IFC Inside: A General Approach to Retrofitting Languages with Dynamic Information Flow Control

Stefan Heule Deian Stefan Edward Z. Yang John C. Mitchell Alejandro Russo
Stanford University, Chalmers University
{sheule,deian,ezyang,mitchell}@cs.stanford.edu, russo@chalmers.se

Abstract

Many important security problems in JavaScript, such as browser extension security, untrusted JavaScript libraries and safe integration of mutually distrustful websites (mash-ups), may be effectively addressed using an efficient implementation of information flow control. We formally specify a coarse-grained IFC system that can be implemented non-intrusively, resulting in much greater implementation efficiency than previous finer-grained approaches. The central concept is an isolation mechanism whose operation does not depend too much on the details of the programming language used. Thus, we have a *general* method for adding IFC to existing languages that is largely *indifferent* to the choice of language. Additionally, we describe how to relate optimized concrete implementations to the IFC systems produced by our technique and show how various proposed and existing IFC systems conform to our formal framework and therefore enjoy the semantic properties we prove.

1. Introduction

Modern web content is rendered using a potentially large number of different components with differing provenance. Disparate and untrusting components may arise from browser extensions (whose JavaScript code runs alongside website code), web applications (with possibly untrusted third-party libraries), and mashups (which combine code and data from websites that may not even be aware of each other's existence.) While just-in-time combination of untrusting components offers great flexibility, it also poses complex security challenges. In particular, maintaining data privacy in the face of malicious extensions, libraries, and mashup components has been difficult, and in general effectively impossible.

Information flow control (IFC) is a promising technique that provides security by tracking the flow of sensitive data through a system. Untrusted code is confined so that it cannot exfiltrate data, except as per an information flow policy. Significant research has been devoted to adding various forms of IFC to different kinds of programming languages and systems. In the context of the web, however, there is a strong motivation to preserve the language JavaScript, and retrofit dynamic information flow control on top of it. A key challenge in dynamic information flow control is implementation efficiency: when information flow is tracked at a fine-grained level, e.g., per value in the system, systems may incur impractical runtime overhead [26]. Moreover, the task of adding fine-grained IFC even to a simple language is non-trivial [29]; when considering a real-world language, such as JavaScript, this challenge becomes almost insurmountable.

In the alternative *coarse-grained* approach to IFC, a system is divided into relatively coarse computational units, each with a single label dictating its security policy. Only communication between isolated computational units must be tracked. A recent system named SWAPI [54] adds IFC to the browser by identifying *web workers* as an existing computational unit. Communication between workers is mediated using IFC to ensure non-interference, preventing sensitive workers from influencing the computation of less sensitive workers. This coarse-grained approach provides a number of advantages: (1) reduced runtime overhead because checks need only be performed at isolation boundaries, (2) minimal changes to an existing programming language when adding IFC, which (3) allows the reuse of existing programs. Finally, (4) associating a single security label with the entire computational unit simplifies understanding and reasoning about the correctness of the system, without information flow reasoning about the technical details of the semantics of the programming language.

A further benefit of the coarse-grained approach is that a single IFC design and semantics can be generalized to a large class of programming languages. Leveraging this insight, we define a general form of coarse-grained dynamic IFC and show how it can be combined with *any* programming language conforming to a general semantic framework. At the core of our IFC system is a sandboxing construct that isolates potentially sensitive computations, allowing them to communicate only through clearly specified interfaces. Using the Matthews-Findler approach [37] for combining operational semantics, we formally state and prove non-interference guarantees that are independent of the choice of specific target language.

One challenge in the design of our system is that the semantics for the retrofitted IFC language may not accurately correspond to the optimized implementation details of a particular target language. We discuss and partially address this issue by characterizing isomorphisms between the operational semantics of our defined language and a concrete implementation, showing that if this relationship holds, then non-interference in the abstract specification carries over to the concrete specification.

Our contributions can be summarized as follows:

- We give formal semantics for a core coarse-grained information flow control language. We then show how a large class of target languages can be combined with this IFC language and prove that the result provides non-interference. (Section 2)
- We provide a proof technique to show the non-interference of a concrete semantics for a potentially optimized IFC language by means of an isomorphism, and show a class of restrictions on the IFC language that preserves non-interference. (Section 3)

- We connect our formal semantics to real implementations of coarse-grained IFC systems for JavaScript and Haskell, and explain how our system could be used to enforce IFC in C. (Section 5)
- We briefly describe ways to enrich the core IFC language with more advanced concepts such as labeled values, privileges, or a notion of clearance. (Section 6)
- We informally relate our results to a monadic interpretation of IFC which guided our design. (Section 8)

2. Retrofitting Languages with IFC

Our overall plan will be to take a **target language** with a formal operational semantics and combine it with an *information flow control language*. For example, taking ECMAScript as the target language and combining it with our IFC language should produce the formal semantics for SWAPI. In this presentation, we use a simple, untyped lambda calculus with mutable references and fixpoint in place of ECMAScript, to demonstrate some of the key properties without having too much detail; we discuss the proper embedding in more detail in Section 5.

Notation We have typeset nonterminals of the target language using **bold font** while the nonterminals of the IFC language have been typeset with *italic font*. Readers are encouraged to view a color copy of this paper, where target language nonterminals are colored **red** while IFC language nonterminals are colored *blue*.

2.1 Target Language: Mini-ES

In Figure 1, we give a simple, untyped lambda calculus with mutable references and fixpoint, prepared for combination with an information flow control language. The presentation is mostly standard, and utilizes Felleisen-Hieb reduction semantics [20] to define the operational semantics of the system. One peculiarity is that our language defines an evaluation context $\mathbf{E}$, but, the evaluation rules have been expressed in terms of a different evaluation context $\mathcal{E}_{\mathbf{\Sigma}}$; Here, we follow the approach of Matthews and Findler [37] in order to simplify combining semantics of multiple languages. To derive the usual operational semantics for this language, the evaluation context merely needs to be defined as $\mathcal{E}_{\mathbf{\Sigma}}\,[\mathbf{e}] \triangleq \mathbf{\Sigma}, \mathbf{E}\,[\mathbf{e}]$. However, when we combine this language with an IFC language, we reinterpret the meaning of this evaluation context.

In general, we require that a target language be expressed in terms of some global machine state $\mathbf{\Sigma}$, some evaluation context $\mathbf{E}$, some expressions $\mathbf{e}$, some set of values $\mathbf{v}$ and a *deterministic* reduction relation on full configurations $\mathbf{\Sigma} \times \mathbf{E} \times \mathbf{e}$.

2.2 IFC Language

Most modern, dynamic information flow control languages encode policy by associating a *label* with data, which encodes whether or not data tagged with one label l_1 can flow to another label l_2 (written as $l_1 \sqsubseteq l_2$). Our embedding is agnostic to the choice of labeling scheme; we only require the labels to form a lattice [15] with the partial order $\sqsubseteq$, join $\sqcup$, and meet $\sqcap$. In this paper, we simply represent labels with the metavariable l, but do not discuss them in more detail.

As opposed to more traditional fine-grained language-based IFC systems that associate labels with values [5, 47], we follow [51, 65] in taking a coarse-grained floating-label approach and associating labels with *tasks*. The task label—we refer to the label of the currently executing task as the *current label*—serves to protect everything in the task's scope; all data in a task shares this common label. Hence before performing a reads or a write, the IFC monitor inspects the current label to decide whether the operation is permitted. A task can only write to entities that are at least as sensitive. Similarly, it can only read from entities that are less sensitive.

$$
\begin{array}{rl}
\mathbf{v} & ::= \lambda \mathbf{x}.\mathbf{e} \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{a} \\
\mathbf{e} & ::= \mathbf{v} \mid \mathbf{x} \mid \mathbf{e}\ \mathbf{e} \mid \mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{e}\ \mathbf{else}\ \mathbf{e} \\
 & \quad \mid \mathbf{ref}\ \mathbf{e} \mid !\mathbf{e} \mid \mathbf{e} := \mathbf{e} \mid \mathbf{fix}\ e \\
\mathbf{E} & ::= [\cdot]_{\mathbf{T}} \mid \mathbf{E}\ \mathbf{e} \mid \mathbf{v}\ \mathbf{E} \mid \mathbf{if}\ \mathbf{E}\ \mathbf{then}\ \mathbf{e}\ \mathbf{else}\ \mathbf{e} \\
 & \quad \mid \mathbf{ref}\ \mathbf{E} \mid !\mathbf{E} \mid \mathbf{E} := \mathbf{e} \mid \mathbf{v} := \mathbf{E} \mid \mathbf{fix}\ E \\
\mathbf{e}_1; \mathbf{e}_2 & \triangleq (\lambda \mathbf{x}.\mathbf{e}_2)\ \mathbf{e}_1\ \mathbf{where}\ \mathbf{x} \notin \mathcal{FV}\ (\mathbf{e}_2) \\
\mathbf{let}\ \mathbf{x} = \mathbf{e}_1\ \mathbf{in}\ \mathbf{e}_2 & \triangleq (\lambda \mathbf{x}.\mathbf{e}_2)\ \mathbf{e}_1
\end{array}
$$

$$\text{T-APP}\quad \frac{}{\mathcal{E}_{\mathbf{\Sigma}}\,[(\lambda x.\mathbf{e})\ \mathbf{v}] \rightarrow \mathcal{E}_{\mathbf{\Sigma}}\,[\{\mathbf{v}\ /\ x\}\ \mathbf{e}]}$$

$$\text{T-IFTRUE}\quad \frac{}{\mathcal{E}_{\mathbf{\Sigma}}\,[\ \mathbf{if}\ \mathbf{true}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2] \rightarrow \mathcal{E}_{\mathbf{\Sigma}}\,[\mathbf{e}_1]}$$

$$\text{T-IFFALSE}\quad \frac{}{\mathcal{E}_{\mathbf{\Sigma}}\,[\ \mathbf{if}\ \mathbf{false}\ \mathbf{then}\ \mathbf{e}_1\ \mathbf{else}\ \mathbf{e}_2] \rightarrow \mathcal{E}_{\mathbf{\Sigma}}\,[\mathbf{e}_2]}$$

$$\text{T-REF}\quad \frac{\mathrm{fresh}(\mathbf{a})}{\mathcal{E}_{\mathbf{\Sigma}}\,[\mathbf{ref}\ \mathbf{v}] \rightarrow \mathcal{E}_{\mathbf{\Sigma}[\mathbf{a}\mapsto\mathbf{v}]}\,[\mathbf{a}]} \qquad \text{T-DEREF}\quad \frac{(\mathbf{a}, \mathbf{v}) \in \mathbf{\Sigma}}{\mathcal{E}_{\mathbf{\Sigma}}\,[!\mathbf{a}] \rightarrow \mathcal{E}_{\mathbf{\Sigma}}\,[\mathbf{v}]}$$

$$\text{T-ASS}\quad \frac{}{\mathcal{E}_{\mathbf{\Sigma}}\,[\mathbf{a} := \mathbf{v}] \rightarrow \mathcal{E}_{\mathbf{\Sigma}[\mathbf{a}\mapsto\mathbf{v}]}\,[\mathbf{v}]}$$

$$\text{T-FIX}\quad \frac{}{\mathcal{E}_{\mathbf{\Sigma}}\,[\mathbf{fix}\ (\lambda x.e)] \rightarrow \mathcal{E}_{\mathbf{\Sigma}}\,[\{\mathbf{fix}\ (\lambda x.e)\ /\ x\}\ e]}$$

Figure 1. λ_{ES}: simple untyped lambda calculus extended with booleans, mutable references and general recursion. $\mathcal{FV}\ (\mathbf{e})$ returns the set of free variables in express $\mathbf{e}$.

