From Fine- to Coarse-Grained Dynamic Information Flow Control and Back

Downloaded from: https://research.chalmers.se, 2022-12-13 22:41 UTC

Citation for the original published paper (version of record):
http://dx.doi.org/10.1145/3290389

N.B. When citing this work, cite the original published paper.
From Fine- to Coarse-Grained Dynamic Information Flow Control and Back

MARCO VASSENA, Chalmers University of Technology, Sweden
ALEJANDRO RUSSO, Chalmers University of Technology, Sweden
DEEPAK GARG, Max Planck Institute for Software Systems, Germany
VINEET RAJANI, Max Planck Institute for Software Systems, Germany
DEIAN STEFAN, University of California San Diego, USA

We show that fine-grained and coarse-grained dynamic information-flow control (IFC) systems are equally expressive. To this end, we mechanize two mostly standard languages, one with a fine-grained dynamic IFC system and the other with a coarse-grained dynamic IFC system, and prove a semantics-preserving translation from each language to the other. In addition, we derive the standard security property of non-interference of each language from that of the other, via our verified translation. This result addresses a longstanding open problem in IFC: whether coarse-grained dynamic IFC techniques are less expressive than fine-grained dynamic IFC techniques (they are not!). The translations also stand to have important implications on the usability of IFC approaches. The coarse- to fine-grained direction can be used to remove the label annotation burden that fine-grained systems impose on developers, while the fine- to coarse-grained translation shows that coarse-grained systems—which are easier to design and implement—can track information as precisely as fine-grained systems and provides an algorithm for automatically retrofitting legacy applications to run on existing coarse-grained systems.

CCS Concepts: • Security and privacy → Formal security models;
Additional Key Words and Phrases: Information-flow control, verified source-to-source transformations, Agda

ACM Reference Format:

1 INTRODUCTION

Dynamic information-flow control (IFC) is a principled approach to protecting the confidentiality and integrity of data in software systems. Conceptually, dynamic IFC systems are very simple—they associate security levels or labels with every bit of data in the system to subsequently track and restrict the flow of labeled data throughout the system, e.g., to enforce a security property such as non-interference [Goguen and Meseguer 1982]. In practice, dynamic IFC implementations are considerably more complex—the granularity of the tracking system alone has important implications for the usage of IFC technology. Indeed, until somewhat recently [Roy et al. 2009; Stefan et al. 2017], granularity was the main distinguishing factor between dynamic IFC operating systems and...
programming languages. Most IFC operating systems (e.g., [Efstathopoulos et al. 2005; Krohn et al. 2007; Zeldovich et al. 2006]) are coarse-grained, i.e., they track and enforce information flow at the granularity of a process or thread. Conversely, most programming languages with dynamic IFC (e.g., [Austin and Flanagan 2009; Hedin et al. 2014; Hritcu et al. 2013; Yang et al. 2012; Zdancewic 2002]) track the flow of information in a more fine-grained fashion, e.g., at the granularity of program variables and references.

Dynamic coarse-grained IFC systems in the style of LIO [Buiras et al. 2015; Heule et al. 2015; Stefan et al. 2012, 2017, 2011; Vassena et al. 2017] have several advantages over dynamic fine-grained IFC systems. Such coarse-grained systems are often easier to design and implement—they inherently track less information. For example, LIO protects against control-flow-based implicit flows by tracking information at a coarse-grained level—to branch on secrets, LIO programs must first taint the context where secrets are going to be observed. Finally, coarse-grained systems often require considerably fewer programmer annotations—unlike fine-grained ones. More specifically, developers often only need a single label-annotation to protect everything in the scope of a thread or process responsible to handle sensitive data.

Unfortunately, these advantages of coarse-grained systems give up on the many benefits of fine-grained ones. For instance, one main drawback of coarse-grained systems is that it requires developers to compartmentalize their application in order to avoid both false alarms and the label creep problem, i.e., wherein the program gets too “tainted” to do anything useful. To this end, fine-grained systems often create special abstractions (e.g., event processes [Efstathopoulos et al. 2005], gates [Zeldovich et al. 2006], and security regions [Roy et al. 2009]) that compensate for the conservative approximations of the coarse-grained tracking approach. Furthermore, fine-grained systems do not impose the burden of focusing on avoiding the label creep problem on developers. By tracking information at fine granularity, such systems are seemingly more flexible and do not suffer from false alarms and label creep issues [Austin and Flanagan 2009] as coarse-grained systems do. Indeed, fine-grained systems such as JSFlow [Hedin et al. 2014] can often be used to secure existing, legacy applications; they only require developers to properly annotate the application.

This paper removes the division between fine- and coarse-grained dynamic IFC systems and the belief that they are fundamentally different. In particular, we show that dynamic fine-grained and coarse-grained IFC are equally expressive. Our work is inspired by the recent work of Rajani et al. [2017]; Rajani and Garg [2018], who prove similar results for static fine-grained and coarse-grained IFC systems. Specifically, they establish a semantics- and type-preserving translation from a coarse-grained IFC type system to a fine-grained one and vice-versa. We complete the picture by showing a similar result for dynamic IFC systems that additionally allow introspection on labels at run-time. While label introspection is meaningless in a static IFC system, in a dynamic IFC system this feature is key to both writing practical applications and mitigating the label creep problem [Stefan et al. 2017].

Using Agda, we formalize a traditional fine-grained system (in the style of [Austin and Flanagan 2009]) extended with label introspection primitives, as well as a coarse-grained system (in the style of [Stefan et al. 2017]). We then define and formalize modular semantics-preserving translations between them. Our translations are macro-expressible in the sense of Felleisen [1991].

We show that a translation from fine- to coarse-grained is possible when the coarse-grained system is equipped with a primitive that limits the scope of tainting (e.g., when reading sensitive data). In practice, this is not an imposing requirement since most coarse-grained systems rely on such primitives for compartmentalization. For example, Stefan et al. [2012, 2017], provide toLabel blocks and threads for precisely this purpose. Dually, we show that the translation from coarse- to fine-grained is possible when the fine-grained system has a primitive taint(·) that relaxes precision to keep the program counter label synchronized when translating a program to the
Type: \( \tau ::= \text{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \mathcal{L} \mid \text{Ref } \tau \)

Labels: \( \ell, \text{pc} \in \mathcal{L} \)

Address: \( n \in \mathbb{N} \)

Environment: \( \theta \in \text{Var} \rightarrow \text{Value} \)

Raw Value: \( r ::= () \mid (x, e, \theta) \mid \text{inl}(v) \mid \text{inr}(v) \mid (v_1, v_2) \mid \ell \mid n_\ell \)

Value: \( v ::= r^\ell \)

Expression: \( e ::= x \mid \lambda x. e \mid e_1 \cdot e_2 \mid () \mid \ell \mid \text{inl}(e) \mid \text{inr}(e) \mid \text{case}(e, x, e_1, x, e_2) \mid (e_1, e_2) \mid \text{fst}(e) \mid \text{snd}(e) \mid \text{.getLabel} \mid \text{labelOf}(e) \mid \text{taint}(e_1, e_2) \mid \text{new}(e) \mid ! e \mid e_1 := e_2 \mid \text{labelOfRef}(e) \mid e_1 \sqsubseteq^? e_2 \)

Type System: \( \Gamma \vdash e : \tau \)

Configuration: \( c ::= (\Sigma, e) \)

Store: \( \Sigma \in (\ell : \text{Label}) \rightarrow \text{Memory } \ell \)

Memory \( \ell \): \( M ::= [] \mid r : M \)

Fig. 1. Syntax of \( \lambda^{dFG} \).

course-grained language. While this primitive is largely necessary for us to establish the coarse- to fine-grained translation, extending existing fine-grained systems with it is both secure and trivial.

The implications of our results are multi-fold. The fine- to coarse-grained translation formally confirms an old OS-community hypothesis that it is possible to restructure a system into smaller compartments to address the label creep problem—indeed our translation is a (naive) algorithm for doing so. This translation also allows running legacy fine-grained IFC compatible applications atop coarse-grained systems like LIO. Dually, the coarse- to fine-grained translation allows developers building new applications in a fine-grained system to avoid the annotation burden of the fine-grained system by writing some of the code in the coarse-grained system and compiling it automatically to the fine-grained system with our translation. The technical contributions of this paper are:

- A pair of semantics-preserving translations between traditional dynamic fine-grained and coarse-grained IFC systems equipped with label introspection (Theorems 3 and 5).
- Two different proofs of termination-insensitive non-interference (TINI) for each calculus: one is derived directly in the usual way (Theorems 1 and 2), while the other is recovered via our verified translation (Theorems 4 and 6).
- Mechanized Agda proofs of our results (~4,000 LOC).

