def}}{=} \quad (\lambda x.m)^{k \sqcup pc}$$

The rule [M-APP] evaluates the body of the called function $(\lambda x.m)^k$ with upgraded program counter label $pc \sqcup k$, since the callee "knows" that that function was invoked. As usual, we assume $L \sqsubset H$, and so, for example, $L \sqcup H = H$.

The [M-LABEL] rule for $H : m$ evaluates the subexpression m and joins the label H to the resulting value, marking the result as private. The [M-PRED] rule evaluates an expression to a value and returns true (Church-encoded) if either the program counter or the value's label is private. Otherwise, it returns false.

Mutable reference cells must be handled carefully in order to guarantee non-interference. The [M-DEREF] rule reads a value from the store and returns it, joining the program counter to the value. The real complexity lies in handling updates to the store. Much as in several previous systems, the [M-ASSIGN] rule uses the no-sensitive-upgrade check [3, 23], which forbids updates to public reference cells when in a private block of code. Here the function *label* extracts the label from a value:

$$\mathit{label}(\ (\lambda x.m)^k\ ) \quad \stackrel{\text{def}}{=} \quad k$$

3 Non-Interference in ImpFlow (Start)

We briefly consider information-flow guarantees satisfied by ImpFlow.

For this purpose, we consider two ImpFlow values to be *equivalent modulo under labels* if they only differ under H superscripts. More formally, the equivalent modulo under labels relation on ImpFlow values, terms, or stores is the congruence closure of the rule:

$$(\lambda x.m_1)^H \quad \sim_{ul} \quad (\lambda x.m_2)^H$$

The key property of the ImpFlow semantics is termination-insensitive non-interference: if two computations from equivalent initial states terminate, then they yield equivalent results.

Figure 1: The Imperative Language ImpFlow

Syntax:

$m ::=$	*Term*
x	variable
$(\lambda x.m)$	abstraction
$m_1\ m_2$	application
$i := m$	assignment
$!i$	dereference
$H : m$	high expression
$H?\ m$	high test
w	values
x, y, z	*Variable*

Runtime Syntax:

$$\begin{array}{rcrcl} w & \in & \mathit{Value} & ::= & (\lambda x.m)^k \\ k, pc & \in & \mathit{Label} & ::= & L \mid H \\ \Sigma & \in & \mathit{Store} & = & \mathit{Addr} \rightarrow w \\ i & \in & \mathit{Addr} & = & \{1, ..., n\} \end{array}$$

Evaluation Rules: $\boxed{m, \Sigma \Downarrow_{pc} w, \Sigma'}$

$$\frac{}{w, \Sigma \Downarrow_{pc} w \sqcup pc, \Sigma} \quad [\text{M-VAL}]$$

$$\frac{}{!i, \Sigma \Downarrow_{pc} \Sigma(i) \sqcup pc, \Sigma} \quad [\text{M-DEREF}]$$

$$\frac{}{(\lambda x.m), \Sigma \Downarrow_{pc} (\lambda x.m)^{pc}, \Sigma} \quad [\text{M-FUN}]$$

$$\frac{m, \Sigma \Downarrow_{pc} w, \Sigma' \qquad pc \sqsubseteq \mathit{label}(\Sigma'(i))}{i := m, \Sigma \Downarrow_{pc} w, \Sigma'[i := w]} \quad [\text{M-ASSIGN}]$$

$$\frac{m_1, \Sigma \Downarrow_{pc} (\lambda x.m)^k, \Sigma_1 \qquad m_2, \Sigma_1 \Downarrow_{pc} w_2, \Sigma_2 \qquad m[x := w_2], \Sigma_2 \Downarrow_{pc \sqcup k} w, \Sigma'}{m_1\ m_2, \Sigma \Downarrow_{pc} w, \Sigma'} \quad [\text{M-APP}]$$

$$\frac{m, \Sigma \Downarrow_{pc} (\lambda x.m')^k, \Sigma' \qquad \text{If } pc = H \text{ or } k = H \text{ then } b = \texttt{true} \text{ else } b = \texttt{false}}{H?\ m, \Sigma \Downarrow_{pc} b^{pc}, \Sigma'} \quad [\text{M-PRED}]$$

$$\frac{m, \Sigma \Downarrow_{pc} w, \Sigma'}{H : m, \Sigma \Downarrow_{pc} w \sqcup H, \Sigma'} \quad [\text{M-LABEL}]$$