However, as in other floating-label systems, this current label can be raised to allow the task to read from more sensitive entities at the cost of giving up the ability to write to certain entities.

In Figure 2, we give the syntax and *single-task* evaluation rules for a minimal information-flow control language. Ordinarily, information flow control languages are defined by directly stating a base language plus information flow control operators. In contrast, our language is purposely minimal: it does not have sequencing operations, control flow, or other constructs. However, it contains support for the following core information flow control features:

- First-class labels, with label values l as well as operations for computing on labels ($\sqsubseteq$, $\sqcup$ and $\sqcap$).
- Operations for inspecting (**getLabel**) and modifying (**setLabel**) the current label of the task (a task can only increase its label),
- Operations for non-blocking inter-task communication (**send** and **recv**), which interact with the global store of per-task message queues Σ, and
- A sandboxing operation used to spawn new isolated tasks. In concurrent settings **sandbox** corresponds to a fork-like primitive, whereas in a sequential setting, it more closely resembles the LIO's **toLabeled** [51], and the Breeze bracket [28].

These operations are all defined with respect to an evaluation context $\mathcal{E}_{\Sigma}^{i,l}$ that represents the context of the current task which, by conventions is running. The evaluation context has three important pieces of state: the global message queues Σ, the current label l and the task ID i.

We note that first class labels, tasks (albeit named differently), and operations for inspecting the current label are are essentially universal to all floating-label systems. However, our choice of communication primitives is motivated by those present in browsers, namely `postMessage` [58]. Of course, other choices, such as blocking communication or labeled channels, are possible (see Section 6).

These asynchronous communication primitives are worth further discussion. When a task is sending a message using **send**, it also labels that message with a label l' (which must be above the tasks current label l). Messages can only be received by a task if its current label is at least as high as the label of the message. Specifically, receiving a message using

$$\mathbf{recv}\ x_1, x_2\ \mathbf{in}\ e_1\ \mathbf{else}\ e_2$$

binds the message and the senders task identifier to local variables x_1 and x_2, respectively, and then executes e_1. Otherwise the message gets removed from the queue by rule I-NORECV, and that tasks continues its execution with e_2. We denote the filtering of the message queue by $\Theta \preceq l$, which is defined as follows. If Θ is the empty list ϵ, the function is simply the identity function, i.e., $\epsilon \preceq l = \epsilon$, and otherwise:

$$(l', i, e), \Theta \preceq l = \begin{cases} (l', i, e), \Theta \preceq l & \text{if } l' \sqsubseteq l \\ \Theta \preceq l & \text{otherwise} \end{cases}$$

This ensures that tasks cannot receive messages that are more sensitive than their current label would allow.

2.3 The Embedding

Figure 3 provides all of the rules responsible for actually carrying out the embedding of the IFC language with the target language. The most important feature of this embedding is that every task maintains its own copy of the target language global state and evaluation context, thus enforcing isolation between various tasks. In more detail:

- We extend the values, expressions and evaluation contexts of both languages to allow for embeddings terms in one language to be embedded in the other. [37] In the target language, an IFC expression appears as $_{\text{TI}}\lfloor e \rfloor$ ("target-outside, IFC-inside"); in the IFC language, a target language expression appears as $^{\text{IT}}\lceil \mathbf{e} \rceil$ ("IFC-outside, target-inside").
- We reinterpret $\mathcal{E}$ to be evaluation contexts on task lists, providing definitions for $\mathcal{E}_{\mathbf{\Sigma}}$ and $\mathcal{E}^{i,l}_{\Sigma}$. These rules only operate on the first task in the task list, which by convention is the only task executing.
- We reinterpret $\rightarrow$, an operation on a single task, in terms of $\hookrightarrow$, operation on task lists. The correspondence is simple: a task executes a step and then is rescheduled in the task list according to schedule policy α. Figure 4 defines two concrete schedulers.
- Finally, we define some rules for scheduling, handling sandboxing tasks (which interact with the state of the target language), and intermediating between the borders of the two languages.

The I-SANDBOX rule is used to create a new isolated task that executes separately from the existing tasks (and can be communicated with via **send** and **recv**). When the new task is created, there is the question of what the target language state of the new task should be. Our rule is stated generically in terms of a function κ. Conservatively, κ may be simply thought of as the identity function, in which case the semantics of sandbox are such that the state of the target language is *cloned* when sandboxing occurs. However, this is not necessary: it is also valid for κ to remove entries from the state. In Section 4, we give a more detailed discussion of the implications of the choice of κ, but all our security claims will hold regardless of the choice of κ.

$$\begin{array}{rcl} \otimes & ::= & \sqsubseteq \mid \sqcup \mid \sqcap \\ v & ::= & i \mid l \mid \mathbf{true} \mid \mathbf{false} \mid \langle\rangle \\ e & ::= & v \mid x \mid e \otimes e \mid \mathbf{getLabel} \mid \mathbf{setLabel}\ e \mid \mathbf{taskId} \\ & & \mid \mathbf{sandbox}\ e \mid \mathbf{send}\ e\ e\ e \mid \mathbf{recv}\ x, x\ \mathbf{in}\ e\ \mathbf{else}\ e \\ E & ::= & [\cdot]_I \mid E \otimes e \mid v \otimes E \mid \mathbf{setLabel}\ E \mid \mathbf{sandbox}\ E \\ & & \mid \mathbf{send}\ E\ e\ e \mid \mathbf{send}\ v\ E\ e \mid \mathbf{send}\ v\ v\ E \\ \theta & ::= & (l, i\ e) \\ \Theta & ::= & \epsilon \mid \theta, \Theta \\ \Sigma & ::= & \emptyset \mid \Sigma\,[i \mapsto \Theta] \end{array}$$

$$\text{I-GETTASKID}\ \frac{}{\mathcal{E}^{i,l}_{\Sigma}\,[\mathbf{taskId}] \rightarrow \mathcal{E}^{i,l}_{\Sigma}\,[i]} \qquad \text{I-GETLABEL}\ \frac{}{\mathcal{E}^{i,l}_{\Sigma}\,[\mathbf{getLabel}] \rightarrow \mathcal{E}^{i,l}_{\Sigma}\,[l]}$$

$$\text{I-SETLABEL}\ \frac{l \sqsubseteq l'}{\mathcal{E}^{i,l}_{\Sigma}\,[\mathbf{setLabel}\ l'] \rightarrow \mathcal{E}^{i,l'}_{\Sigma}\,[\langle\rangle]}$$

$$\text{I-SEND}\ \frac{l \sqsubseteq l' \qquad \Sigma\,(i') = \Theta \qquad \Sigma' = \Sigma\,[i' \mapsto (l', i, v), \Theta]}{\mathcal{E}^{i,l}_{\Sigma}\,[\mathbf{send}\ i'\ l'\ v] \rightarrow \mathcal{E}^{i,l}_{\Sigma'}\,[\langle\rangle]}$$

$$\text{I-RECV}\ \frac{(\Sigma\,(i) \preceq l) = \theta_1, ..., \theta_k, (l', i', v) \qquad \Sigma' = \Sigma\,[i \mapsto (\theta_1, ..., \theta_k)]}{\mathcal{E}^{i,l}_{\Sigma}\,[\mathbf{recv}\ x_1, x_2\ \mathbf{in}\ e_1\ \mathbf{else}\ e_2] \rightarrow \mathcal{E}^{i,l}_{\Sigma'}\,[\{v\ /\ x_1, i'\ /\ x_2\}\ e_1]}$$

$$\text{I-NORECV}\ \frac{\Sigma\,(i) \preceq l = \epsilon \qquad \Sigma' = \Sigma\,[i \mapsto \epsilon]}{\mathcal{E}^{i,l}_{\Sigma}\,[\mathbf{recv}\ x_1, x_2\ \mathbf{in}\ e_1\ \mathbf{else}\ e_2] \rightarrow \mathcal{E}^{i,l}_{\Sigma'}\,[e_2]}$$

$$\text{I-LABELOP}\ \frac{[\![l_1 \otimes l_2]\!] = v}{\mathcal{E}^{i,l}_{\Sigma}\,[l_1 \otimes l_2] \rightarrow \mathcal{E}^{i,l}_{\Sigma}\,[v]}$$

Figure 2. IFC language with all single-task operations.

The rule I-NOSTEP says something about configurations for which it is not possible to take a transition. The notation $c \overset{?}{\not\rightarrow}$ in the premise is meant to be understood as follows: If the configuration c cannot take a step by any rule other than I-NOSTEP, then I-NOSTEP applies and the stuck task gets removed.

Rules I-DONE and I-NOSTEP define the behavior of the system when the current thread has reduced to a value, or gotten stuck, respectively. While these definitions simply rely on the underlying scheduling policy, α, to modify the task list, as we describe in Sections 3 and 6, these rules (notably, I-NOSTEP) are crucial to proving our security guarantees. For instance, it is unsafe for the whole system to get stuck if a particular task gets stuck, since a sensitive thread may then leverage this to leak information through the termination channel. Instead, as our example RR scheduler shows, such tasks should simply be removed from the task list.