The rest of this paper is organized as follows. Our dynamic fine- and coarse-grained IFC calculi are introduced in Sections 2 and 3, respectively. We also prove their soundness guarantees (i.e., termination-insensitive non-interference). Section 4 presents the translation from the fine- to the coarse-grained calculus and recovers the non-interference of the former from the non-interference theorem of the latter. Section 5 has similar results in the other direction. Related work is described in Section 6 and Section 7 concludes the paper.

2 FINE-GRAINED CALCULUS

In order to compare in a rigorous way fine- and coarse-grained dynamic IFC techniques, we formally define the operational semantics of two \( \lambda \)-calculi that respectively perform fine- and coarse-grained IFC dynamically. Figure 1 shows the syntax of the dynamic fine-grained IFC calculus \( \lambda^{dFG} \), which is inspired by Austin and Flanagan [2009] and extended with a standard (security unaware) type system \( \Gamma \vdash e : \tau \) (omitted), sum and product data types and security labels \( \ell \in \mathcal{L} \) that form a lattice.

\[^1\text{Artifact available at https://hub.docker.com/r/marcovassena/granularity/}\]
(\mathcal{L}, \sqsubseteq). In order to capture flows of information precisely at run-time, the \(\lambda^{dFG}\)-calculus features \textit{intrinsically labeled} values, written \(r^\ell\), meaning that raw value \(r\) has security level \(\ell\). Compound values, e.g., pairs and sums, carry labels to tag the security level of each component, for example a pair containing a secret and a public boolean would be written (true\(^H\), false\(^L\)).

Functional values are closures (\(x.e.\theta\)), where \(x\) is the variable that binds the argument in the body of the function \(e\) and all other free variables are mapped to some labeled value in the environment \(\theta\). The \(\lambda^{dFG}\)-calculus features a labeled partitioned stored, i.e., \(\Sigma \in (\ell : \mathcal{L}) \rightarrow \text{Memory } \ell\), where \(\text{Memory } \ell\) is the memory that contains values at security level \(\ell\). Each reference carries an additional label annotation that records the label of the memory it refers to—reference \(n_\ell\) points to the \(n\)-th cell of the \(\ell\)-labeled memory, i.e., \(\Sigma(\ell)\). Notice that this label has nothing to do with the \textit{intrinsic} label that decorates the reference itself. For example, a reference \((n_H)^L\) represents a secret reference in a public context, whereas \((n_L)^H\) represents a public reference in a secret context. Notice that there is no order invariant between those labels—in the latter case, the IFC runtime monitor prevents writing data to the reference to avoid \textit{implicit flows}. A program can create, read and write a labeled reference via constructs \texttt{new}(e), !e and \(e_1 := e_2\) and inspect its subscripted label with the primitive \texttt{labelOfRef}(\cdot).

### 2.1 Dynamics

The operational semantics of \(\lambda^{dFG}\) includes a security monitor that propagates the label annotations of input values during program execution and assigns security labels to the result accordingly. The monitor prevents information leakage by stopping the execution of potentially leaky programs, which is reflected in the semantics by not providing reduction rules for the cases that may cause insecure information flow. The relation \(\langle \Sigma, e \rangle \Downarrow^\ell_\theta \langle \Sigma', \nu \rangle\) denotes the evaluation of program \(e\) with initial store \(\Sigma\) that terminates with labeled value \(\nu\) and final store \(\Sigma'\). The environment \(\theta\) stores the input values of the program and is extended with intermediate results during function application and case analysis. The subscript \(pc\) is the \textit{program counter} label [Sabelfeld and Myers 2006]—it is a label that represents the security level of the context in which the expression is evaluated. The semantics employs the program counter label to (i) propagate and assign labels to values computed by a program and (ii) prevent implicit flow leaks that exploit the control flow and the store (explained below).

In particular, when a program produces a value, the monitor tags the raw value with the program counter label in order to record the security level of the context in which it was computed. For this reason all the introduction rules for ground and compound types ([Unit,Label,Func,Inl,Inr,Pair]) assign security level \(pc\) to the result. Other than that, these rules are fairly standard—we simply note that rule [Func] creates a closure by capturing the current environment \(\theta\).

When the control flow of a program \textit{depends} on some intermediate value, the program counter label is joined with the value’s label so that the label of the final result will be tainted with the result of the intermediate value. For instance, consider case analysis, i.e., \texttt{case } e \ x . e_1 \ x . e_2\). Rules [Case1] and [Case2] evaluate the scrutinee \(e\) to a value (either \texttt{inl}(\nu)^\ell\) or \texttt{inr}(\nu)^\ell\), add the value to the environment, i.e., \(\theta[ x \mapsto \nu]\), and then execute the appropriate branch with a program counter label tainted with \(\nu\)’s security label, i.e., \(pc \sqcup \ell\). As a result, the monitor tracks data dependencies across control flow constructs through the label of the result. Function application follows the same

---

2. The lattice is arbitrary and fixed. In examples we will often use the two point lattice \(\{L, H\}\), which only disallows secret to public flow of information, i.e., \(H \not\sqsubseteq L\).

We define the boolean type \texttt{bool} = \texttt{unit + unit}, boolean values as raw values, i.e., \texttt{true} = \texttt{inl}(\texttt{true})\(^L\), \texttt{false} = \texttt{inr}(\texttt{false})\(^L\), and if \(e\) then \(e_1\) else \(e_2\) = case \(e \mapsto e_1 \ldots e_2\).

4. In this work, we ignore leaks that exploit program termination and prove \textit{termination insensitive} non-interference for \(\lambda^{dFG}\) (Theorem 1).
H. L

when the program counter is secret, for example \( v \) counter label for security reasons. In rules elimination rules for variables and pairs taint the label of the corresponding value with the program of the result depends exclusively on the program counter label, e.g., is beneficial to not over-tainting the result: if the function never uses its argument then the label \( \ell \) security level

\[
\begin{align*}
(VAR) & & \langle \Sigma, x \rangle \Downarrow^{\theta} \langle \Sigma, \theta(x) \sqcup pc \rangle \\
(UNIT) & & \langle \Sigma, () \rangle \Downarrow^{\theta} \langle \Sigma, ()pc \rangle \\
(LABEL) & & \langle \Sigma, \ell \rangle \Downarrow^{\theta} \langle \Sigma, \ell pc \rangle \\
(FUN) & & \langle \Sigma, \lambda x. e \rangle \Downarrow^{\theta} \langle \Sigma, (x.e, \theta)pc \rangle \\
(APP) & & \langle \Sigma, e_1 \rangle \Downarrow^{\theta} \langle \Sigma', (x.e, \theta')f \rangle \\
 & & \langle \Sigma', e_2 \rangle \Downarrow^{\theta} \langle \Sigma'', v_2 \rangle \\
 & & \langle \Sigma'', e \rangle \Downarrow^{\theta[x\rightarrow v_1]} \langle \Sigma''', v \rangle \\
 & & \langle \Sigma, e_1 \rangle \Downarrow^{\theta} \langle \Sigma', v \rangle \\
 & & \langle \Sigma, e \rangle \Downarrow^{\theta} \langle \Sigma', v \rangle \\
(INL) & & \langle \Sigma, \text{inl}(e) \rangle \Downarrow^{\theta} \langle \Sigma', \text{inl}(v)pc \rangle \\
(INR) & & \langle \Sigma, \text{inr}(e) \rangle \Downarrow^{\theta} \langle \Sigma', \text{inr}(v)pc \rangle \\
(CASE1) & & \langle \Sigma, e \rangle \Downarrow^{\theta} \langle \Sigma', \text{inl}(v_1)f \rangle \\
 & & \langle \Sigma', e_1 \rangle \Downarrow^{\theta[x\rightarrow v_1]} \langle \Sigma''', v \rangle \\
 & & \langle \Sigma, \text{case}(e, x.e_1, x.e_2) \rangle \Downarrow^{\theta} \langle \Sigma''', v \rangle \\
(CASE2) & & \langle \Sigma, e \rangle \Downarrow^{\theta} \langle \Sigma', \text{inr}(v_2)f \rangle \\
 & & \langle \Sigma', e_2 \rangle \Downarrow^{\theta[x\rightarrow v_2]} \langle \Sigma'', v \rangle \\
 & & \langle \Sigma, \text{case}(e, x.e_1, x.e_2) \rangle \Downarrow^{\theta} \langle \Sigma'', v \rangle \\
(PAIR) & & \langle \Sigma, e_1 \rangle \Downarrow^{\theta} \langle \Sigma', v_1 \rangle \\
 & & \langle \Sigma', e_2 \rangle \Downarrow^{\theta} \langle \Sigma'', v_2 \rangle \\
 & & \langle \Sigma, (e_1, e_2) \rangle \Downarrow^{\theta} \langle \Sigma''', (v_1, v_2)pc \rangle \\
(FST) & & \langle \Sigma, e \rangle \Downarrow^{\theta} \langle \Sigma', (v_1, v_2)f \rangle \\
 & & \langle \Sigma, \text{fst}(e) \rangle \Downarrow^{\theta} \langle \Sigma', v_1 \sqcup \ell \rangle \\
(SND) & & \langle \Sigma, e \rangle \Downarrow^{\theta} \langle \Sigma', (v_1, v_2)f \rangle \\
 & & \langle \Sigma, \text{snd}(e) \rangle \Downarrow^{\theta} \langle \Sigma', v_2 \sqcup \ell \rangle \\
(TAINT) & & \langle \Sigma, e_1 \rangle \Downarrow^{\theta} \langle \Sigma', \ell'' \rangle \\
 & & \ell' \sqsubseteq \ell \\
 & & \langle \Sigma', e_2 \rangle \Downarrow^{\theta} \langle \Sigma'', v \rangle \\
 & & \langle \Sigma, \text{taint}(e_1, e_2) \rangle \Downarrow^{\theta} \langle \Sigma'', v \rangle \\
\end{align*}
\]