Standard Encodings:

$$\begin{array}{rcl} \texttt{true} & \stackrel{\text{def}}{=} & (\lambda x.(\lambda y.x)) \\ \texttt{false} & \stackrel{\text{def}}{=} & (\lambda x.(\lambda y.y)) \\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 & \stackrel{\text{def}}{=} & (e_1\ (\lambda d.e_2)\ (\lambda d.e_3))\ (\lambda x.x) \end{array}$$

Theorem 1 (Termination-insensitive non-interference for ImpFlow). *Suppose* $m_1 \sim_{ul} m_2$ *and*

$$m_1, \Sigma \Downarrow_{pc} w_1, \Sigma_1$$
$$m_2, \Sigma \Downarrow_{pc} w_2, \Sigma_2$$

where w_1, w_2 *are public Booleans (*$\texttt{true}^L$ *or* $\texttt{false}^L$*). Then* $w_1 = w_2$.

As usual, the theorem could be refined to say that the final stores Σ_1 and Σ_2 are related, and it could also be generalized to allow different but related initial stores and to apply to non-Boolean results. Such changes are fairly routine, and we avoid them for simplicity.

Theorem 1 can be proved directly. (See for example Austin and Flanagan [3] for analogous results). Below, we study how to obtain non-interference for ImpFlow via a translation from ImpFlow into a functional language. We proceed in this manner not so much because the direct proof would be hard, but rather in order to give evidence that the translation is helpful and sensible. The resulting indirect proof is remarkably simple.

4 A Functional Language with Information-Flow Control

Next we consider a functional language called FunFlow with information-flow control. FunFlow is an extension of the lambda calculus (see Figure 2). It includes variables (x), functions ($\lambda x.e$), function applications ($e_1\ e_2$), a mechanism for specifying high expressions ($H : e$), and label reflection ($H?\ e$). Unlike ImpFlow, this language is purely functional and does not include reference cells. It also leaves implicit the label for low-security data: H is the only label.

Though this language is minimal, additional constructs can be encoded, as usual. Figure 2 details the constructs employed in our translation. Boolean values (`true` and `false`) and conditional evaluation (`if` e_1 `then` e_2 `else` e_3) are Church-encoded in the usual way. Tuples ($\langle e_1, ..., e_n \rangle$) contain a list of expressions (and are used when translating stores from ImpFlow to FunFlow, which we discuss in more depth in Section 6). There are operations for projecting the ith value from a tuple ($e.i$) and setting the value at the ith position ($e[i := e']$). The let construct includes a variant (`let` $\langle x_1, x_2 \rangle$ `=` e_1 `in` e_2) that deconstructs pairs.

We formalize the operational semantics of FunFlow via the small-step relation:

$$e_1 \rightarrow e_2$$

This relation evaluates an expression e_1 and returns a new expression e_2. Instead of using a program counter to track the current influences on execution, the semantics for FunFlow divides evaluation contexts ($\mathcal{E}$) into private contexts ($Context_H$), which have an enclosing $H :$, and public contexts ($Context_L$). Values in FunFlow are either functions ($\lambda x.e$) or labeled values ($H : v$).

The rule [E-BETA] applies β-reduction, replacing occurrences of the specified variable with the given value. The rule [E-LIFT] handles the application of a private function $H : v$ to an argument v' by moving the entire application under the H, yielding $H : (v\ v')$.

Figure 2: FunFlow Language

Syntax:

$e ::=$		*Term*
	x	variable
	$(\lambda x.e)$	abstraction
	$e_1\ e_2$	application
	$H : e$	high expression
	$H?\ e$	high test
x, y, z		*Variable*

Runtime Syntax:

$$\begin{array}{rcll} v & \in & \mathit{Value} & ::= \quad (\lambda x.e) \mid H : v \\ \mathcal{E} & \in & \mathit{Context} & ::= \quad \mathcal{E}\ e \mid v\ \mathcal{E} \mid H : \mathcal{E} \mid H?\ \mathcal{E} \mid \bullet \end{array}$$

$$\begin{array}{rcl} \mathit{Context}_H & = & \mathcal{E}_1[H : \mathcal{E}_2] \\ \mathit{Context}_L & = & \mathit{Context} \ / \ \mathit{Context}_H \end{array}$$