As in [37], rules T-BORDER and I-BORDER define the syntactic boundaries between the IFC and target languages. Intuitively, the boundaries respectively correspond to an upcall into and downcall from the IFC runtime. As an example, taking λ_{ES} as the target languge, we can now define a blocking receive (inefficiently) in terms of the asynchronous **recv** as series of cross-language calls:

$$
\begin{array}{lll}
v & ::= \cdots \mid {}^{\text{IT}}\lceil \mathbf{v} \rceil & \mathbf{v} ::= \cdots \mid {}_{\text{TI}}\lfloor v \rfloor \\
e & ::= \cdots \mid {}^{\text{IT}}\lceil \mathbf{e} \rceil & \mathbf{e} ::= \cdots \mid {}_{\text{TI}}\lfloor e \rfloor \\
E & ::= \cdots \mid {}^{\text{IT}}\lceil \mathbf{E} \rceil & \mathbf{E} ::= \cdots \mid {}_{\text{TI}}\lfloor E \rfloor
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}_{\mathbf{\Sigma}}\,[\mathbf{e}] \triangleq \Sigma; \langle \mathbf{\Sigma}, E[\mathbf{e}]_{\mathbf{T}} \rangle_l^i, \ldots \\
\mathcal{E}_{\Sigma}^{i,l}\,[e] \triangleq \Sigma; \langle \mathbf{\Sigma}, E[e]_I \rangle_l^i, \ldots \\
\mathcal{E}\,[e] \to \Sigma; t, \ldots \triangleq \mathcal{E}\,[e] \overset{\alpha}{\hookrightarrow} \Sigma; \alpha_{\text{step}}(t, \ldots)
\end{array}
$$

I-SANDBOX

$$
\frac{\Sigma' = \Sigma\,[i' \mapsto \epsilon] \qquad \mathbf{\Sigma}' = \kappa\,(\mathbf{\Sigma}) \qquad t_1 = \langle \mathbf{\Sigma}, E[i'] \rangle_l^i \qquad t_{\text{new}} = \langle \mathbf{\Sigma}', {}_{\text{TI}}\lfloor e \rfloor \rangle_l^{i'} \qquad \text{fresh}(i')}{\Sigma; \langle \mathbf{\Sigma}, E[\mathbf{sandbox}\ e]_I \rangle_l^i, \ldots \overset{\alpha}{\hookrightarrow} \Sigma'; \alpha_{\text{sandbox}}(t_1, \ldots, t_{\text{new}})}
$$

I-DONE

$$
\frac{}{\Sigma; \langle \mathbf{\Sigma}, v \rangle_l^i, \ldots \overset{\alpha}{\hookrightarrow} \Sigma; \alpha_{\text{done}}(\langle \mathbf{\Sigma}, v \rangle_l^i, \ldots)}
$$

I-NOSTEP

$$
\frac{\Sigma; t, \ldots \overset{\alpha}{\not\hookrightarrow}}{\Sigma; t, \ldots \overset{\alpha}{\hookrightarrow} \Sigma; \alpha_{\text{noStep}}(t, \ldots)}
$$

I-BORDER

$$
\frac{}{\mathcal{E}_{\Sigma}^{i,l}\left[{}^{\text{IT}}\lceil {}_{\text{TI}}\lfloor e \rfloor \rceil\right] \to \mathcal{E}_{\Sigma}^{i,l}\,[e]}
$$

T-BORDER

$$
\frac{}{\mathcal{E}_{\mathbf{\Sigma}}\left[{}_{\text{TI}}\lfloor {}^{\text{IT}}\lceil \mathbf{e} \rceil \rfloor\right] \to \mathcal{E}_{\mathbf{\Sigma}}\,[\mathbf{e}]}
$$

Figure 3. The embedding $L_{\text{IFC}}(\alpha, \lambda)$, where $\lambda = (\mathbf{\Sigma}, \mathbf{E}, \mathbf{e}, \mathbf{v}, \to)$

Concurrent, round robin:

$$
\begin{array}{l}
\text{RR}_{\text{step}}(t_1, t_2, \ldots) = t_2, \ldots, t_1 \\
\text{RR}_{\text{done}}(t_1, t_2, \ldots) = t_2, \ldots \\
\text{RR}_{\text{noStep}}(t_1, t_2, \ldots) = t_2, \ldots \\
\text{RR}_{\text{sandbox}}(t_1, t_2, \ldots) = t_2, \ldots, t_1
\end{array}
$$

Sequential:

$$
\begin{array}{ll}
\text{SEQ}_{\text{step}}(t_1, t_2, \ldots) & = t_1, t_2, \ldots \\
\text{SEQ}_{\text{noStep}}(t_1, t_2, \ldots) & = t_1, t_2, \ldots \\
\text{SEQ}_{\text{done}}(t) & = t \\
\text{SEQ}_{\text{done}}(t_1, t_2, \ldots) & = t_2, \ldots \\
\text{SEQ}_{\text{sandbox}}(t_1, t_2, \ldots, t_n) & = t_n, t_1, t_2, \ldots
\end{array}
$$

Figure 4. Scheduling policies

$$
\begin{array}{l}
\mathbf{blockingRecv}\ x_1, x_2\ \mathbf{in}\ e \triangleq \\
\quad {}^{\text{IT}}\lceil \mathbf{fix}\ (\lambda k.{}_{\text{TI}}\lfloor \mathbf{recv}\ x_1, x_2\ \mathbf{in}\ e\ \mathbf{else}\ {}^{\text{IT}}\lceil k \rceil \rfloor) \rceil
\end{array}
$$

We note that our embedding corresponds to the lump embedding of [37] where terms in one language are treated opaquely in the other. Indeed since our embedding is defined for an arbitrary language λ, we cannot do better. However, when we have more information about the target language, it is possible to add rules for a natural embedding of expressions that occur in both languages. In our specific example with λ_{ES}, we can choose to preserve booleans across language boundaries as follows:

$$
\frac{b \in \{\mathbf{true}, \mathbf{false}\}}{E[{}^{\text{IT}}\lceil b \rceil] \to E[b]} \qquad \frac{b \in \{\mathbf{true}, \mathbf{false}\}}{\mathbf{E}[{}_{\text{TI}}\lfloor b \rfloor] \to \mathbf{E}[b]}
$$

For any target language λ and scheduling policy α, this embedding defines an IFC language, which we will refer to as $L_{\text{IFC}}(\alpha, \lambda)$.

2.4 Examples

To get a feeling for how these operational semantics operate, we describe a few sample programs, and their corresponding behavior.

Sending mutable references As each task maintains its own global state for the target language, one might reasonably wonder what would happen if you attempted to send a reference to another task.

$$
\begin{array}{l}
\mathbf{let}\ i = {}_{\text{TI}}\lfloor \mathbf{sandbox}\ (\mathbf{blockingRecv}\ x, i'\ \mathbf{in}\ {}^{\text{IT}}\lceil\, !\, {}_{\text{TI}}\lfloor x \rfloor \rceil) \rfloor \\
\mathbf{in}\ {}_{\text{TI}}\lfloor \mathbf{send}\ {}^{\text{IT}}\lceil i \rceil\ \mathbf{getLabel}\ {}^{\text{IT}}\lceil \mathbf{ref}\ \mathbf{true} \rceil \rfloor
\end{array}
$$

It would be a violation of information flow if such an operation actually resulted in references being shared between two tasks. In fact, the address will be received by the sandboxed task, but it will be a dangling pointer; the task will get stuck (under the RR scheduler the I-NOSTEP will in turn remove it from the task list). Readers who are curious whether or not a static discipline could be enforced to prevent these dangling pointers should check Section 6.

Status of the heap after a sandbox Consider a program whose semantics depends on the definition of κ:

$$
\begin{array}{l}
\mathbf{let}\ x = \mathbf{ref}\ \mathbf{true} \\
\mathbf{in}\ {}_{\text{TI}}\lfloor \mathbf{sandbox}\ {}^{\text{IT}}\lceil\, !\, x; !x \rceil \rfloor; x := \mathbf{false}
\end{array}
$$

When κ is the identity, the sandboxed task successfully dereferences the contents of the reference to be **true** twice (recall that references are cloned into the new global state). When κ is the constant function to the empty store, x is a dangling pointer.

Exceptions One language feature that has caused difficulty for many information flow control systems is handling exceptions. The key difficulty appears when considering the propagation of exceptions across stack boundaries where the current label is restored, e.g., from secret to public, as done by LIO's **toLabeled** and Breeze's brackets [28, 53]. If the exception crosses this boundary, it can leak information about the inner secret computation into a public computation; hence, both systems explicitly ensure that **toLabeled** does not propagate exceptions.

Our general embedding approach provides these semantics by construction. To show this, we first define an encoding of the sequential **toLabeled** construct in our framework:

$$
\begin{array}{l}
\mathbf{toLabeled}\ l\ \mathbf{e} \triangleq \\
\quad \mathbf{let}\ i = \mathbf{taskId}; \\
\quad \mathbf{let}\ j = {}_{\text{TI}}\lfloor \mathbf{sandbox}\ {}^{\text{IT}}\lceil {}_{\text{TI}}\lfloor \mathbf{setLabel}\ l \rfloor; {}_{\text{TI}}\lfloor \mathbf{send}\ i\ l\ {}^{\text{IT}}\lceil \mathbf{e} \rceil \rfloor \rceil \rfloor \\
\quad \mathbf{in}\ (\lambda\_.{}_{\text{TI}}\lfloor \mathbf{setLabel}\ l \rfloor; {}_{\text{TI}}\lfloor \mathbf{recv}\ x, \_\ \mathbf{in}\ x\ \mathbf{else}\ \langle\rangle \rfloor)
\end{array}
$$

When using the sequential scheduler SEQ, the sandboxed task runs to completion before returning to the task that sandboxed it. (Here, a task list can be interpreted as a number of information flow aware stack frames.) To read the result of a **toLabeled** operation (which is communicated via message passing), the parent task must raise its label; otherwise the label will get dropped, as per the I-RECV rule.[1]

Now suppose that our λ_{ES} is extended with a single exception expression **fail**, which reduces to **fail**, removing all containing $\mathbf{E}$ evaluation contexts, i.e., it propagates up to a ${}^{\text{IT}}\lceil - \rceil$ boundary. In this setting, it is clear that exceptions cannot cross boundaries. For instance,

$$
{}_{\text{TI}}\lfloor \mathbf{toLabeled}\ l\ {}^{\text{IT}}\lceil\ \mathbf{if}\ \mathbf{e}\ \mathbf{then}\ \mathbf{fail}\ \mathbf{else}\ \mathbf{true} \rceil \rfloor; \mathbf{true}
$$

[1] A keen reader may note that with a single, per-task channel, we might mix up our messages. Indeed, a proper development requires the ability to allocate fresh channels; for purposes of simplicity, this has been omitted.

always evaluates to `true`, regardless of the value of $\mathbf{e}$. Just like in both LIO and Breeze, however, only when calling `recv` on such messages will the exception be propagated. This of course, is not a leak since the current label must be raised before receiving sensitive message, regardless of their content.