Fig. 2. Big-step semantics for \( \lambda^{dFG} \) (part I).

principle. In rule [APP], since the first premise evaluates the function to some closure \((x.e, \theta')\) at security level \( \ell \), the third premise evaluates the body with program counter label raised to \( pc \sqcup \ell \). The evaluation strategy is call-by-value: it evaluates the argument before the body in the second premise and binds the corresponding variable to its value in the environment of the closure, i.e., \( \theta' [ x \mapsto v_2 ] \). Notice that the security level of the argument is irrelevant at this stage and that this is beneficial to not over-tainting the result: if the function never uses its argument then the label of the result depends exclusively on the program counter label, e.g., \((\lambda x.()) y \Downarrow^{\lambda^{\ell}+42vH} ()H\). The elimination rules for variables and pairs taint the label of the corresponding value with the program counter label for security reasons. In rules [VAR,FST,SND] the notation, \( \ell \sqcup \ell' \) upgrades the label of \( v \) with \( \ell' \)—it is a shorthand for \( r.e \sqcup r' \) with \( v = r'. \) Intuitively, public values must be considered secret when the program counter is secret, for example \( x \Downarrow^{H^{\ell-4rH}} ()H\).
\[ (\text{LABELOf}) \]
\[ \langle \Sigma, e \rangle \uparrow^\theta_{pc} \langle \Sigma', r^{\ell'} \rangle \]
\[ \langle \Sigma, \text{LABELOf}(e) \rangle \uparrow^\theta_{pc} \langle \Sigma', r^{\ell'} \rangle \]

\[ (\text{GetLabel}) \]
\[ \langle \Sigma, \text{GetLabel} \rangle \uparrow^\theta_{pc} \langle \Sigma'', pe^{pc} \rangle \]

\[ (\equiv^\gamma_-\text{T}) \]
\[ \langle \Sigma, e_1 \rangle \uparrow^\theta_{pc} \langle \Sigma', \ell_1^{\ell_1} \rangle \]
\[ \langle \Sigma', e_2 \rangle \uparrow^\theta_{pc} \langle \Sigma'', \ell_2^{\ell_2} \rangle \]
\[ \ell_1 \subseteq \ell_2 \]
\[ \langle \Sigma, e_1 \equiv^\gamma_2 e_2 \rangle \uparrow^\theta_{pc} \langle \Sigma'', \text{inl}(()^{pc})^{\ell_1^\gamma_2 \cup \ell_2^{\ell_2}} \rangle \]

\[ (\equiv^\gamma_-\text{F}) \]
\[ \langle \Sigma, e_1 \rangle \uparrow^\theta_{pc} \langle \Sigma', \ell_1^{\ell_1} \rangle \]
\[ \langle \Sigma', e_2 \rangle \uparrow^\theta_{pc} \langle \Sigma'', \ell_2^{\ell_2} \rangle \]
\[ \ell_1 \not\subseteq \ell_2 \]
\[ \langle \Sigma, e_1 \equiv^\gamma_2 e_2 \rangle \uparrow^\theta_{pc} \langle \Sigma'', \text{inr}(()^{pc})^{\ell_1^\gamma_2 \cup \ell_2^{\ell_2}} \rangle \]

Fig. 3. Big-step semantics for \( \lambda^{dFG} \) (part II).

Label Introspection. The \( \lambda^{dFG} \)-calculus features primitives for label introspection, namely getLabel, labelOf(·) and \( \equiv^\gamma_- \)—see Figure 3. These operations allow to respectively retrieve the current program counter label, obtain the label annotations of values, and compare two labels (inspecting labels at run-time is useful for controlling and mitigating the label creep problem).

Enabling label introspection raises the question of what label should be assigned to the label itself (in \( \lambda^{dFG} \) every value, including all label values, must be annotated with a label). As a matter of fact, labels can be used to encode secret information and thus careless label introspection may open the doors to information leakage [Stefan et al. 2017]. Notice that in \( \lambda^{dFG} \), the label annotation on the result is computed by the semantics together with the result and thus it is as sensitive as the result itself (the label annotation on a value depends on the sensitivity of all values affecting the control-flow of the program up to the point where the result is computed). This motivates the design choice to protect each projected label with the label itself, i.e., \( \ell^{\ell} \) and \( pe^{pc} \) in rules [GetLabel] and [LABELOf] in Figure 2. We remark that this choice is consistent with previous work on coarse-grained IFC languages [Buiras et al. 2014; Stefan et al. 2017], but novel in the context of fine-grained IFC.

Finally, primitive \text{taint}(e_1, e_2) \text{ temporarily raises the program counter label to the label given by the first argument in order to evaluate the second argument. The fine-to-coarse translation in Section 4 uses \text{taint}(·) to loosen the precision of \( \lambda^{dFG} \) in a controlled way and match the coarse approximation of our coarse-grained IFC calculus (\( \lambda^{dCG} \)) by upgrading the labels of intermediate values systematically. In rule \text{Taint}, the constraint \( \ell'' \subseteq \ell \) ensures that the label of the nested context \( \ell \) is at least as sensitive as the program counter label \( pc \). In particular, this constraint ensures that the operational semantics have Property 1 ("the label of the result is at least as sensitive as the program counter label") even with rule \text{Taint}.

Property 1. If \( \langle \Sigma, e \rangle \uparrow^\theta_{pc} \langle \Sigma', r^{\ell} \rangle \) then \( pc \subseteq \ell \).

Proof. By induction on the given evaluation derivation.

References. We now extend the semantics presented earlier with primitives that inspect, access and modify the labeled store via labeled references. See Figure 4. Rule [New] creates a reference \( n^{\ell} \), labeled with the security level of the initial content, i.e., label \( \ell \), in the \( \ell \)-labeled memory \( \Sigma(\ell) \).
Since the security level of the reference is as sensitive as the content, which is at least as sensitive as the program counter label by Property 1 ($pc \subseteq \ell$) this operation does not leak information via implicit flows. When reading the content of reference $n_\ell$ at security level $\ell'$, rule [READ] retrieves the corresponding raw value from the $n$-th cell of the $\ell$-labeled memory, i.e., $\Sigma'(\ell)[n] = r$ and upgrades its label to $\ell \sqcup \ell'$ since the decision to read from that particular reference depends on information at security level $\ell'$. When writing to a reference the monitor performs security checks to avoid leaks via explicit or implicit flows. Rule [WRITE] achieves this by evaluating the reference, i.e., $(n_\ell)^{\ell_1}$ and replacing its content with the value of the second argument, i.e., $r^{\ell_2}$, under the conditions that the decision of “which” reference to update does not depend on data more sensitive than the reference itself, i.e., $\ell_1 \subseteq \ell$ (not checking this would leak via an implicit flow), and that the new content is no more sensitive than the reference itself, i.e., $\ell_2 \subseteq \ell$ (not checking this would leak sensitive information to a less sensitive reference via an explicit flow). Lastly, rule [LABELOFRef] retrieves the label of the reference and protects it with the label itself (as explained before) and taints it with the security level of the reference, i.e., $\ell^{\ell \sqcup \ell'}$ to avoid leaks. Intuitively, the label of the reference, i.e., $\ell$, depends also on data at security level $\ell'$ as seen in the premise.