Evaluation Rules: $\boxed{e \ \rightarrow \ e'}$

$$\begin{array}{rll} \mathcal{E}[(\lambda x.e)\ v] \rightarrow \mathcal{E}[e[x := v]] & & [\text{E-BETA}] \\ \mathcal{E}[(H : v)\ v'] \rightarrow \mathcal{E}[H : (v\ v')] & & [\text{E-LIFT}] \\ \mathcal{E}[H?\ (H : v)] \rightarrow \mathcal{E}[\texttt{true}] & & [\text{E-PRED-TRUE1}] \\ \mathcal{E}[H?\ (\lambda x.e)] \rightarrow \mathcal{E}[\texttt{true}] & & [\text{E-PRED-TRUE2}] \\ \qquad \text{if } \mathcal{E} \in \mathit{Context}_H & & \\ \mathcal{E}[H?\ (\lambda x.e)] \rightarrow \mathcal{E}[\texttt{false}] & & [\text{E-PRED-FALSE}] \\ \qquad \text{if } \mathcal{E} \in \mathit{Context}_L & & \end{array}$$

Standard Encodings:

$$\begin{array}{rcl} \texttt{true} & \stackrel{\text{def}}{=} & (\lambda x.(\lambda y.x)) \\ \texttt{false} & \stackrel{\text{def}}{=} & (\lambda x.(\lambda y.y)) \\ \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 & \stackrel{\text{def}}{=} & (e_1\ (\lambda d.e_2)\ (\lambda d.e_3))\ (\lambda x.x) \\ \langle e_1, ..., e_n \rangle & \stackrel{\text{def}}{=} & (\lambda f.(f\ e_1..e_n)) \\ e.i & \stackrel{\text{def}}{=} & e\ (\lambda x_1..x_n.x_i) \\ e[i := e'] & \stackrel{\text{def}}{=} & e\ (\lambda x_1..x_n.(\lambda f.f\ x_1..x_{i-1}\ e'\ x_{i+1}..x_n)) \\ \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 & \stackrel{\text{def}}{=} & (\lambda x.e_2)\ e_1 \\ \texttt{let } \langle x_1, x_2 \rangle \texttt{ = } e_1 \texttt{ in } e_2 & \stackrel{\text{def}}{=} & \texttt{let } x \texttt{ = } e_1 \texttt{ in} \\ & & \texttt{let } x_1 \texttt{ = } x.1 \texttt{ in} \\ & & \texttt{let } x_2 \texttt{ = } x.2 \texttt{ in } e_2 \text{ for } x \notin FV(e_2) \\ e_1\ ;\ e_2 & \stackrel{\text{def}}{=} & \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \text{ for } x \notin FV(e_2) \end{array}$$

Label reflection determines if a value is private, either by an explicit label (rule [E-PRED-TRUE1]) or if it is in a private context (rule [E-PRED-TRUE2]). When the value is not explicitly H-labeled and is in a public context, then the result is `false` (rule [E-PRED-FALSE]).

For consistency with prior work, there are significant differences in the formal semantics of ImpFlow and FunFlow. The semantics of ImpFlow follows that of Austin and Flanagan [3]; it is a big-step semantics with *universal labeling*, where every value has exactly one associated label. In contrast, the semantics of FunFlow follows that of prior work on labeled lambda calculus [2]; it is a small-step semantics with no *pc* label, and with *sparse-labeling*, where values may have zero or multiple enclosing labels.