3. Security Guarantees

We are interested in proving non-interference about many programming languages. This requires an appropriate definition of this notion that is language agnostic, so in this section, we present a few general definitions for what an information flow control language is and what non-interference properties it may have. We also state theorems about our system which we prove in Section 7. In particular, we show that $L_{\text{IFC}}(\alpha, \lambda)$, with an appropriate scheduler α, satisfies non-interference [23], without making any reference to properties of λ.

3.1 Erasure Function

When defining the security guarantees of an information flow control, we must characterize what the *secret inputs* of a program are. Like other work [35, 46, 51, 52], we specify and prove non-interference using *term erasure*. Intuitively, term erasure allows us to show that an attacker does not learn any sensitive information from a program if the program behaves identically (from the attackers point of view) to a program with all sensitive data "erased". To interpret a language under information flow control, we define a function ε_l that performs erasures by mapping configurations to erased configurations, usually by rewriting (parts of) configurations that are more sensitive than l to a new syntactic construct $\bullet$. We can the define an information flow control language as follows:

Definition 1 (Information flow control language)**.** An information flow control language $\mathbf{L}$ is a tuple $(\Delta, \hookrightarrow, \varepsilon_l)$, where Δ is the type of machine configurations (members of which are usually denoted by the metavariable c), $\hookrightarrow$ is a reduction relation between machine configurations and $\varepsilon_l : \Delta \to \varepsilon(\Delta)$ is an erasure function parametrized on labels from machine configurations to *erased* machine configurations $\varepsilon(\Delta)$. Sometimes, we use V to refer to set of terminal configurations in Δ, i.e., configurations where no further transitions are possible.

Our language $L_{\text{IFC}}(\alpha, \lambda)$ fulfills this definition as $(\Delta, \overset{\alpha}{\hookrightarrow}, \varepsilon_l)$, where $\Delta = \Sigma \times \text{List}(t)$. The set of terminal conditions V is $\Sigma \times t_V$, where $t_V \subset t$ is the type for tasks whose expressions have been reduced to values.[2] The erased configuration $\varepsilon(\Delta)$ extends Δ with configurations containing $\bullet$, and Figure 5 gives the precise definition for our erasure function ε_l. Essentially, a task and its corresponding message queue is completely erased from the task list if its label does not flow to the attacker observation level l. Otherwise, we apply the erasure function homomorphically and remove any message from the task's message queue that are more sensitive than l.

The definition of an erasure function is quite important: it captures the attack model, stating what can and cannot be observed by the attacker. In our case, we assume that the attacker cannot observe sensitive tasks or messages, or even the number of such entities. While such assumptions are standard [12, 52], our definitions allow for stronger attackers that may be able to inspect resource usage.[3]

[2] Here, we abuse notation by describing types for configuration parts using the same metavariables as the "instance" of the type, e.g., t for the type of task.

[3] We believe that we can extend $L_{\text{IFC}}(\alpha, \lambda)$ to such models using the resource limits techniques of [61]. We leave this extension to future work.

$$\varepsilon_l(\Sigma; ts) = \varepsilon_l(\Sigma); \text{filter } (\lambda t.t = \bullet) \text{ (map } \varepsilon_l \ ts)$$

$$\langle \Sigma, e \rangle^i_{l'} \begin{cases} \bullet & l' \not\sqsubseteq l \\ \langle \Sigma, e \rangle^i_{l'} & \text{otherwise} \end{cases}$$

$$\varepsilon_l(\emptyset) = \emptyset$$

$$\varepsilon_l(\Sigma\,[i \mapsto \Theta]) = \begin{cases} \varepsilon_l(\Sigma) & l' \not\sqsubseteq l, \text{ where } l' \text{ is} \\ & \text{the label of thread } i \\ \varepsilon_l(\Sigma)\,[i \mapsto \varepsilon_l(\Theta)] & \text{otherwise} \end{cases}$$

$$\varepsilon_l(\Theta) = \Theta \preceq l$$

$$\varepsilon_l(v) = v$$

Figure 5. Erasure function for tasks, queue maps, message queues, and configurations. In all other cases, including target-language constructs, ε_l is applied homomorphically.

3.2 Non-Interference

Given an information flow control language, we can now define non-interference. Intuitively, we want to make statements about the attacker's observational power at some security level l. This is done by defining an equivalence relation called l-equivalence on configurations: an attacker should not be able to distinguish two configuration that are l-equivalent. Since our erasure function captures what an attacker can or cannot observe, we simply define this equivalence as the syntactic-equivalence of erased configurations [52].

Definition 2 (l-equivalence)**.** In a language $(\Delta, \hookrightarrow, \varepsilon_l)$, two machine configurations $c, c' \in \Delta$ are considered l-equivalent, written as $c \approx_l c'$, if $\varepsilon_l(c) = \varepsilon_l(c')$.

We can now state that a language satisfies non-interference if an attacker at level l cannot distinguish the runs of any two l-equivalent configurations. This particular property is called termination sensitive non-interference (TSNI). Besides the obvious requirement to not leak secret information to public channels, this definition also requires the termination of public tasks to be independent of secret tasks. Formally, we define TSNI as follows:

Definition 3 (Termination Sensitive Non-Interference (TSNI))**.** A language $(\Delta, \hookrightarrow, \varepsilon_l)$ satisfies termination sensitive non-interference if for any label l, and configurations $c_1, c_1', c_2 \in \Delta$, if

$$c_1 \approx_l c_2 \qquad \text{and} \qquad c_1 \hookrightarrow^* c_1' \tag{1}$$

then there exists a configuration $c_2' \in \Delta$ such that

$$c_1' \approx_l c_2' \qquad \text{and} \qquad c_2 \hookrightarrow^* c_2' \,. \tag{2}$$

In other words if we take two l-equivalent configurations, then for every intermediate step taken by the first configuration, there is a corresponding number of steps that the second configuration can take to result in a configuration that is l-equivalent to the first resultant configuration.

Our IFC language satisfies TSNI under the round-robin scheduler RR given in Figure 4.

Theorem 1 (Concurrent IFC language is TSNI)**.** *For any target language λ, $L_{IFC}(\text{RR}, \lambda)$ satisfies TSNI.*

In general, however, non-interference will not hold for an arbitrary scheduler α. For example, $L_{\text{IFC}}(\alpha, \lambda)$ with a scheduler that inspects a sensitive tasks current state when deciding which task to schedule next will in general break non-interference [6, 45].

However, even non-adversarial schedulers are not always safe. Consider, for example, the sequential scheduling policy SEQ given in Figure 4. It is easy to show that $L_{\text{IFC}}(\text{SEQ}, \lambda)$ does not satisfy TSNI: consider a target language similar to λ_{ES} with an additional

expression terminal $\Uparrow$ that denotes a divergent computation, i.e., $\Uparrow$ always reduces to $\Uparrow$ and a simple label lattice $\{\mathsf{pub}, \mathsf{sec}\}$ such that $\mathsf{pub} \sqsubseteq \mathsf{sec}$, but $\mathsf{sec} \not\sqsubseteq \mathsf{pub}$. Consider the following two configurations in this language:

$$c_1 = \Sigma; \langle \mathbf{\Sigma}_1, {}^{\mathrm{IT}}\lceil \textbf{if false then } \Uparrow \textbf{ else true} \rceil \rangle^1_{\mathsf{sec}}, \langle \mathbf{\Sigma}_1, e \rangle^2_{\mathsf{pub}}$$
$$c_2 = \Sigma; \langle \mathbf{\Sigma}_1, {}^{\mathrm{IT}}\lceil \textbf{if true then } \Uparrow \textbf{ else true} \rceil \rangle^1_{\mathsf{sec}}, \langle \mathbf{\Sigma}_2, e \rangle^2_{\mathsf{pub}}$$

These two configurations are pub-equivalent, but c_1 will reduce (in two steps) to $c_1 = \Sigma; \langle \mathbf{\Sigma}_1, {}^{\mathrm{IT}}\lceil \textbf{true} \rceil \rangle^2_{\mathsf{pub}}$, whereas c_2 will not make any progress. Suppose that e is a computation that writes to a pub channel,[4] then the sec task's decision to diverge or not is directly leaked to a public entity.

To accommodate for sequential languages, or cases where a weaker guarantee is sufficient, we consider an alternative non-interference property called termination insensitive non-interference (TINI). This property can also be upheld by sequential languages at the cost of leaking through (non)-termination [4].

Definition 4 (Termination insensitive non-interference (TINI))**.** A language $(\Delta, V, \hookrightarrow, \varepsilon_l)$ is termination insensitive non-interfering if for any label l, and configurations $c_1, c_2 \in \Delta$ and $c_1', c_2' \in V$, it holds that

$$(c_1 \approx_l c_2 \;\wedge\; c_1 \hookrightarrow^* c_1' \;\wedge\; c_2 \hookrightarrow^* c_2') \implies c_1' \approx_l c_2'$$

TSNI states that if we take two l-equivalent configurations, and both configurations reduce to final configurations (i.e., configurations for which there are no possible further transitions), then the end configurations are also l-equivalent. We highlight that this statement is much weaker than TSNI: it only states that terminating programs do not leak sensitive data, but makes no statement about non-terminating programs.

As shown by compilers [42, 48], interpreters [26], and libraries [46, 51], TINI is useful for sequential settings. In our case, we show that our IFC language with the sequential scheduling policy SEQ satisfies TINI.