Other Extensions. We consider $\lambda^{dFG}$ equipped with references as sufficient foundation to study the relationship between fine-grained and coarse-grained IFC. We remark that extending it with other side-effects such as file operations, or other IO-operations would not change our claims in Section 4 and 5. The main reason for this is that, typically, handling such effects would be done at the same granularity in both IFC enforcements. For instance, when adding file operations, both fine- (e.g., [Broberg et al. 2013]) and coarse-grained (e.g., [Efstathopoulos et al. 2005; Krohn et al. 2007; Russo et al. 2009; Stefan et al. 2011]) enforcements are likely to assign a single flow-insensitive label to each file in order to denote the sensitivity of its content. Then, those features could be handled flow-insensitively in both systems (e.g., [Myers et al. 2006; Pottier and Simonet 2003; Stefan et al. 2011; Vassena and Russo 2016]), in a manner similar to what we have just shown for references in $\lambda^{dFG}$.

5 $|M|$ denotes the length of memory $M$—memory indices start at 0.
6 Notice that $pc \subseteq \ell_1$ by Property 1, thus $pc \subseteq \ell_1 \subseteq \ell$ by transitivity. An implicit flow would occur if a reference is updated in a high branch, i.e., depending on the secret, e.g., let $x = \text{new}(0)$ in if $\text{secret}$ then $x := 1$ else $()$. 

Fig. 4. Big-step semantics for $\lambda^{dFG}$ (references).
We now prove that \( \lambda^{dFG} \) is secure, i.e., it satisfies termination insensitive non-interference (TINI) [Goguen and Meseguer 1982; Volpano and Smith 1997]. Intuitively, the security condition says that no terminating \( \lambda^{dFG} \) program leaks information, i.e., changing secret inputs does not produce any publicly visible effect. The proof technique is standard and based on the notion of \( L \)-equivalence, written \( v_1 \approx_L v_2 \), which relates values (and similarly raw values, environments, stores and configurations) that are indistinguishable for an attacker at security level \( L \). For clarity we use the 2-points lattice, assume that secret data is labeled with \( H \) and that the attacker can only observe data at security level \( L \). Our mechanized proofs are parametric in the lattice and in the security level of the attacker. \( L \)-equivalence for values and raw-values is defined formally by mutual induction in Figure 5. Rule [VALUE\( _L \)] relates observable values, i.e., raw values labeled below the security level of the attacker. These values have the same observable label (\( L \subseteq L \)) and related raw values, i.e., \( r_1 \approx_L r_2 \).

Rule [VALUE\( _H \)] relates non-observable values, which may have different labels not below the attacker level, i.e., \( \ell_1 \not\subseteq L \) and \( \ell_2 \not\subseteq L \). In this case, the raw values can be arbitrary. Raw values are \( L \)-equivalent when they consist of the same ground value ([UNIT, LABEL]), or are homomorphically related for compound values. For example, for the sum type the relation requires that both values are either a left or a right injection ([INL, INR]). In particular, closures are related if they contain the same function (up to \( \alpha \)-renaming)\(^7\) and \( L \)-equivalent environments, i.e., the environments are \( L \)-equivalent pointwise. Formally, \( \theta_1 \approx_L \theta_2 \) iff \( \text{Dom}(\theta_1) \equiv \text{Dom}(\theta_2) \) and \( \forall x. \theta_1(x) \approx_L \theta_2(x) \).

We define \( L \)-equivalence for stores pointwise, i.e., \( \Sigma_1 \approx_L \Sigma_2 \) iff for all labels \( \ell \in \mathcal{L} \), \( \Sigma_1(\ell) \approx_L \Sigma_2(\ell) \). Memory \( L \)-equivalence relates arbitrary \( L \)-labeled memories if \( \ell \not\subseteq L \), and pointwise otherwise, i.e., \( M_1 \approx_L M_2 \) iff \( M_1 \) and \( M_2 \) are memories labeled with \( \ell \subseteq L \), \( |M_1| = |M_2| \) and for all \( n \in \{0 \ldots |M_1|\} \), \( M_1[n] \approx_L M_2[n] \). Similarly, \( L \)-equivalence relates any two secret references (rule [REF\( _L \)]) but requires the same label and address for public references (rule [REF\( _H \)]). We naturally lift \( L \)-equivalence to initial configurations, i.e., \( c_1 \approx_L c_2 \) iff \( c_1 = (\Sigma_1, e_1) \), \( c_2 = (\Sigma_2, e_2) \), \( \Sigma_1 \approx_L \Sigma_2 \) and \( e_1 \equiv_{\alpha} e_2 \), and final configurations, i.e., \( c'_1 \approx_L c'_2 \) iff \( c'_1 = (\Sigma'_1, v_1) \), \( c'_2 = (\Sigma'_2, v_2) \) and \( \Sigma'_1 \approx_L \Sigma'_2 \) and \( v_1 \approx_L v_2 \).

We now formally state and prove that \( \lambda^{dFG} \) semantics preserves \( L \)-equivalence of configurations under \( L \)-equivalent environments, i.e., termination-insensitive non-interference (TINI).

**Theorem 1 (\( \lambda^{dFG} \)-TINI).** If \( c_1 \mathrel{\mathop{
mid}_{\mathcal{L}}^{\alpha}} c'_1 \), \( c_2 \mathrel{\mathop{
mid}_{\mathcal{L}}^{\alpha}} c'_2 \), \( \theta_1 \approx_L \theta_2 \) and \( c_1 \approx_L c_2 \), then \( c'_1 \approx_L c'_2 \).

\(^7\)Symbol \( \equiv_{\alpha} \) denotes \( \alpha \)-equivalence. In our mechanized proofs we use De Bruijn indexes and syntactic equivalence.
The type constructor shows the syntax of.

Stefan et al. whose state consists of a labeled store and the program counter label. In addition to standard explicitly labeled values with labels that represent their sensitivity and a label-aware instrumented security monitor 3 COARSE-GRAINED CALCULUS 

true are not intrinsically labeled, thus we encode boolean constants simply as true = inl() and false = inr().

As in \( \lambda^{dFG} \), we define bool = unit + unit and if \( e \) then \( e_1 \) else \( e_2 \) = case \( e \rightarrow e_1 \leftarrow e_2 \). Unlike \( \lambda^{dFG} \) values, \( \lambda^{dCG} \) values are not intrinsically labeled, thus we encode boolean constants simply as true = inl() and false = inr().

Fig. 6. Syntax of \( \lambda^{dCG} \).

Proof. By induction on the derivations. Dynamic language-based fine-grained IFC, of which \( \lambda^{dFG} \) is just a particular instance, represents an intuitive approach to tracking information flows in programs. Programmers annotate input values with labels that represent their sensitivity and a label-aware instrumented security monitor propagates those labels during execution and computes the result of the program together with a conservative approximation of its sensitivity. The next section describes an IFC monitor that tracks information flows at coarse granularity.

3 COARSE-GRAINED CALCULUS

One of the drawbacks of dynamic fine-grained IFC is that the programming model requires all input values to be explicitly and fully annotated with their security labels. Imagine a program with many inputs and highly structured data: it quickly becomes cumbersome, if not impossible, for the programmer to specify all the labels. The label of some inputs may be sensitive (e.g., passwords, pin codes, etc.), but the sensitivity of the rest may probably be irrelevant for the computation, yet a programmer must use appropriate labels for them as well. The programmer is then torn between two opposing risks: over-approximating the actual sensitivity can negatively affect execution (the monitor might stop secure programs), under-approximating the sensitivity can endanger security. Even worse, specifying many labels manually is error-prone and assigning the wrong security label to a piece of sensitive data can be catastrophic for security and completely defeat the purpose of IFC. Dynamic coarse-grained IFC represents an attractive alternative that requires fewer annotations, in particular it allows the programmer to label only the inputs that need to be protected.