5 Non-Interference in FunFlow

We consider two FunFlow expressions to be *equivalent modulo under labels* if the only difference between the two expressions is in their private components. More formally, the relation $e_1 \sim_{ul} e_2$ is defined as the congruence closure of

$$H : e_1 \sim_{ul} H : e_2$$

If two expressions are equivalent modulo under labels, evaluating them produces values that are equivalent modulo under labels, assuming both evaluations terminate. It is possible that only one of the computations will diverge, thereby leaking one bit of information. This notion is formalized below:

Theorem 2 (Termination-insensitive non-interference for FunFlow). *If $e_1 \sim_{ul} e_2$ and $e_1 \to^* v_1$ and $e_2 \to^* v_2$, then $v_1 \sim_{ul} v_2$.*

This non-interference proof follows from the following lemma that relates each evaluation step of e_1 to a corresponding computation (possibly divergent) of e_2:

Lemma 1. *If $e_1 \sim_{ul} e_2$ and $e_1 \to e_1'$, then either $e_2 \to^\infty$ or there exists e_2' such that $e_2 \to^* e_2'$ and $e_1' \sim_{ul} e_2'$.*

The proof of this lemma is via a case analysis of $e_1 \to e_1'$.

6 Translating Imperative Information-Flow Control to Functional Information-Flow Control

We next explain how to translate ImpFlow programs into FunFlow. Our translation preserves semantics (at least for properly terminating programs). It also preserves information-flow guarantees, as we show in the next section. Of course, other translations may be invented, possibly with somewhat different properties and making different trade-offs. For instance, it may be viable to define a translation that passes two stores, basically one for each level but with "low writes" modifying both stores. Our translation, instead, occasionally creates two stores but then combines them into a single one.

Figure 3: Translation Rules

Term Translation Rules: $\boxed{[\![\bullet]\!] : m \ \rightarrow \ e}$

$$
\begin{array}{rll}
[\![x]\!] & = \lambda\sigma.\langle x, \sigma\rangle & [\text{TR-VAR}] \\
[\![(\lambda x.m)]\!] & = \lambda\sigma.\langle(\lambda x\sigma'.[\![m]\!]\ \sigma'), \sigma\rangle & [\text{TR-LAM}] \\
[\![H : m]\!] & = \lambda\sigma.\texttt{let}\ \langle x, \sigma'\rangle = [\![m]\!]\ \sigma\ \texttt{in}\ \langle H : x, \sigma'\rangle & [\text{TR-HI}] \\
[\![H?\ m]\!] & = \lambda\sigma.\texttt{let}\ \langle x, \sigma'\rangle = [\![m]\!]\ \sigma & [\text{TR-PRED}] \\
 & \quad \texttt{in}\ \langle \texttt{if}\ H?\ x\ \texttt{then}\ [\![\texttt{true}^L]\!]_{\texttt{val}}\ \texttt{else}\ [\![\texttt{false}^L]\!]_{\texttt{val}}, \sigma'\rangle & \\
[\![!i]\!] & = \lambda\sigma.\langle\sigma.i, \sigma\rangle & [\text{TR-DEREF}] \\
[\![i := m]\!] & = \lambda\sigma.\texttt{let}\ \langle x, \sigma'\rangle = [\![m]\!]\ \sigma & [\text{TR-ASSIGN}] \\
 & \quad \texttt{in}\ \langle x, \sigma'[i := x]\rangle & \\
[\![w]\!] & = \lambda\sigma.\langle [\![w]\!]_{\texttt{val}}, \sigma\rangle & [\text{TR-VAL}] \\
[\![m_1\ m_2]\!] & = \lambda\sigma.\texttt{let}\ \langle x_1, \sigma_1\rangle = [\![m_1]\!]\ \sigma\ \text{in} & [\text{TR-APPLY}] \\
 & \quad\ \ \texttt{let}\ \langle x_2, \sigma_2\rangle = [\![m_2]\!]\ \sigma_1\ \text{in} & \\
 & \quad\ \ \texttt{let}\ \langle x, \sigma'\rangle = x_1\ x_2\ \sigma_2\ \text{in} & \\
 & \quad \texttt{if}\ H?\ x_1 & \\
 & \quad \texttt{then}\ \langle x, \mathit{merge}\ \sigma_2\ \sigma'\rangle & \\
 & \quad \texttt{else}\ \langle x, \sigma'\rangle &
\end{array}
$$

Value Translation Rules: $\boxed{[\![\bullet]\!]_{\texttt{val}} : w \ \rightarrow \ v}$

$$
\begin{array}{ll}
[\![(\lambda x.m)^L]\!]_{\texttt{val}} = (\lambda x\sigma.[\![m]\!]\ \sigma) & [\text{TR-LAM-LO}] \\
[\![(\lambda x.m)^H]\!]_{\texttt{val}} = H : (\lambda x\sigma.[\![m]\!]\ \sigma) & [\text{TR-LAM-HI}]
\end{array}
$$