Theorem 2 (Sequential IFC language is TINI)**.** *For any target language λ, $L_{IFC}(\text{SEQ}, \lambda)$ satisfies TINI.*

4. Isomorphisms and Restrictions

The operational semantics we have defined in the previous section satisfy non-interference by design. We achieve this general statement that works for a large class of languages by having different tasks executing completely isolated from each other, such that every task has its own state. In some cases, this strong separation is desirable, or even necessary. Languages like C provide direct access to memory locations without mechanisms in the language to achieve a separation of the heap. On the other hand, for other languages this strong isolation of tasks can be undesirable, e.g., for performance reasons. For instance, for the language λ_{ES}, our presentation so far requires a separate heap per task, which is not very practical. Instead, we would like to more tightly couple the integration of the target and IFC language by reusing existing infrastructure. In the running example, a concrete implementation might use a single global heap. More precisely, instead of using a configuration of the form

$$\Sigma; \langle \mathbf{\Sigma}_1, e_1 \rangle^{i_1}_{l_1}, \langle \mathbf{\Sigma}_2, e_2 \rangle^{i_2}_{l_2} \;\ldots$$

we would like a single global heap as in

$$\Sigma; \mathbf{\Sigma}; \langle e_1 \rangle^{i_1}_{l_1}, \langle e_2 \rangle^{i_2}_{l_2}, \ldots$$

If the operational rules are adapted naïvely to this new setting, then non-interference can be violated. In particular, it is not sound to share references to a heap cell between different tasks. Otherwise, information can be trivially leaked by copying secret data to a shared reference in one task, and then reading it again by a another task with a lower label. What we would like is a way of characterizing safe modifications to the semantics which preserve non-interference. The intention of our single heap implementation was to be more efficient by using one global heap, but *conceptually maintain isolation between task by not allowing sharing of references between them.* This intuition of having a different (potentially more efficient) concrete semantics that behave like the abstract semantics can be formalized by the following definition:

Definition 5 (Isomorphism of information-flow control languages)**.** A language $(\Delta, \hookrightarrow, \varepsilon_l)$ is *isomorphic* to a language $(\Delta', \hookrightarrow', \varepsilon_l')$ if there exist total functions $f : \Delta \to \Delta'$ and $f^{-1} : \Delta' \to \Delta$ such that $f \circ f^{-1} = id_\Delta$ $f^{-1} \circ f = id_{\Delta'}$. Furthermore, f and f^{-1} are functorial (e.g. if $x'\ R'\ y'$ then $f(x')\ R\ f(y')$) over both l-equivalences and $\hookrightarrow$.

If we weaken this restriction such that f^{-1} does not have to be functorial over $\hookrightarrow$, we call the language $(\Delta, \hookrightarrow, \varepsilon_l)$ *weakly isomorphic* to $(\Delta', \hookrightarrow', \varepsilon_l')$.

Providing an isomorphism between the two languages allows us to preserve (termination sensitive or insensitive) non-interference as the following two theorems state.

Theorem 3 (Isomorphism preserves termination sensitive non-interference)**.** *If L is isomorphic to L' and L' satisfies TSNI, then L satisfies TSNI.*

Proof. Shown by transporting configurations and reduction derivations from L to L', applying TSNI, and then transporting the resulting configuration, l-equivalence and multi-step derivation back. □

Theorem 4 (Weak isomorphism preserves termination insensitive non-interference)**.** *If a language L is weakly isomorphic to a language L', and L' satisfies TINI, then L satisfies TINI.*

Proof. Shown by transporting configurations and reduction derivations from L to L', applying TINI and transporting the resulting equivalence back using functoriality of f^{-1} over l-equivalences. □

Unfortunately, an isomorphism is often too strong of a requirement. To obtain an isomorphism with our single heap semantics, we need to mimic the behavior of several heaps with a single actual heap. The interesting cases are when we sandbox an expression and when messages are sent and received. The rule for sandboxing is parametrized by the strategy κ (cf. Section 2), which defines what heap the new task should execute with. We have considered two choices:

- When we sandbox into an empty heap, existing addresses in the sandboxed expression are no longer valid and the task will get stuck (and then removed by I-NOSTEP). Thus, we must rewrite the sandboxed expression so that all addresses point to fresh addresses guaranteed to not occur in the heap. Similarly, sending a memory address should be rewritten.
- When we clone the heap, we have to create a copy of everything reachable from the sandboxed expression (not just the values for the memory addresses in the expression), and replace all addresses correspondingly. Even worse, the behavior of sending a memory address now depends on whether that address existed at the time the receiving task was sandboxed; if it did, then the address should be rewritten to the new one, otherwise to a fresh address.

[4] Though we do not model labeled channels, extending the calculus with such a feature is straight forward, see Section 6.

This complicated behavior is counter to our initial motivation of implementing a single heap for efficiency.

4.1 Restricting the IFC Language

A better solution is to forbid sandboxed expressions as well as messages sent to other tasks to contain memory addresses in the first place. In a statically typed language, the type system could prevent this from happening, and we talk more about how our approach integrates with typed languages in Section 6.2. In a dynamically type languages such as λ_{ES}, we might restrict the transition for **sandbox** and **send** to only allow expressions without memory addresses.

While this sounds plausible, it is worth noting that we are modifying the IFC language semantics, which raises the question whether non-interference is preserved. This question can be subtle: it is easy to remove a transition from a language and invalidate TSNI. Intuitively if the restriction depends on secret data, then a public thread can observe if some other task terminates or not, and by that obtain information about the secret data that was used to restrict the transition. With this in mind, we require semantic rules to get restricted only based on information observable by the task triggering them. This ensures that non-interference is preserved, as the restriction does not depend on confidential information. Below, we give the formal definition of this condition for the abstract IFC language $L_{\text{IFC}}(\alpha, \lambda)$.

Definition 6 (Restricted IFC language)**.** For a family of predicates $\mathcal{P}$ (one for every reduction rule), we call $L^{\mathcal{P}}_{\text{IFC}}(\alpha, \lambda)$ a restricted IFC language if its definition is equivalent to the abstract language $L_{\text{IFC}}(\alpha, \lambda)$, with the following exception: The reduction rules are restricted by adding a predicate $P \in \mathcal{P}$ to the premise of all rule other than I-NOSTEP. Furthermore, the predicates P can depend only on the *erased* configuration $\varepsilon_l(c)$, where l is the label of the first task in the task list and c the full configuration.

By the following theorem, the restricted IFC language with an appropriate scheduling policy is non-interfering.

Theorem 5. *For any target language λ and family of predicates $\mathcal{P}$, the restricted IFC language $L^{\mathcal{P}}_{IFC}(\text{RR}, \lambda)$ is TSNI. Furthermore, the IFC language $L^{\mathcal{P}}_{IFC}(\text{SEQ}, \lambda)$ is TINI.*

4.2 IFC Language with a Single Heap

We are now ready to make our single heap IFC language precise and ensure its non-interference using the techniques presented. First, we can construct the restricted language $L^{\mathcal{P}_{\text{norefs}}}_{\text{IFC}}(\alpha, \lambda_{\text{ES}})$, where $\mathcal{P}_{\text{norefs}}$ is the family of always valid predicates, except for the ones for I-SANDBOX and I-SEND, which we define as

$$P(e) = (\mathcal{AV}(e) = \emptyset)$$

where $\mathcal{AV}(e)$ denotes the set of address variables in e. That is, we do not restrict any rules except for I-SANDBOX and I-SEND. Since P only depends on e, which is part of the current task and thus never erased w.r.t. the label of the first task, this language satisfies non-interference by Theorem 5.

The essential parts of the semantics for the concrete language with a single heap, which we call $L^{\text{Heap}}_{\text{IFC}}(\alpha)$, are given in Figure 6. Most rules are straight-forward translations of the rules in Figures 2 and 3 but for a single heap. For conciseness, we only show the interesting ones. Now, we can show an isomorphism between this language and $L^{\mathcal{P}_{\text{norefs}}}_{\text{IFC}}(\alpha, \lambda_{\text{ES}})$, which (by Theorem 3 and 4) guarantees non-interference for an appropriate scheduling policy α.

To this end, we represent addresses in the concrete language as pairs $(i, \mathbf{a})$ where i is a task identifier, and $\mathbf{a}$ an address in the

C-SANDBOX

$$\frac{\mathcal{AV}(e) = \emptyset \qquad \Sigma' = \Sigma\left[i' \mapsto \epsilon\right] \qquad t_1 = \langle E[i']\rangle^{i}_{l} \qquad t_{\text{new}} = \langle \text{TI}\lfloor e \rfloor\rangle^{i'}_{l} \qquad \text{fresh } (i')}{\Sigma; \mathbf{\Sigma}; \langle E[\mathbf{sandbox}\ e]_I\rangle^{i_1}_{l_1}, \ldots \hookrightarrow \Sigma'; \mathbf{\Sigma}; \alpha_F(t_1, \ldots, t_{\text{new}})}$$

C-SEND

$$\frac{\mathcal{AV}(e) = \emptyset \qquad l \sqsubseteq l' \qquad \Sigma\ (i') = \Theta \qquad \Sigma' = \Sigma\left[i' \mapsto (l', i, v), \Theta\right]}{\Sigma; \mathbf{\Sigma}; \langle E[\mathbf{send}\ i'\ l'\ v]_I\rangle^{i}_{l}, \ldots \to \Sigma; \mathbf{\Sigma}; \alpha_{\text{step}}(\langle\langle\rangle\rangle^{i}_{l}, \ldots)}$$

Figure 6. A selection of the reduction rules for $L^{\text{Heap}}_{\text{IFC}}(\alpha)$.

abstract system[5]. We also formulate the following well-formedness condition for configurations:

$$\text{wf}(c) = \forall \langle e\rangle^{i}_{l} \in c.\ \{(i', e') \in \mathcal{AV}(e) \mid i \neq i'\} = \emptyset$$

Essentially, every address in a given task must have the correct identifier as the first part of the address. It is easy to see that the initial configuration satisfies this condition, and any step in the concrete semantics preserves the condition. Therefore, we only need to consider well-formed configurations, which allows us to give the two required functions f and f^{-1} for the isomorphism. For conciseness, we only give the interesting parts of their definition, and leave the straight-forward proof that they actually provide an isomorphism.

- Addresses can be directly translated with $f((i, \mathbf{a})) = \mathbf{a}$, and $f^{-1}(\mathbf{a}) = (i, \mathbf{a})$ for an address $\mathbf{a}$ that occurs in task i.
- f splits the single heap into multiple heaps based on the i of the addresses. f^{-1} produces a single heap by translating the addresses and collapsing everything to a single store.