Figure 6 shows the syntax of \( \lambda^{dCG} \), a standard simply-typed \( \lambda \)-calculus extended with security primitives for dynamic coarse-grained IFC, inspired by Stefan et al. [2011] and adapted to use call-by-value instead of call-by-name to match \( \lambda^{dFG} \). The \( \lambda^{dCG} \)-calculus features both standard (unlabeled) values and explicitly labeled values. For example, Labeled H true represents a secret boolean value of type Labeled bool. The type constructor LIO encapsulates a security state monad, whose state consists of a labeled store and the program counter label. In addition to standard return(·) and bind(·) constructs, the monad provides primitives that regulate the creation and the
inspection of labeled values, i.e., toLabeled(·), unlabel(·) and labelOf(·), and the interaction with the labeled store, allowing the creation, reading and writing of labeled references \( n_e \) through the constructs new(e), \( !e \), \( e_1 := e_2 \), respectively. The primitives of the LIO monad are listed in a separate sub-category of expressions called thunk. Intuitively, a thunk is just a description of a stateful computation, which only the top-level security monitor can execute—a thunk closure, i.e., \( (t, \theta) \), provides a way to suspend computations.

### 3.1 Dynamics

In order to track information flows dynamically at coarse granularity, \( \lambda^{dCG} \) employs a technique called floating-label, which was originally developed for IFC operating systems (e.g., [Zeldovich et al. 2006, 2008]) and that was later applied in a language-based setting. In this technique, throughout a program’s execution, the program counter floats above the label of any value observed during program execution and thus represents (an upper-bound on) the sensitivity of all the values that are not explicitly labeled. For this reason, \( \lambda^{dCG} \) stores the program counter label in the program configuration, so that the primitives of the LIO monad can control it explicitly (in technical terms the program counter is flow-sensitive, i.e., it may assume different values in the final configuration depending on the control flow of the program).[9]

Like \( \lambda^{dFG} \), the operational semantics of \( \lambda^{dCG} \) consists of a security monitor that fully evaluates secure programs but prevents the execution of insecure programs and similarly enforces termination-insensitive non-interference (Theorem 2). Figure 7 shows the big-step operational semantics of \( \lambda^{dCG} \) in two parts: (i) a top-level security monitor for monadic programs and (ii) a straightforward call-by-value side-effect-free semantics for pure expressions. The semantics of the security monitor is further split into two mutually recursive reduction relations, one for arbitrary expressions (Fig. 7a) and one specific to thunks (Fig. 7c). These constitute the forcing semantics of the monad, which reduce a thunk to a pure value and perform side-effects. In particular, given the initial store \( \Sigma \), program counter label \( pc \), expression \( e \) of type LIO \( \tau \) for some type \( \tau \) and input values \( \theta \) (which may or may not be labeled), the monitor executes the program, i.e., \( \langle \Sigma, pc, e \rangle \downarrow^\theta \langle \Sigma', pc', v \rangle \) and gives an updated store \( \Sigma' \), updated program counter \( pc' \) and a final value \( v \) of type \( \tau \), which also might not be labeled. The execution starts with rule [Force], which reduces the pure expression to a thunk closure, i.e., \( (t, \theta') \) and then forces the thunk \( t \) in its environment \( \theta' \) with the thunk semantics. The pure semantics is fairly standard—we report some selected rules in Fig. 7b for comparison with \( \lambda^{dFG} \). A pure reduction, written \( e \downarrow^\theta v \), evaluates an expression \( e \) with an appropriate environment \( \theta \) to a pure value \( v \). Notice that, unlike \( \lambda^{dFG} \), all reduction rules of the pure semantics ignore security, even those that affect the control flow of the program, e.g., rule [App]: they do not feature the program counter label or label annotations. They are also pure—they do not have access to the store, thus only the security monitor needs to protect against implicit flows.

If the pure evaluation reaches a side-effectful computation, i.e., thunk \( t \), it suspends the computation by creating a thunk closure that captures the current environment \( \theta \) (see rule [Thunk]). Notice that thunk closures and function closures are distinct values created by different rules, [Thunk] and [Fun] respectively.[10] Function application succeeds only when the function evaluates to a function closure (rule [App]). In the thunk semantics, rule [Return] evaluates a pure value embedded in the monad via return(·) and leaves the state unchanged, while rule [Bind] executes the first computation with the forcing semantics, binds the result in the environment i.e., \( \theta[x \mapsto v_1] \), passes it on to the second computation together with the updated state and returns the final result and

---

[9] In contrast, we consider \( \lambda^{dFG} \)’s program counter flow-insensitive because it is part of the evaluation judgment and its value changes only inside nested judgments.

[10] It would have also been possible to define thunk values in terms of function closures using explicit suspension and an opaque wrapper, e.g., LIO (·, t, θ).
From Fine- to Coarse-Grained Dynamic Information Flow Control and Back

(a) Forcing semantics: $\langle \Sigma, pc, e \rangle \Downarrow^\theta \langle \Sigma', pc', v \rangle$.

(b) Pure semantics: $e \Downarrow^\theta v$ (selected rules).

(c) Thunk semantics: $\langle \Sigma, pc, t \rangle \Downarrow^\theta \langle \Sigma', pc', v \rangle$.

state. Rule [Unlabel] is interesting. Following the floating-label principle, it returns the value wrapped inside the labeled value, i.e., $v$, and raises the program counter with its label, i.e., $pc \sqsubseteq \ell$, to reflect the fact that new data at security level $\ell$ is now in scope.

Floating-label based coarse-grained IFC systems like LIO suffer from the label creep problem, which occurs when the program counter gets over-tainted, e.g., because too many secrets have
Furthermore, notice that Stefan et al. 2017

\[ \text{ℓ} \text{ is the user-assigned label} \]

with the user-assigned label \( \ell \). proc receives as an argument, and checks that

\[ \text{LabelOfRef} \]

reference and rule \( \text{new}(e) \)\( \ell \)\( n \Rightarrow v \), \( \text{pc}, n_\ell \) \hfill (Read) \hfill \begin{align*}
& e \not\Downarrow n_\ell \quad \Sigma(\ell)[n] = v \\
& \langle \Sigma, \text{pc}, \text{eq} \rangle \not\Downarrow \langle \Sigma, \text{pc} \sqcup \ell, v \rangle
\end{align*}

(Write) \hfill \begin{align*}
& e_1 \not\Downarrow n_{\ell_1} \\
& e_2 \not\Downarrow n_{\ell_2} \\
& \text{Labeled } \ell_2 v \\
& \ell_2 \subseteq \ell_1 \\
& \text{pc} \subseteq \ell_1
\end{align*} \hfill \langle \Sigma, \text{pc}, e_1 := e_2 \rangle \not\Downarrow \langle \Sigma[\ell_1 \mapsto \Sigma(\ell_1)[n \mapsto v]], \text{pc}, () \rangle

(LabelOfRef) \hfill \begin{align*}
& e \not\Downarrow n_\ell \\
& \langle \Sigma, \text{pc}, \text{LabelOfRef}(e) \rangle \not\Downarrow \langle \Sigma, \text{pc} \sqcup \ell, \ell \rangle
\end{align*}

Fig. 8. Big-step semantics for \( \lambda_{dCG} \) (references).

unlabeled, to the point that no useful further computation can be performed. Primitive \text{toLabeled}(\cdot) provides a mechanism to address this problem by (i) creating a separate context where some sensitive computation can take place and (ii) restoring the original program counter label afterwards. Rule [ToLabeled] formalizes this idea. Notice that the result of the nested sensitive computation, i.e., \( v \), cannot be simply returned to the lower context—that would be a leak, so \text{toLabeled}(\cdot) wraps that piece of information in a labeled value protected by the final program counter of the sensitive computation, i.e., \text{Labeled } pc' v.\footnote{Stefan et al. [2017] have proposed an alternative flow-insensitive primitive, i.e., \text{toLabeled}(\ell, e), which labels the result with the user-assigned label \( \ell \). The semantics of \( \lambda_{dFG} \) forced us to use \text{toLabeled}(e).}

Furthermore, notice that \( pc' \), the label that tags the result \( v \), is as sensitive as the result itself because the final program counter depends on all the unlabeled operations performed to compute the result. This motivates why primitive \text{labelOf}(\cdot) does not simply project the label from a labeled value, but additionally taints the program counter with the label itself in rule [LabelOf]—a label in a labeled value has sensitivity equal to the label itself, thus the program counter label rises to accommodate reading new sensitive data.