Store Translation Rules: $\boxed{[\![\bullet]\!]_{\texttt{st}} : w \ \rightarrow \ v}$

$$[\![\Sigma]\!]_{\texttt{st}} = \langle [\![\Sigma(i)]\!]_{\texttt{val}}^{i \in 1..n}\rangle \qquad [\text{TR-STORE}]$$

Auxiliary Definition:

$$\mathit{merge} = (\lambda\sigma\sigma'.\langle(\texttt{if}\ H?\ (\sigma.i)\ \texttt{then}\ \sigma'.i\ \texttt{else}\ \sigma.i)^{i \in 1..n}\rangle)$$

The translation is defined in Figure 3 by three mutually recursive functions, $[\![m]\!]$, $[\![w]\!]_{\texttt{val}}$, and $[\![\Sigma]\!]_{\texttt{st}}$, which translate ImpFlow terms (m), values (w), and stores (Σ) into FunFlow. The translation $[\![\Sigma]\!]_{\texttt{st}}$ converts an ImpFlow store Σ into a FunFlow n-tuple by translating each contained value. The value translation $[\![(\lambda x.m)^k]\!]_{\texttt{val}}$ maps an ImpFlow value to a FunFlow function that adds an additional parameter σ for the threaded store, as follows:

$$(\lambda x\sigma.[\![m]\!]\ \sigma)$$

The value translation in turn calls the expression translation function $[\![m]\!]$. If the ImpFlow value is H-labeled, then the translation wraps the above FunFlow value in an enclosing $H\colon$ via the rule [TR-LAM-HI].

The expression translation function $[\![m]\!]$ converts an ImpFlow expression m into a FunFlow expression of the form $(\lambda\sigma.e)$, where σ represents the encoding of a store. The result of applying this function $(\lambda\sigma.e)$ to an encoded store results in a pair $\langle v, \sigma'\rangle$ where v represents a value $(\lambda x\sigma.e')$ and σ' represents the possibly modified store.

The [TR-VAR] rule is one of the simplest translations and illustrates the design. The function $[\![x]\!] = (\lambda\sigma.\langle x, \sigma\rangle)$ takes the store σ and returns a pair of a variable x and the unmodified store σ. ImpFlow functions are translated by the rule [TR-LAM] in a similar manner. The resulting FunFlow function takes an encoded store σ and returns a pair of a function and the store σ. The returned function $(\lambda x\sigma'.[\![m]\!]\ \sigma')$ applies the translated function body to store σ'.

The translation of $H\,{:}\,m$, via the [TR-HI] rule, takes the expression m resulting in the pair $\langle x, \sigma'\rangle$, and then returns the pair $\langle H : x, \sigma'\rangle$ where x is labeled as private. Critically, the store σ' is *not* labeled as private. Likewise, the translation of $H?\ m$ converts the expression m to a function that returns the pair $\langle x, \sigma'\rangle$ via the [TR-PRED] rule. It performs label reflection on x and then either returns (appropriately translated versions of) `true` or `false`.

The translations for operations on mutable reference cells are fairly straightforward. The translated function for dereferencing i returns the ith element of the store tuple, as indicated by the rule [TR-DEREF]. The [TR-ASSIGN] rule for $i\,{:=}\,m$ translates the right-hand side m, resulting in a function that will return a pair $\langle x, \sigma'\rangle$ when it is applied to store σ. This pair is returned, except that position i in σ' is replaced with x.

The translation of a function application $(m_1\ m_2)$ is subtle and requires some care. A naive approach is to thread the store through m_1, m_2, and the callee in the standard manner:

$$\begin{aligned}[\![m_1\ m_2]\!] = \lambda\sigma.&\texttt{let}\ \langle x_1, \sigma_1\rangle = [\![m_1]\!]\ \sigma\ \texttt{in}\\ &\texttt{let}\ \langle x_2, \sigma_2\rangle = [\![m_2]\!]\ \sigma_1\ \texttt{in}\\ &\texttt{let}\ \langle x, \sigma'\rangle = x_1\ x_2\ \sigma_2\ \texttt{in}\\ &\langle x, \sigma'\rangle\end{aligned} \qquad \text{[TR-APPLY-NAIVE]}$$

Unfortunately, this translation results in making the entire store private whenever a private function is applied, as discussed in the introduction. For example, $[\![(\lambda x.x)^H\ w]\!]\ \sigma \to^* H\,{:}\,\langle [\![w]\!]_{\texttt{val}}, \sigma\rangle$.