5. Real World Languages

In the introduction, we stated that the original motivation behind developing this embedding was in order to formally specify a coarse-grained information flow control language for JavaScript. In Section 2, however, we used only a simplified language as an example target language. In this section, we consider the application of our embedding to JavaScript and two other real world target languages, C and Haskell. Our goal is to show the flexibility of the embedding in settings with vastly different features and properties, as well as discuss the semantic gap that must be overcome in some cases to apply our formalism to other systems. Two of the systems we describe have been implemented: the Haskell system [51] and the SWAPI [54] system; we leave the implementation of the IFC system for C to future work.

5.1 C

C programs are able to execute arbitrary (machine) code, access arbitrary memory, and perform arbitrary syscalls. Thus, the confinement of C programs must be imposed by the underlying OS and hardware. For instance, this isolation can be achieved using Dune's hardware protection mechanisms [8], similar to a simpler version of Wedge [8, 11] with an information flow control policy. Using page tables, a (trusted) IFC runtime could ensure that each task, implemented as a lightweight process, can only access the memory it allocates–tasks do not have access to any shared memory. In addition, ring protection could be used to intercept syscalls performed

[5] Note that this does not make the isomorphism trivial, as in the single heap, there is nothing preventing task 1 to access an address (2,**a**). Furthermore, it is common to represent addresses in this way for efficient garbage collection of dead tasks.

by a task and only permit those corresponding to our IFC language (such as **getLabel** or **send**). Dunes hardware protection mechanism allows us to provide a concrete implementation that is efficient and relatively simple to reason about, but other sandboxing mechanisms could be used in place of Dune.

In this setting, the combined language of Section 2 can be interpreted in the following way: calling from the target language to the IFC language corresponds to invoking a syscall. Creating a new task with the **sandbox** syscall is corresponds to *forking* a process. Here, page tables limit what memory a task can access: we can ensure there will be no memory (effectively defining $\kappa\ (\mathbf{\Sigma}) = \mathbf{\Sigma}_0$, where $\mathbf{\Sigma}_0$ is the set of pages necessary to bootstrap a lightweight process.) Similarly, control over page tables and protection bits allows us to define a **send** syscall that copies pages to our (trusted) runtime queue; and, correspondingly, a **recv** that copies the pages from the runtime queue to the (untrusted) receiver. Since C is not memory safe, conditions on these syscalls are meaningless.

5.2 Haskell

In contrast with the C embedding which relies on hardware protection mechanisms, a Haskell implementation can leverage Haskell's strong data abstraction and static type system, monadic approach to effects, and lightweight concurrency to implement the embedding in a more lightweight manner. We briefly describe one such system, LIO [51], which implements IFC as a library.

LIO is implemented by defining a new monad, `LIO`, which wraps Haskell's `IO` monad. The purpose of this monad is twofold: it restricts the use of arbitrary effects that would ordinarily be allowed by the `IO` monad, and it associates labels with tasks. Computations in the `LIO` monad can be thought to be operating within the IFC system. One important aspect of using `IO` as the base for this implementation is that it allows use of Haskell's efficient implementation of threads, channels, etc. (e.g., in the concurrent version of LIO, we use Haskell's `forkIO` to fork a lightweight thread in the case of **sandbox** [52]), in contrast to defining them in a completely pure fashion (as suggested in Section 8).

What is the interpretation of this system as per Section 2? Here, the *pure subset* of Haskell is the target language, while the monadic subset of Haskell in the `LIO` monad is the IFC language. Crucially, the lack of unrestricted mutation in the pure fragment of Haskell prevents direct communication between tasks, even when memory is shared between them.[6]

Since this IFC system is implemented as a library in Haskell, implementation-wise, we must ensure that these languages are indeed the subsets of Haskell we claim, i.e., while the concrete language is all of Haskell, we must ensure that we can restrict programs to our subset of Haskell that encodes the combined language. To this end, `LIO` relies on Haskell's strong data abstraction and type system (as enforced by Safe Haskell [55]) to ensure that arbitrary `IO` actions cannot be lifted into `LIO`. In other words, assuming `LIO` is implemented correctly, programs written in the `LIO` monad cannot perform arbitrary `IO` actions without breaking abstraction.

We refer the interested reader to [51, 52] for additional details on the various implementations of this system.[7]

5.3 JavaScript

JavaScript (JS), as specified by ECMAScript [18], does not have any built-in functionality for I/O. Capabilities such as the ability to mutate the DOM and read user input are APIs defined on top of the ECMAScript specification. Consequently, and in contrast to our C and Haskell embeddings—for which we must eliminate external effects—the embedding for JS is trivial.

We describe SWAPI [54], an implementation of this system for JavaScript (which we here denote by λ_{JS}). The direct approach for implementing the embedded language $L_{\text{IFC}}(\text{RR}, \lambda_{\text{JS}})$ is by running multiple instances of the JS runtime, i.e., in separate OS-level threads.[8] In SWAPI, the IFC language functionality is implemented in the JS runtime much like browser layout engines implement the DOM, and expose the new functions, e.g., `getElementById`, by attaching them to the global JS object. In turn, each task created with **sandbox** is executed in a separate thread, running a separate, *fresh* instance of our modified JS runtime. Formally, **sandbox** is defined with $\kappa\ (\mathbf{\Sigma}) = \mathbf{\Sigma}_0$, where $\mathbf{\Sigma}_0$ is the global object corresponding to the standard JS library (e.g., $\mathbf{\Sigma}_0$ contains `Object`, `Array`, etc.).

Since this implementation approach relies on multiple runtimes of the language, sending arbitrary objects between tasks can result in unexpected behavior, e.g., when the object contains references. As discussed in Section 4, we need to restrict **send** to expressions that can be marshalled as strings, i.e., structurally clonable objects. In our formalization, this amounts to restricting the IFC language rule for **send** such that only strings can be shared:

$$\text{JS-SEND}\quad \frac{l \sqsubseteq l' \qquad \Sigma\,(i') = \Theta \qquad \Sigma' = \Sigma\,[i' \mapsto (l', i, v), \Theta] \qquad e = {}^{\text{IT}}\lceil \mathbf{e} \rceil \qquad \mathcal{E}_{\mathbf{\Sigma}}\,[\texttt{typeOf(}\mathbf{e}\texttt{)} \texttt{ === "string"}] \to \mathcal{E}_{\mathbf{\Sigma}}\,[\texttt{true}]}{\Sigma; \langle \mathbf{\Sigma}, E[\textbf{send}\ i'\ l'\ v]_I \rangle^i_l, \ldots \hookrightarrow \Sigma'; \alpha_\Sigma(\langle \mathbf{\Sigma}, E[\langle\rangle]_I \rangle^i_l, \ldots)}$$

We remark that this is similar to the existing `postMessage` API used for iframe and worker communications [27]. Thus, the SWAPI implementation provides an IFC version of `postMessage`, defined in terms of **send**, that additionally takes a label argument.

Unsurprisingly, our coarse-grained combined language approach has been inspired by existing browser (security) architectures. Browsers, for example, isolate pages of different origins by running them in separate runtimes. Similarly, they provide the worker JS object [27], which allows JavaScript to execute code in separate threads with separate JS runtimes (and fresh global objects). In both cases, code relies on the `postMessage` message-passing API for communication, similar to our system.[9] As suggested by our formal semantics, SWAPI can directly leverage these mechanisms to implement the semantics of $L_{\text{IFC}}(\text{RR}, \lambda_{\text{JS}})$ without modifying the JS runtime or intrusively changing the browser layout engine.

However, a simple implementation of just associating a label with workers and browsing context (iframes and top-level pages) to enforce IFC on the executing JavaScript is not sufficient, as APIs which introduce extra computational effects must be manually made aware of information flow control. For example, the initial global object of a worker contains the `XMLHttpRequest` (XHR) object, while the global object of a browsing context additionally contains the DOM. In both cases, JS code can trivially leak information (e.g., to the network with XHR or persistent storage using the DOM). To address this, SWAPI additionally uses content security policy (CSP) and (iframe) sandboxes [59, 60] to restrict external effects according to the label of a task (worker or browsing

[6] However, lazy evaluation can still produce a covert channel.

[7] It's worth noting that the proofs of non-interference we have given for asynchronous communication primitives are new and not in the original presentation of LIO.

[8] The Firefox implementation of SWAPI also considers the case where tasks share the JavaScript runtime (and are cooperatively scheduled on the event-loop), but the heap and execution context of each task is disjoint. This is similar to the single-heap language of Section 4. However, since the heap separation is provided by the browser the browser, we do not discuss this further. The interested reader is referred to [54] for more details.

[9] The message-passing approach is, in part, due to ECMAScript's lack of well-defined semantics for concurrency. Hence sharing and accessing objects such as the DOM across worker threads is undefined.

context). Accommodating these features lies outside the purview of our formalism.

6. Extensions and Limitations

In this section we consider several extensions to our IFC system. Specifically, we discuss how the specification language can be extended to consider other IFC-aware features (e.g., fine-grained labeled values), a static type system, or more complex scheduling policies. When considering these extensions we also discuss some limitations of our approach.

6.1 IFC Language Features

Different concrete IFC implementations may wish to extend our minimal system with labeled constructs appropriate for their envisioned applications. For example, SWAPI provides a labeled version of the XHR object, while LIO implements a labeled file system, used in server-side web applications [22]. Below we consider the addition of several labeled constructs to the abstract specification. We remark that that the non-interference proofs must account for the new constructs, since they amend the IFC calculus semantics (usually by addition).

Labeled values In traditional language-based dynamic IFC systems, a label is associated with values. Hence, a program that, for example, simply writes labeled messages to a labeled log can operate on both public and sensitive values. In its simplest form, our coarse grained system requires that the current label of a task be at least at the level of the sensitive data to reflect the fact that such data is in scope.

Thus, an extension that allows supports such fine-grained explicitly labeled values is essential. To this end, we extend the IFC calculus with explicitly labeled values, much like those of LIO and Breeze [28, 51]: $\mathbf{v} ::= \cdots \mid \textbf{Labeled}\ l\ \mathbf{v}$. Following LIO, we say that the value $\mathbf{v}$ is protected by label l, while the label l itself is protected by the task's current label. The label of such values can be inspected the task without requiring the current label to be raised. However, when a task wishes to inspect the protected value $\mathbf{v}$, it must first raise it label to at least l to reflect that it is incorporating data at such sensitivity level in its scope. When creating labeled values the label l must be above the current label; otherwise it cannot be said that protection has been transferred from the current label to l. (For these rules, we assume that the **Labeled** constructor is not part of the surface syntax and thus cannot be accessed directly to construct or destruct such values without label checks.) We omit the reduction rules for labeled values since they closely follow those presented in [51].