Lastly, rule [GetLabel] returns the value of the program counter, which does not rise (because \( pc \sqcup pc = pc \)), and rule [Taint] simply taints the program counter with the given label and returns unit (this primitive matches the functionality of \text{taint}(\cdot) in \( \lambda_{dFG} \)). Note that, in \( \lambda_{dCG} \), \text{taint}(\cdot) takes only the label with which the program counter must be tainted whereas, in \( \lambda_{dFG} \), it additionally requires the expression that must be evaluated in the tainted environment. This difference highlights the flow-sensitive nature of the program counter label in \( \lambda_{dCG} \).

References. \( \lambda_{dCG} \) features flow-insensitive labeled references similar to \( \lambda_{dFG} \) and allows programs to create, read, update and inspect the label inside the LIO monad (see Figure 8). The API of these primitives takes explicitly labeled values as arguments, by making explicit at the type level, the tagging that occurs in memory, which was left implicit in previous work [Stefan et al. 2017]. Rule [New] creates a reference labeled with the same label annotation as that of the labeled value it receives as an argument, and checks that \( pc \subseteq \ell \) in order to avoid implicit flows. Rule [Read] retrieves the content of the reference from the \( \ell \)-labeled memory and returns it. Since this brings data at security level \( \ell \) in scope, the program counter is tainted accordingly, i.e., \( pc \sqcup \ell \). Rule [Write] performs security checks analogous to those in \( \lambda_{dFG} \) and updates the content of a given reference and rule [LabelOfRef] returns the label on a reference and taints the context accordingly.
\[
\begin{align*}
\text{Labeled } \ell \vdash L & \quad \text{Labeled } v_1 \approx_L v_2 \quad \text{Labeled } \ell_1 \not\vdash L & \quad \text{Labeled } v_1 \approx_L v_2 \quad \text{Labeled } \ell_2 \not\vdash L & \quad \text{Labeled } v_2 \approx_L v_2
\end{align*}
\]

We conclude this section by noting that the forcing and the thunk semantics of \(\lambda^{dCG}\) satisfy Property 2 ("the final value of the program counter is at least as sensitive as the initial value").

**Property 2.**

- If \(\langle \Sigma, pc, e \rangle \not\downarrow^\theta \langle \Sigma', pc', v \rangle\) then \(pc \not\subseteq pc'\).
- If \(\langle \Sigma, pc, t \rangle \not\downarrow^\theta \langle \Sigma', pc', v \rangle\) then \(pc \not\subseteq pc'\).

**Proof.** By mutual induction on the given evaluation derivations.

### 3.2 Security

We now prove that \(\lambda^{dCG}\) is secure, i.e., it satisfies termination-insensitive non-interference. The meaning of the security condition is intuitively similar to that presented in Section 2.2 for \(\lambda^{dFG}\) — when secret inputs are changed, terminating programs do not produce any publicly observable effect—and based on a similar indistinguishability relation. Figure 9 presents the definition of \(L\)-equivalence for the interesting cases only. Firstly, \(L\)-equivalence for \(\lambda^{dCG}\) labeled values relates public and secret values analogously to \(\lambda^{dFG}\) values. Specifically, rule [\text{LabeledL}] relates public labeled values that share the same observable label (\(\ell \subseteq L\)) and contain related values, i.e., \(v_1 \approx_L v_2\), while rule [\text{LabeledH}] relates secret labeled values, with arbitrary sensitivity labels not below \(L\) (\(\ell_1 \not\subseteq L\) and \(\ell_2 \not\subseteq L\)) and contents. Secondly, \(L\)-equivalence relates standard (unlabeled) values homomorphically. For example, values of the sum type are related only as follows: \(\text{inl}(v_1) \approx_L \text{inl}(v_1')\) iff \(v_1 \approx_L v_1'\) and \(\text{inr}(v_2) \approx_L \text{inr}(v_2')\) iff \(v_2 \approx_L v_2'\). Closures and thunks are related if the function and the monadic computations are \(a\)-equivalent and their environments are related, i.e., \(\theta_1 \approx_L \theta_2\) iff \(\text{Dom}(\theta_1) \equiv \text{Dom}(\theta_2)\) and \(\forall x. \theta_1(x) \approx_L \theta_2(x)\). Labeled references, memories and stores are related by \(L\)-equivalence analogously to \(\lambda^{dFG}\). Lastly, \(L\)-equivalence relates initial configurations with related stores, equal program counters and \(a\)-equivalent expressions (resp. thunks), i.e., \(c_1 \approx_L c_2\) iff \(c_1 = \langle \Sigma_1, pc_1, e_1 \rangle, c_2 = \langle \Sigma_2, pc_2, e_2 \rangle, \Sigma_1 \approx_L \Sigma_2, pc_1 \equiv pc_2,\) and \(e_1 \equiv_a e_2\) (resp. \(t_1 \equiv_a t_2\) for thunks \(t_1\) and \(t_2\)), and final configurations with related stores and (i) equal public program counter, i.e., \(pc \subseteq L\), and related values \([\text{PC}_L]\), or (ii) arbitrary secret public counters, i.e., \(pc \not\subseteq L\) and \(pc \not\subseteq L\), and arbitrary values \([\text{PC}_H]\).

We now formally state and prove that \(\lambda^{dCG}\) semantics preserve \(L\)-equivalence under \(L\)-equivalent environments, i.e., termination-insensitive non-interference (TINI).

**Theorem 2 (\(\lambda^{dCG}\)-TINI).** If \(c_1 \not\downarrow^{\theta_1} c_1', c_2 \not\downarrow^{\theta_1} c_2', \theta_1 \approx_L \theta_2\) and \(c_1 \approx_L c_2\) then \(c_1' \approx_L c_2'\).
This section presents a provably semantics-preserving program translation from the fine-grained dynamic IFC calculus $\lambda^{dFG}$ to the coarse-grained calculus $\lambda^{dCG}$. At a high level, the translation performs two tasks (i) it embeds the intrinsic label annotation of $\lambda^{dFG}$ values into an explicitly labeled $\lambda^{dCG}$ value via the Labeled type constructor and (ii) it restructures $\lambda^{dFG}$ side-effectful expressions into monadic operations inside the LIO monad. Our type-driven approach starts by formalizing this intuition in the function $\llcorner \cdot \rrcorner$, which maps the $\lambda^{dFG}$ type $\tau$ to the corresponding $\lambda^{dCG}$ type $\llcorner \tau \rrcorner$ (see Figure 10a). The function is defined by induction on types and recursively adds the Labeled type constructor to each existing $\lambda^{dFG}$ type constructor. For the function type $\tau_1 \rightarrow \tau_2$, the result is additionally monadic, i.e., $\llcorner \tau_1 \rrcorner \rightarrow \text{LIO} \llcorner \tau_2 \rrcorner$. This is because the function’s body in $\lambda^{dFG}$ may have side-effects. The translation for values (Figure 10b) is straightforward. Each $\lambda^{dFG}$ label tag becomes the label annotation in a $\lambda^{dCG}$ labeled value. The translation is homomorphic in the constructors on raw values. The translation converts a $\lambda^{dFG}$ function closure into a $\lambda^{dCG}$ thunk.

Proof. By induction on the derivations.

At this point, we have formalized two calculi—$\lambda^{dFG}$ and $\lambda^{dCG}$—that perform dynamic IFC at fine and coarse granularity, respectively. While they have some similarities, i.e., they are both functional languages that feature labeled annotated data, references and label introspection primitives, and ensure a termination-insensitive security condition, they also have striking differences. First and foremost, they differ in the number of label annotations—pervasive in $\lambda^{dFG}$ and optional in $\lambda^{dCG}$—with significant implications for the programming model and usability. Second, they differ in the nature of the program counter, flow-insensitive in $\lambda^{dFG}$ and flow-sensitive in $\lambda^{dCG}$. Third, they differ in the way they deal with side-effects—$\lambda^{dCG}$ allows side-effectful computations exclusively inside the monad, while $\lambda^{dFG}$ is impure, i.e., any $\lambda^{dFG}$ expression can modify the state. This difference affects the effort required to implement a system that performs language-based fine- and coarse-grained dynamic IFC. In fact, several coarse-grained IFC languages [Buiras et al. 2015; Jaskelioff and Russo 2011; Russo 2015; Russo et al. 2009; Schmitz et al. 2018; Tsai et al. 2007] have been implemented as an embedded domain specific language (EDSL) in a Haskell library with little effort, exploiting the strict control that the host language provides on side-effects. Adapting an existing language to perform fine-grained IFC requires major engineering effort, because several components (all the way from the parser to the runtime system) must be adapted to be label-aware.