The [TR-APPLY] rule applies a more intricate translation in cases where the target function x_1 is private (that is, when $H?\ x_1$ is true). This translation determines the stores σ_2 and σ' before and after the call to x_1, and then combines these two stores via the auxiliary function *merge* $\sigma_2\ \sigma'$. Any entry $\sigma_2.i$ that is public should not be updated during the call, according to the no-sensitive-upgrade rule, so in this case $(merge\ \sigma_2\ \sigma').i$ returns $\sigma_2.i$. Conversely, for private entries $\sigma_2.i$, $(merge\ \sigma_2\ \sigma').i$ can safely return the private value $\sigma'.i$.

Note that this transformation ignores updates to public entries in the store in a manner that is somewhat analogous to the no-sensitive-upgrade rule, which forbids such updates. Both approaches guarantee non-interference. We can formalize the same behavior for the ImpFlow semantics as follows:

$$\frac{m, \Sigma \Downarrow_{pc} w, \Sigma' \qquad pc \not\sqsubseteq label(\Sigma'(i))}{i := m, \Sigma \Downarrow_{pc} w, \Sigma'}\ [\text{M-ASSIGN-IGNORE}]$$

Critically, the translation preserves *equivalent modulo under labels*.

Lemma 2. *If* $m_1 \sim_{ul} m_2$*, then* $[\![m_1]\!] \sim_{ul} [\![m_2]\!]$.

Consequently, if $m_1 \sim_{ul} m_2$ and

$$[\![m_1]\!]\ [\![\Sigma]\!]_{\texttt{st}} \to^* \langle v_1, \sigma_1 \rangle$$
$$[\![m_2]\!]\ [\![\Sigma]\!]_{\texttt{st}} \to^* \langle v_2, \sigma_2 \rangle$$

then $v_1 \sim_{ul} v_2$. That is, implementing ImpFlow via our translation to FunFlow preserves termination-insensitive non-interference.

It is not obvious how the semantics via translation to FunFlow corresponds to the big-step operational semantics of ImpFlow from Figure 3, particularly given the difference between the two languages: imperative vs. functional, universal vs. sparse labeling, and the no-sensitive-upgrade check vs. the *merge* function.

We aim to establish that the translation preserves semantics. We might conjecture a correspondence property of the form:

> If $m, \Sigma \Downarrow_L w, \Sigma'$ then $[\![m]\!][\![\Sigma]\!]_{\texttt{st}} \to^* \langle v, \sigma \rangle$ where $v = [\![w]\!]_{\texttt{val}}$ and $\sigma = [\![\Sigma']\!]_{\texttt{st}}$.

Unfortunately this correspondence does not hold, since v and $[\![w]\!]_{\texttt{val}}$ may have syntactic differences in labels that are not observable. That is, a label H that is nested under another label H is redundant. Therefore we define a $\sim_{lul}$ relation that determines if two expressions are *equivalent modulo labels under labels*. This relation relies on a $\sim_l$ relation, which compares two expressions and tests if they are *equivalent modulo labels*. The rules for both relations are given below.

$$\frac{}{x \sim_{lul} x} \quad [\text{BEQ-VAR}] \qquad \frac{}{x \sim_{l} x} \quad [\text{BEQ-H-VAR}]$$

$$\frac{e \sim_{lul} e'}{(\lambda x.e) \sim_{lul} (\lambda x.e')} \quad [\text{BEQ-FUN}] \qquad \frac{e \sim_{l} e'}{(\lambda x.e) \sim_{l} (\lambda x.e')} \quad [\text{BEQ-H-FUN}]$$

$$\frac{e \sim_{lul} e'}{H?\ e \sim_{lul} H?\ e'} \quad [\text{BEQ-PRED}] \qquad \frac{e \sim_{l} e'}{H?\ e \sim_{l} H?\ e'} \quad [\text{BEQ-H-PRED}]$$