Labeled mutable references/variables/channels Extending the calculus with other labeled features, such as references, mutable variables (MVars) [31], or channels, follows in a similar way. In these cases, however, we must keep track of the additional state (as opposed to the labeled value case which requires no additional bookkeeping). While we can modify the IFC calculus to add such labeled state to each individual task, adding it to the system state Σ leads to more interesting use cases. Specifically, by amending Σ, as in [51, 52], we can allow threads to use these constructs to synchronize, or communicate with constructs other than **send/recv** in a safe manner. For example, when extending the calculus with labeled references, Σ additionally contains a store that maps addresses to $(l, \mathbf{v})$ pairs which can be read and written to by different tasks through a labeled **ref** implementations. Since such labeled constructs have also been considered before in other concurrent systems, we do not discuss them further.

Privileges Decentralized IFC extends IFC with the decentralized label model of Myers and Liskov [40] to allow for more general applications, including systems consisting of mutually distrustful parties. In a decentralized system, a computation is executed with a set of *privileges*, which, when exercised, allow the computation to declassify data (e.g., by lowering the current label). Practical IFC systems (e.g., [28, 41, 51, 65]) rely on privileges to implement many applications. Since both the LIO and SWAPI implementations already support such features, we believe that extending our calculus to consider privileges is straight forward. However, the challenge with such an extension lies in the precise security guarantees that must be proved.

Clearance LIO, SWAPI, and Breeze additionally provide a discretionary access control mechanism—called *clearance*—at the language level [28, 51]; this mechanisms is used to restrict a computation from accessing data (or communicating with entities) above a specified label, the clearance. For simplicity, we omitted clearance from our formalism. We only remark that the mechanism generalizes beyond these IFC systems in a straight forward manner and does not affect our definitions or proofs in any meaningful way.

6.2 Type Safety

One important consideration for any new language feature is whether or not it preserves type safety. If implementations are allowed to have undefined behavior when the system gets stuck, concerns of type safety can be directly applicable to security.

In our presentation, we have demanded that implementations *not* have undefined behavior when getting stuck: in such situations, the I-NOSTEP rule applies, where the stuck thread should be dropped from execution. Our combined language will be not get stuck, even if the target language could get stuck. This sort of guarantee can be achieved at a coarse granularity for languages like C by using hard isolation. However, for many other languages, this is too stringent a requirement.

A possible way to relax this requirement is to drop the I-NOSTEP rule, and instead demand type-safety of the combined language. This depends on the type system of the target language, so it is difficult to make any general statements about how one would go about doing this. However, there are few general remarks to be made:

- In Matthews and Findlers original paper [37], an emphasis was on using type boundaries to mediate between the type systems of the two languages, i.e. handling conversions of values from one language to the other. These techniques are directly applicable here.
- In the case of mini-ES, type safety proceeds by a preservation theorem that refers to well-typed stores. In our setting, there are now multiple stores, and expressions may move from being interpreted with one store to another (e.g. when an address is sent from one thread to another); a clear violation of type safety. In this case, operations such as **send** have their typing rules require the expressions being sent be typeable in an arbitrary well-typed store, statically disallowing the sending of addresses, or any types which may contain addresses. In practice, this may be further restricted to types which can be easily marshalled, e.g. strings.
- Similarly, the choice of κ (from Section 2) directly influences the difficulty of the type-safety proof for **sandbox** when the target language has a store. When κ is identity, most expressions can be allowed, since the store remains the same; if κ drops the store, however, only closed expressions can be sandboxed.

6.3 Scheduling Policies

Our specification language is parametrized by a scheduling policy α, which maps a task list to a potentially different task list. As previously discussed, the precise definition of α dictates whether

the language is TSNI, TINI, or neither. For simplicity we considered two popular policies RR and SEQ for which we showed TSNI and TINI, respectively. However, our definitions allow for other scheduling policies that only rely on the current task list. For example, this allows us to implement the scheduler of [32] that always schedules less sensitive threads first; this can be employed to address external timing covert channels where an attacker can measure the delay between output events.[10] More generally, we believe that deterministic schedulers that always make "low-progress," i.e., if a task whose label is not above the current task's label will will eventually run, and do not decide which low tasks to schedule according to data more sensitive than the tasks' label, i.e., the scheduler does not schedule one public thread over another according to sensitive data, to be safe.

Despite allowing a number of practical schedulers to be expressed, our scheduler parametrization is limited. For instance, since α is a map on task lists, schedulers that rely on the global state Σ or previous task configurations cannot be expressed. Though, we do not believe this to be a fundamental limitation, our current formalization cannot directly be applied to more-powerful, but safe, scheduling policies, as considered by [45]. More fundamentally, our definitions rely on deterministic relations, and thus extending the system to consider non-deterministic schedulers (or even non-deterministic target languages) is non-trivial. For instance, this may potentially require the security condition to consider *low-determinism*, which states that a program is secure only if l-equivalent results are deterministic [47, 64].

6.4 External Effects

Our embedding assumes that the target language not have any primitives that can induce external-world effects. As discussed in Section 5, imposing this restriction can be challenging. Yet, external effects are crucial when implementing more complex real-world applications. For example, code in an IFC browser must load resources or perform XHR to be useful.

Like labeled references, features with external effects must be modeled in the IFC language; we must reason about the precise security implications of features that otherwise inherently leak data. Previous approaches have modeled external effects by internalizing the effects as writes/reads to labeled channels/references [52]. An alternative approach is to model such effects as messages to/from certain labeled tasks. These "special" tasks are trusted with access to the unlabeled primitives that can be used to perform the external effects; since the interface to these tasks is already part of the IFC language, the proof only requires showing that this task does not leak information. This latter approach of wrapping unsafe primitives is, for example, used in SWAPI to allow for controlled network communication. By wrapping the default XHR object, for example, we can allow code to communicate with hosts according to the task's current label.

7. Proofs

In this section we prove all theorems we have stated. We first observe that the non-interference claims for the languages $L_{\text{IFC}}(\text{SEQ}, \lambda)$ and $L_{\text{IFC}}(\text{RR}, \lambda)$ in Theorems 1 and 2 follow directly from Theorems 3 and 4, respectively, where the set of predicates is the set of always valid predicates (i.e., no restriction).

Before we proceed with the proof of Theorem 3, we state a lemma we will use.

Lemma 1. *We consider, for any target language λ, the restricted IFC language $L^{\mathcal{P}}_{IFC}(\alpha, \lambda)$ (according to Definition 6). Then, for any configurations c_1, c_1', c_2, and label l where*

$$c_1 \approx_l c_2 \qquad \textit{and} \qquad c_1 \hookrightarrow c_1' \tag{3}$$

there exists a configuration c_2' such that

$$c_1' \approx_l c_2' \qquad \textit{and} \qquad c_2 \hookrightarrow^* c_2' \,. \tag{4}$$

Proof of Theorem 3. We proof the theorem by induction on the length of the derivation sequence in (1). The base case for derivations of length 0 is trivial, allowing us to simple chose $c_2' = c_2$. In the step case, we split the derivation sequence from (1) of length $n + 1$ as follows:

$$c_1 \hookrightarrow c_1'' \hookrightarrow^n c_1'$$

for some configuration c_1''. By Lemma 1 to the first step and the induction hypothesis to the rest of the derivation sequence, we directly get the desired property. □

Proof sketch of Lemma 1. For space reasons, we only sketch the structure of the proof here. We refer the interested reader to the Appendix of the extended version (which we provide at `http://www.scs.stanford.edu/~deian/ifc-inside_extended.pdf`) for a full proof. The proof distinguishes two cases:

- If the task t_1 executed in c_1 is erased, then there is no need to take a step in c_2, as the computation in c_1 does not have any visible effects after erasure.
- Otherwise, there must be a corresponding task t_2 in c_2, and it can take exactly the same reduction step as the task in c_1. The only difficulty is that there might be secret tasks executing before t_2, but it can be shown that these do not influence the computation after erasure.

□

The proof of Theorem 4 about TINI proceeds largely in the same fashion as for Theorem 3 (TSNI), except that some cases are simpler due to the fact termination is not an issue (cf. [51] for a TINI proof of a similar system).

8. The Monadic Interpretation

In the previous sections, we have been primarily concerned with the combination of operational semantics in order to automatically augment a language with information flow control. In this section, we present a more abstract view of the situation, phrased in the language of monads and monad transformers. The key idea is that the computational effects of the target language should be thought of as a *monad transformer* on top of the information flow control base monad. Readers who are not familiar with monads should feel free to skip this section.

Monads were originally proposed as a method for organizing computational effects [39]. One of the key questions of interest was how to modularly build more complex computational effects out of simpler ones. One particular approach, the *incremental approach* [9, 36], utilizes monad transformers, which add an extra computation feature to a pre-existing monad. In the past, this approach has been criticized for a number of failings, including the fact that the order in meaning of a monad can depend on the order in which the monad transformers are applied. The canonical example of this phenomenon is the composition of the state monad and the error monad: depending on the ordering of the two monad transformers, state is preserved or thrown away upon an error.

These problems can be viewed as defects in a setting where we are simply interested in combining effects. However, in the setting of information flow control, we are primarily interested in augmenting the computational effects of a target language with information flow control *without* compromising non-interference.

[10] When considering a label lattice that is not totally ordered, e.g., DCLabels [40, 50], this scheduling policy is considerably more complex. See [32].

When the base monad is kept abstract (in the style of a restricted IO monad [55]), there is an easy argument why any monad transformer must also preserve non-interference: the monad transformer could have been implemented simply as a *pure* interpreter, and the pure interpreter is just another program like any other that may run on top of a restricted IO monad.

Now, one shouldn't simply force an interpreter to be reimplemented on top of a language supporting information flow control. As suggested by Filinski et al [21], we should use pure implementations of computational effects to understand computational effects when built into information flow languages: they give a good hint what the operational rules should be. This intuition backed the development of Section 2. Furthermore, developing the relationship between concrete semantics and abstract semantics in Section 4 relates to the question of monad isomorphisms in the presence of data abstraction. We leave the investigation of this connection in more detail to future work.