In the next two sections we show that—despite their differences—these two calculi are, in fact, equally expressive.
First, notice that the translation of all constructs follows the same principle. The translation simply nests the translations of the nested block. This achieves two goals, (i) it ensures that the value that results from a translated expression is explicitly labeled and (ii) it creates an isolated nested context where the translated thunk can execute without raising the program counter label at the top level. Inside the toLabeled(·) block, the program counter label may rise, e.g., when some intermediate result occurs inside a block, the program counter label at the end of each nested block coincides with the label annotation of the λdFG value that the original expression evaluates to. For example, introduction forms of ground values (unit, labels, and functions) are simply returned inside the toLabeled(·) block so that they get tagged with the current value of the program counter label just as in the corresponding λdFG introduction rules ([LABEL,UNIT,FUN]).

Fig. 11. Translation from λdFG to λdCG (expressions).

Expressions. We show the translation of λdFG expressions to λdCG monadic thunks in Figure 11. We use the standard do notation for readability. First, notice that the translation of all constructs occurs inside a toLabeled(·) block. This achieves two goals, (i) it ensures that the value that results from a translated expression is explicitly labeled and (ii) it creates an isolated nested context where the translated thunk can execute without raising the program counter label at the top level. Inside the toLabeled(·) block, the program counter label may rise, e.g., when some intermediate result is unlabeled, and the translation relies on LIO’s floating-label mechanism to track dependencies between data of different security levels. In particular, we will show later that the value of the program counter label at the end of each nested block coincides with the label annotation of the λdFG value that the original expression evaluates to. For example, introduction forms of ground values (unit, labels, and functions) are simply returned inside the toLabeled(·) block so that they get tagged with the current value of the program counter label just as in the corresponding λdFG introduction rules ([LABEL,UNIT,FUN]).

Syntax do \( x \leftarrow e_1; e_2 \) desugars to \( bind(e_1, x.e_2) \) and syntax \( e_1; e_2 \) to \( bind(e_1, _, e_2) \).

\[
\begin{align*}
\{(\_}\} & = \text{toLabeled}(\text{return}(\_)) \\
\{(\ell)\} & = \text{toLabeled}(\text{return}(\ell)) \\
\{((\lambda x.e))\} & = \text{toLabeled}(\text{return}(\lambda x.e)) \\
\{\text{inl}(e)\} & = \text{toLabeled}(\text{do} \\
& \quad \begin{array}{l}
le \leftarrow e \\
\text{return}(\text{inl}(lv))
\end{array} \\
\{\text{inr}(e)\} & = \text{toLabeled}(\text{do} \\
& \quad \begin{array}{l}
le \leftarrow e \\
\text{return}(\text{inr}(lv))
\end{array} \\
\{\langle e_1, e_2 \rangle\} & = \text{toLabeled}(\text{do} \\
& \quad \begin{array}{l}
lv_1 \leftarrow e_1 \\
lv_2 \leftarrow e_2 \\
\text{return}(\text{lv}(lv_1, lv_2))
\end{array} \\
\{x\} & = \text{toLabeled}(\text{unlabel}(x)) \\
\{e_1, e_2\} & = \text{toLabeled}(\text{do} \\
& \quad \begin{array}{l}
lv_1 \leftarrow e_1 \\
lv_2 \leftarrow e_2 \\
v_1 \leftarrow \text{unlabel}(lv_1) \\
v_2 \leftarrow \text{unlabel}(lv_2) \\
\text{unlabel}(lv)
\end{array} \\
\{\text{case}(e, x, e_1, x. e_2)\} & = \text{toLabeled}(\text{do} \\
& \quad \begin{array}{l}
lv \leftarrow e \\
v \leftarrow \text{unlabel}(lv) \\
lv' \leftarrow \text{case}(v, x. e_1, x. e_2) \\
\text{unlabel}(lv')
\end{array} \\
\{\text{fst}(e)\} & = \text{toLabeled}(\text{do} \\
& \quad \begin{array}{l}
lv \leftarrow e \\
v \leftarrow \text{unlabel}(lv) \\
\text{unlabel}(\text{fst}(v))
\end{array} \\
\{\text{snd}(e)\} & = \text{toLabeled}(\text{do} \\
& \quad \begin{array}{l}
lv \leftarrow e \\
v \leftarrow \text{unlabel}(lv) \\
\text{unlabel}(\text{snd}(v))
\end{array} \\
\{\text{taint}(e_1, e_2)\} & = \text{toLabeled}(\text{do} \\
& \quad \begin{array}{l}
lv_1 \leftarrow e_1 \\
v_1 \leftarrow \text{unlabel}(lv_1) \\
taint(v_1) \\
v_2 \leftarrow e_2 \\
\text{unlabel}(lv_2)
\end{array} \\
\{\text{unlabel}(e)\} & = \text{toLabeled}(\text{do} \\
& \quad \begin{array}{l}
lv \leftarrow e \\
\text{unlabel}(lv)
\end{array} \\
\{\text{getLabel}\} & = \text{toLabeled}(\text{getLabel})
\end{align*}
\]
For example, the function \( \lambda \) demands \( 10a \). The function \( \lambda \) when started with program counter label \( \text{pc} \) evaluated with program counter label equal to \( \text{pc} \sqcup \ell \), which would raise at level, say \( \ell \), and affect the code that translates the argument, which was to be evaluated with program counter label equal to \( \text{pc} \) by the original flow-insensitive \( \lambda \) reduction [App] for function application.

The translation of variables gives some insight into how the \( \lambda \) floating-label mechanism can simulate \( \lambda \)'s tainting approach. First, the type-driven approach set out in Figure \( \text{10a} \) demands that functions take only labeled values as arguments, so the variables in the source program are always associated to a labeled value in the translated program. The values that correspond to these variables are stored in the environment \( \theta \). For example, the code that translates a function application \( \lambda \) gives, i.e., \( \text{Labeled} \ell (\ell r) \) to execute in the tainted context and unlabeling the program counter label \( \ell \). The translation of projection rules, e.g., in rule \( \text{dFG} \), taints the program counter with its label to make it \( \text{pc} \sqcup \ell \), and returns the content, i.e., \( \ell r \). Since \( \text{unlabel}(x) \) occurs inside a \( \text{toLabeled}(\cdot) \) block, the code above results in \( \text{Labeled} \) \( \text{Labeled} \ell (\ell r) \) when evaluated, matching precisely the tainting behavior of \( \lambda \) rule [VAR], i.e., \( x \uparrow^\theta \ell r \) \( \text{rpc} \sqcup \ell \).

The translation of projection rules first converts the \( \lambda \) pair into a computation that gives a \( \lambda \) labeled pair of labeled values, say \( \text{Labeled} \ell (\ell_1 r_1) \text{, Labeled} \ell_2 (\ell_2 r_2) \) and removes the label tag on the pair via \( \text{unlabel} \), thus raising the program counter label to \( \text{pc} \sqcup \ell \). Then, it projects the appropriate component and unlabels it, thus tainting the program counter label even further with the label of either the first or the second component. This coincides with the tainting mechanism of \( \lambda \) for projection rules, e.g., in rule [FST] where \( \text{fst}(e) \uparrow^\theta \text{rpc} \ell_1 r_2 \sqcup \ell \uparrow^\theta \ell_1 \) if \( e \uparrow^\theta \text{rpc} (\ell_1 r_1, \ell_2 r_2) \).