$$\frac{e_1 \sim_{lul} e_1' \qquad e_2 \sim_{lul} e_2'}{e_1\ e_2 \sim_{lul} e_1'\ e_2'} \quad [\text{BEQ-APP}] \qquad \frac{e_1 \sim_{l} e_1' \qquad e_2 \sim_{l} e_2'}{e_1\ e_2 \sim_{l} e_1'\ e_2'} \quad [\text{BEQ-H-APP}]$$

$$\frac{e \sim_{l} e'}{H\!:\!e \sim_{lul} H\!:\!e'} \quad [\text{BEQ-LABEL}] \qquad \frac{e \sim_{l} e'}{e \sim_{l} H\!:\!e'} \quad [\text{BEQ-H-RIGHT}]$$

$$\frac{e \sim_{l} e'}{H\!:\!e \sim_{l} e'} \quad [\text{BEQ-H-LEFT}]$$

If two expressions in FunFlow are equivalent modulo labels under labels, then evaluation maintains this property, as formalized by the following lemma.

Lemma 3. *If $e_1 \sim_{lul} e_2 \to e_3$ then there exists e_4 such that $e_1 \to^* e_4 \sim_{lul} e_3$.*

The $\sim_{ul}$ relation includes strictly more terms than the $\sim_{lul}$ relation, since it accepts all terms that vary under H.

Lemma 4. $(\sim_{lul}) \subseteq (\sim_{ul})$

We define a new relation that combines the small-step evaluation relation for FunFlow with the equivalent modulo labels under labels relation. This relation $\Rightarrow$ enables us to add and remove redundant labels as necessary to obtain the desired correspondence sketched above.

$$(\Rightarrow) \quad \stackrel{\text{def}}{=} \quad (\to) \ \cup \ (\sim_{lul})$$

Lemma 5. *If $e_1 \Rightarrow^n e_2$ then there exists e_3 such that $e_1 \to^* e_3 \sim_{lul} e_2$.*

Evaluation in a high context does not affect public reference cells.

Lemma 6. *If $m, \Sigma \Downarrow_H w, \Sigma'$ then for all i such that $\Sigma(i) \neq \Sigma'(i)$, $\Sigma(i)$ and $\Sigma'(i)$ are both high.*

If a program does not modify public reference cells in private blocks of code, then the *merge* operation has no effect.

Lemma 7. *Suppose for all i with $\Sigma(i) \neq \Sigma'(i)$ we have that both $\Sigma(i)$ and $\Sigma'(i)$ are high. Then merge $[\![\Sigma]\!]_{\mathtt{st}}\ H\!:\![\![\Sigma']\!]_{\mathtt{st}} \Rightarrow^* [\![\Sigma']\!]_{\mathtt{st}}$.*

We now show the correctness of the translation. If a program of the source language executes successfully, then the translation also executes successfully and exhibits the same behavior.

Theorem 3. *If $m, \Sigma \Downarrow_{pc} w, \Sigma'$ then $\mathcal{E}[[\![m]\!]\ [\![\Sigma]\!]_{\mathtt{st}}] \Rightarrow^* \mathcal{E}[\langle [\![w]\!]_{\mathtt{val}}, [\![\Sigma']\!]_{\mathtt{st}} \rangle]$ for all $\mathcal{E} \in Context_{pc}$.*

Proof. The proof is provided in a related technical report [4].

7 Non-Interference in ImpFlow (Continued)

Showing the correctness of the translation and the non-interference property of FunFlow enables a short proof of non-interference for ImpFlow. This proof gives evidence that the translation is helpful and sensible. Albeit indirect, it is pleasingly simple. We now prove non-interference in the source language (Theorem 1). The proof relies on the correctness of the translation of ImpFlow programs that run to completion, and the non-interference property of the FunFlow semantics.