9. Related Work

Our information flow control system is closely related to the coarse-grained information systems used in operating systems such as Asbestos [19], HiStar [65], and Flume [34], as well as language-based *floating-label IFC systems* such as LIO [51], and Breeze [28], where there is a monotonically increased label associated with threads of execution. Our treatment of termination-sensitive and termination-insensitive interference originates from Smith and Volpano [49, 57].

One information flow control technique designed to handle legacy code is secure multi-execution (SME) [16]. It runs multiple copies of the program, one per security level, where the semantics of I/O interactions is altered. Authors often present SME considering one specific programming language [7, 44]. However, Bielova al. [10] uses a transition system to describe SME, where the details of the underlying language is hidden. Zanarini et al. [63] propose a novel semantics for programs based on interaction trees [30], which treats programs as black-boxes about which nothing is known except what is inferred from their I/O interactions with the environment. Similar to SME, our approach mediates I/O operations; however, our approach only runs the program once.

One of the primary motivations behind this paper is constructing a practical information flow control system for JavaScript. Some of these systems attempt to perform fine-grained information flow of JavaScript [25, 26, 33]. While fine-grained information flow control can result in less false alarms and target legacy code, it comes at the cost of complexity (the system must accommodate the entirety of JavaScript's semantics). Moreover, the runtime performance cost are sometimes too high, e.g., higher than 100%! [26]. Alternatively, coarse-grained approaches have also been proposed for the web scenario [2, 14, 62].

The constructs in our IFC language, as well as the behavior of inter-thread communication, is reminiscent of distributed systems like Erlang [3]. There is a close correspondence: in distributed systems, isolation is required due to physical constraints; in information flow control, isolation is required to enforce noninterference. Papagiannis et al. [43] built an information flow control system on top of Erlang which is quite similar to ours. However, they do not use a floating label system (processes can find out when sending a message failed due to a forbidden information flow), and they include no security proofs.

There appears to be limited literature in general techniques for retrofitting information flow control to arbitrary languages; however, one time-honored technique is defining a fundamental calculus, for which other languages can be desugared into. Abadi et al. [1] motivate their core calculus of dependency by showing how various previous systems can be encoded in it; Tse and Zdancewic [56], in turn, show how this calculus can be encoded in System F via parametricity. Broberg and Sands [13] encodes several IFC systems into Paralocks. However, this line of work is primarily focused on static enforcements.

Our definition of isomorphism of information flow control languages in Section 4 is closely related to the idea of *bisimulation* [38] and process calculi, although as our system is deterministic, we do not use the theory in any deep way.

It has long been observed that information flow control and monads are closely related. This observation has been used by Abadi et al. [1] to structure access to private data by protecting it using a lattice of monads, one per security level. Russo et. al. [46] extend this idea to languages with side-effects. Our system uses monads in a conceptually different manner, where there is a single monad representing the information flow control computational effects; this is inline with the use of monads in other systems [17, 24, 51]. Devriese and Piessens [17] utilize monad transformers as an essential component in their work. However, their use of monad transformers is to explain when extra constraints on a non-information flow control aware base monad should be applied (i.e., during lifting). Harrison and Hook [24] define a notion of *atomic noninterference*, which is preserved through monad transformer application. However, it is not clear what is the relationship between this notion and traditional noninterference, and their paper only proves a weaker property of no write down.

10. Conclusion

We presented a formal semantics which describes how to extend arbitrary languages with language-based coarse-grained IFC. We prove the security guarantees TSNI and TINI, where the proof technique is parametrized on the target language, for two scheduling policies. In addition, we provided conditions on how to securely refine our formal semantics to consider optimizations required in practice. Finally, we described extensions to the core IFC language with more advanced concepts like labeled values, privileges, and a notion of clearance, and therefore connecting it with existing IFC implementations for JavaScript, Haskell, and (in less detail) proposing IFC for C. With this work, we aim to expose how ideas from IFC in OS can be systematically applied in programming languages.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A Core Calculus of Dependency. In *POPL*, pages 147–160, Jan. 1999.

[2] D. Akhawe, F. Li, W. He, P. Saxena, and D. Song. Data-confined HTML5 applications. In *ESORICS*. Springer, 2013.

[3] J. Armstrong. Making reliable distributed systems in the presence of software errors, 2003.

[4] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-Insensitive Noninterference Leaks More Than Just a Bit. ESORICS '08. Springer-Verlag, 2008.

[5] T. H. Austin and C. Flanagan. Efficient Purely-Dynamic Information Flow Analysis. In *PLAS*, June 2009.

[6] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of Multithreaded Programs by Compilation. In *ESORICS*, 2007.

[7] G. Barthe, J. M. Crespo, D. Devriese, F. Piessens, and E. Rivas. Secure multi-execution through static program transformation. In *FMOODS/FORTE 2012*, June 2012.

[8] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *OSDI*, pages 335–348, 2012.

[9] N. Benton, J. Hughes, and E. Moggi. Monads and effects. In *APPSEM'2000*, pages 42–122. Springer-Verlag, 2000.

[10] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive noninterference for a browser model. In *NSS'2011*, Sept. 2011.

[11] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting applications into reduced-privilege compartments. In *NSDI*, 2008.

[12] Boudol and Castellani. Noninterference for concurrent programs. In *Proc. ICALP'01*, volume 2076 of *LNCS*. Springer-Verlag, July 2001.

[13] N. Broberg and D. Sands. Paralocks: Role-based information flow control and beyond. In *POPL'10*. ACM, 2010.

[14] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *CCS'12*, New York, NY, USA, 2012. ACM.

[15] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976. ISSN 0001-0782.

[16] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *SP*. IEEE Computer Society, 2010.

[17] D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *TLDI*, New York, NY, USA, 2011.

[18] Ecma International. ECMAScript language specification. `http://www.ecma.org/`, 2014.

[19] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system, 2005.

[20] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *TCS*, 103(2):235–271, Sept. 1992.

[21] A. Filinski. Monads in action. *SIGPLAN Not.*, 45(1):483–494, Jan. 2010. ISSN 0362-1340.

[22] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *OSDI*, pages 47–60. USENIX, 2012.

[23] J. Goguen and J. Meseguer. Security policies and security Models, April 1982.

[24] W. L. Harrison. Achieving information flow security through precise control of effects. In *CSF*, pages 16–30. IEEE Computer Society, 2005.

[25] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *CSF*. IEEE Computer Society, 2012.

[26] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proc. 29th ACM Symposium on Applied Computing*, 2014.

[27] I. Hickson. Web workers, 2012.

[28] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett. All your IFCException are belong to us. In *SP'13*, pages 3–17, Washington, DC, USA, 2013.

[29] C. Hritcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos. Testing noninterference, quickly. In *ICFP*, pages 455–468. ACM, 2013.

[30] B. Jacobs and J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997.

[31] S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL*. ACM, 1996.

[32] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proc. of IEEE Symposium on Sec. and Privacy*. IEEE, 2011.

[33] C. Kerschbaumer, E. Hennigan, S. Brunthaler, P. Larsen, and M. Franz. Integrity considerations for secure computer systems. Technical Report 12-01, Department of Information and Computer Science, Univ. of California Irvine, October 2012.

[34] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *21st Symp. on Operating Systems Principles*, October 2007.

[35] P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19):1974–1994, 2010.

[36] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL*, 1995.

[37] J. Matthews and R. B. Findler. Operational semantics for multi-language programs. In *POPL*, pages 3–10, New York, NY, USA, 2007.

[38] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. ISBN 0131149849.

[39] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1): 55–92, July 1991. ISSN 0890-5401.

[40] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*, pages 129–142, 1997.

[41] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Comput. Syst.*, 9(4):410–442, October 2000.

[42] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java Information Flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[43] I. Papagiannis, M. Migliavacca, D. M. Eyers, B. Sh, J. Bacon, and P. Pietzuch. Enforcing user privacy in web applications using Erlang.

[44] W. Rafnsson and A. Sabelfeld. Secure multi-execution: fine-grained, declassification-aware, and transparent. 2013.

[45] A. Russo and A. Sabelfeld. Securing Interaction between Threads and the Scheduler. In *CSFW*, pages 177–189, July 2006.

[46] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Haskell*, 2008.

[47] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications*, 21(1):5–19, 2003.

[48] V. Simonet. The Flow Caml System. Software release at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, July 2003.

[49] G. Smith and D. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *POPL*, pages 355–364, Jan. 1998.

[50] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *NordSec*, volume 7161 of *LNCS*, pages 223–239. Springer, October 2011.

[51] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible Dynamic Information Flow Control in Haskell. In *Haskell*, 2011.

[52] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *ICFP*, September 2012.

[53] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arXiv:1207.1457*, 2012.

[54] D. Stefan, P. Marchenko, B. Karp, D. Mazières, and D. Herman. Protecting users by confining JavaScript with SWAPI. Technical report, Stanford, March 2014.

[55] D. Terei, S. Marlow, S. Peyton Jones, and D. Mazières. Safe haskell. In *Proceedings of the 2012 Haskell Symposium*, Haskell'12, pages 137–148, New York, NY, USA, 2012.

[56] S. Tse and S. Zdancewic. Translating dependency into parametricity. In *Proc. of the Ninth ACM SIGPLAN International Conference on Functional Programming*. ACM, 2004.

[57] D. Volpano and G. Smith. Eliminating Covert Flows with Minimum Typings. In *CSFW*. IEEE Computer Society, 1997.

[58] W3C. HTML5 web messaging. `http://www.w3.org/TR/webmessaging/`, May 2012.

[59] W3C. Content security policy 1.1. `https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html`, Nov. 2013.

[60] W3C. HTML5. `http://www.w3.org/TR/html5/`, Aug. 2013.

[61] E. Z. Yang and D. Mazières. Space limits for Haskell. In *PLDI*, page To appear, 2014. Available at http://ezyang.com/papers/ezyang14-rlimits.pdf.

[62] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with BFlow. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09. ACM, 2009.

[63] D. Zanarini, M. Jaskelioff, and A. Russo. Precise enforcement of confidentiality for reactive systems. In *CSF*, pages 18–32. IEEE, 2013.

[64] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *CSFW*, pages 29–43, June 2003.

[65] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2006.