Proof (of Theorem 1). By Lemma 2, $[\![m_1]\!]\ [\![\Sigma]\!]_{\mathtt{st}} \sim_{ul} [\![m_2]\!]\ [\![\Sigma]\!]_{\mathtt{st}}$.
By Theorem 3, $[\![m_i]\!]\ [\![\Sigma]\!]_{\mathtt{st}} \Rightarrow^* \langle [\![w_i]\!]_{\mathtt{val}}, [\![\Sigma_i]\!]_{\mathtt{st}} \rangle$ for all i.
Hence by Lemma 5, for all i there exists v_i such that
$[\![m_i]\!]\ [\![\Sigma]\!]_{\mathtt{st}} \rightarrow^* v_i \sim_{lul} \langle [\![w_i]\!]_{\mathtt{val}}, [\![\Sigma_i]\!]_{\mathtt{st}} \rangle$.
By Theorem 2, $v_1 \sim_{ul} v_2$, so by Lemma 4, $[\![w_1]\!]_{\mathtt{val}} \sim_{ul} [\![w_2]\!]_{\mathtt{val}}$.
Note that $[\![\mathtt{true}^L]\!]_{\mathtt{val}} \not\sim_{ul} [\![\mathtt{false}^L]\!]_{\mathtt{val}}$, hence w_1 and w_2 are not distinct public Booleans, so $w_1 = w_2$. □

8 Conclusion

This paper aims to further understanding of dynamic information flow in a language with imperative updates through translation to a purely functional core. A naive translation is unnecessarily restrictive, but through careful handling of stores, a translation can preserve the flexibility of the source language. Reasoning about the translation and the non-interference properties of the target language enables a simple proof of non-interference in the source language.

Both the source language and the target language embody fairly standard ideas present in the literature. We hope that our results thus shed some light on other languages. Nevertheless, we recognize that not all variants of the languages will necessarily lend themselves easily to analogous treatments. The detailed consideration of those variants may be a worthwhile subject for further research.

Acknowledgements We would like to thank Gérard Boudol, Anindya Banerjee, Steve Zdancewicz, Andrew Myers, Greg Morrisett, Alejandro Russo, Jean-Jacques Lévy, Damien Doligez, and Cédric Fournet for helpful discussions. This work was supported by the NSF under grants CNS-0905650 and CCF-1116883.

References

1. Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Symposium on Principles of Programming Languages*, 1999.
2. Martín Abadi, Butler W. Lampson, and Jean-Jacques Lévy. Analysis and caching of dependencies. In *International Conference on Functional Programming*, 1996.
3. Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Workshop on Programming Languages and Analysis for Security*, 2009.

4. Thomas H. Austin, Cormac Flanagan, and Martín Abadi. A functional view of imperative information flow, extended version. Technical Report UCSC-SOE-12-15, The University of California at Santa Cruz, 2012.
5. Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *Conference on Programming Language Design and Implementation*, 2009.
6. Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.
7. Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
8. Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *Annual Computer Security Applications Conference*, 2009.
9. J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974.
10. FlowCaml homepage. `http://pauillac.inria.fr/~simonet/soft/flowcaml/`, accessed May 2010.
11. Gurvan Le Guernic, Anindya Banerjee, Thomas P. Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. In *Asian Computing Science Conference on Secure Software*, 2006.
12. Daniel Hedin and Andrei Sabelfeld. Information-flow security for a core of JavaScript. In *Computer Security Foundations Symposium*, 2012.
13. Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Symposium on Principles of Programming Languages*, 1998.
14. Jif homepage, 2010. http://www.cs.cornell.edu/jif/, accessed October 2010.
15. Maxwell N. Krohn, Petros Efstathopoulos, Cliff Frey, M. Frans Kaashoek, Eddie Kohler, David Mazières, Robert Morris, Michelle Osborne, Steve Vandebogart, and David Ziegler. Make least privilege a right (not a privilege). In *Workshop on Hot Topics in Operating Systems*, 2005.
16. Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, 1999.
17. François Pottier and Vincent Simonet. Information flow inference for ML. *Transactions on Programming Languages and Systems*, 25(1):117–158, 2003.
18. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
19. Paritosh Shroff, Scott F. Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *Computer Security Foundations Symposium*, 2007.
20. Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Symposium on Haskell*, 2011.
21. Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. 2007.
22. 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.
23. Stephan Arthur Zdancewic. *Programming languages for information security*. PhD thesis, Cornell University, 2002.
24. Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. *Communications of the ACM*, 54(11):93–101, 2011.