Lastly, translating \( \text{taint}(e_1, e_2) \) requires (i) translating the expression \( e_1 \) that gives the label, (ii) using \( \text{taint}(\cdot) \) from \( \lambda \) to explicitly taint the program counter label with the label that \( e_1 \) gives, and (iii) translating the second argument \( e_2 \) to execute in the tainted context and unlabeled the result. The construct \( \text{labelOf}(e) \) of \( \lambda \) uses the corresponding \( \lambda \) primitive applied on the corresponding labeled value, say \( \text{Labeled} \ell (\ell r) \), obtained from the translated expression. Notice

\[
\frac{(\text{WkenType})}{\Gamma \vdash \overline{x} : \tau} \quad \frac{(\text{Wken})}{e_\theta \vdash \overline{x} : \nu} \quad \frac{(\text{Wken})}{\overline{x} \vdash \text{wken} \nu_1 : \tau}
\]

Fig. 12. Typing and semantics rules of \text{wken} for \( \lambda \).

\[\text{weaken} \]

expressions inside the same constructor, without raising the program counter label. This matches the behavior of the corresponding \( \lambda \) rules [INL,INR,PAIR]. For example, the \( \lambda \) reduction \( (\ell, r) \downarrow^\theta \ell \) maps to a \( \lambda \) reduction that yields \( \text{Labeled} \ell \) \( \text{Labeled} \ell (\ell r) \) when started with program counter label \( \ell \).

\[\text{weaken} \]

The translation of variables gives some insight into how the \( \lambda \) floating-label mechanism can simulate \( \lambda \)'s tainting approach. First, the type-driven approach set out in Figure \( \text{10a} \) demands that functions take only labeled values as arguments, so the variables in the source program are always associated to a labeled value in the translated program. The values that correspond to these variables are stored in the environment \( \theta \). For example, the code that translates a function application \( \lambda \) gives, i.e., \( \text{Labeled} \ell (\ell r) \) to execute in the tainted context and unlabeling the program counter label \( \ell \). The translation of projection rules, e.g., in rule \( \text{dFG} \), taints the program counter with its label to make it \( \text{pc} \sqcup \ell \), and returns the content, i.e., \( \ell r \). Since \( \text{unlabel}(x) \) occurs inside a \( \text{toLabeled}(\cdot) \) block, the code above results in \( \text{Labeled} \) \( \text{Labeled} \ell (\ell r) \) when evaluated, matching precisely the tainting behavior of \( \lambda \) rule [VAR], i.e., \( x \uparrow^\theta \ell r \) \( \text{rpc} \sqcup \ell \).

The translation of projection rules first converts the \( \lambda \) pair into a computation that gives a \( \lambda \) labeled pair of labeled values, say \( \text{Labeled} \ell (\ell_1 r_1) \text{, Labeled} \ell_2 (\ell_2 r_2) \) and removes the label tag on the pair via \( \text{unlabel} \), thus raising the program counter label to \( \text{pc} \sqcup \ell \). Then, it projects the appropriate component and unlabels it, thus tainting the program counter label even further with the label of either the first or the second component. This coincides with the tainting mechanism of \( \lambda \) for projection rules, e.g., in rule [FST] where \( \text{fst}(e) \uparrow^\theta \text{rpc} \ell_1 r_2 \sqcup \ell \uparrow^\theta \ell_1 \) if \( e \uparrow^\theta \text{rpc} (\ell_1 r_1, \ell_2 r_2) \).

Lastly, translating \( \text{taint}(e_1, e_2) \) requires (i) translating the expression \( e_1 \) that gives the label, (ii) using \( \text{taint}(\cdot) \) from \( \lambda \) to explicitly taint the program counter label with the label that \( e_1 \) gives, and (iii) translating the second argument \( e_2 \) to execute in the tainted context and unlabeled the result. The construct \( \text{labelOf}(e) \) of \( \lambda \) uses the corresponding \( \lambda \) primitive applied on the corresponding labeled value, say \( \text{Labeled} \ell (\ell r) \), obtained from the translated expression. Notice

\[\text{weaken} \]

We name a variable \( lv \) if it gets bound to a labeled value, i.e., to indicate that the variable has type \( \text{Labeled} \tau \). For example, \( x \) is translated to \( \text{Labeled} \ell r \) when evaluated, matching precisely the tainting behavior of \( \lambda \) rule [VAR], i.e., \( x \uparrow^\theta \ell r \) \( \text{rpc} \sqcup \ell \).

Notice that it is incorrect to unlabel the function before translating the argument, because that would taint the program counter label, which would raise at level, say \( \text{pc} \sqcup \ell \), and affect the code that translates the argument, which was to be evaluated with program counter label equal to \( \text{pc} \) by the original flow-insensitive \( \lambda \) rule [App] for function application.
\[
\begin{align*}
\langle \text{new}(e) \rangle &= \text{toLabeled}(\text{do}) \\
&\quad \langle \text{do} \rangle \\
&\quad \langle e \rangle \\
&\quad \langle \text{do} \rangle
\end{align*}
\]

\[
\begin{align*}
&\lambda \text{toLabeled}(\text{do}) \\
&\quad \langle \text{do} \rangle \\
&\quad \langle e_1 \rangle \\
&\quad \langle \text{do} \rangle \\
&\quad \langle e_2 \rangle
\end{align*}
\]

\[
\begin{align*}
&\langle \text{labelOfRef}(e) \rangle = \text{toLabeled}(\text{do}) \\
&\quad \langle \text{do} \rangle \\
&\quad \langle e \rangle \\
&\quad \langle \text{do} \rangle \\
&\quad \langle \text{do} \rangle
\end{align*}
\]

Fig. 13. Translation \(\lambda^{dFG}\) to \(\lambda^{dCG}\) (references).

that \(\text{labelOf}(\cdot)\) taints the program counter label in \(\lambda^{dCG}\), which rises to \(pc \sqcup \ell\), so the code just described results in \(\text{Labeled}(pc \sqcup \ell)\), which corresponds to the translation of the result in \(\lambda^{dFG}\), i.e., \(\langle \ell' \rangle = \text{Labeled} \ell \ell\) because \(pc \sqcup \ell \equiv \ell\), since \(pc \sqsubseteq \ell\) from Property 1. The translation of \(\text{getLabel}\) follows naturally by simply wrapping \(\lambda^{dCG}\)'s \(\text{getLabel}\) inside a \(\text{toLabeled}(\cdot)\), which correctly returns the program counter label labeled with itself, i.e., \(\text{Labeled} pc\).

Note on Environments. The semantics rules of \(\lambda^{dFG}\) and \(\lambda^{dCG}\) feature an environment \(\theta\) for input values that gets extended with intermediate values during program evaluation and that may be captured inside a closure. Unfortunately, this capturing behavior is undesirable for our program translation. The program translation defined above introduces temporary auxiliary variables that carry the value of intermediate results, e.g., the labeled value obtained from running a computation that translates some \(\lambda^{dFG}\) expression. When the translated program is executed, these values end up in the environment, e.g., by means of rules \([\text{App}]\) and \([\text{Bind}]\), and mix with the input values of the source program and output values as well, thus complicating the correctness statement of the translation, which now has to account for those extra variables as well. In order to avoid this nuisance, we employ a special form of weakening that allows shrinking the environment at run-time and removing spurious values that are not needed in the rest of the program. In particular, expression \(\text{wken} \bar{x} e\) has the same type as \(e\) if variables \(\bar{x}\) are not free in \(e\), see the formal typing rule \([\text{WkenType}]\) in Figure 12. At run-time, the expression \(\text{wken} \bar{x} e\) evaluates \(e\) in an environment from which variables \(\bar{x}\) have been dropped, so that they do not get captured in any closure created during the execution of \(e\). Rule \([\text{Wken}]\) is part of the pure semantics of \(\lambda^{dCG}\)—the semantics of \(\lambda^{dFG}\) includes an analogous rule (the issue of contaminated environments arises in the translations in both directions, thus both calculi feature \(\text{wken}\)). We remark that this expedient is not essential—we can avoid it by slightly complicating the correctness statement to explicitly account for those extra variables. Nor is this expedient particularly interesting. In fact, we omit \(\text{wken}\) from the code of the program translations to avoid clutter (our mechanization includes \(\text{wken}\) in the appropriate places).

References. Figure 13 shows the program translation of \(\lambda^{dFG}\) primitives that access the store via references. The translation of \(\lambda^{dFG}\) values wraps references in \(\lambda^{dCG}\) labeled values (Figure 10b), so the translations of Figure 13 take care of boxing and unboxing references. The translation of \(\text{new}(e)\) has a top-level \(\text{toLabeled}(\cdot)\) block that simply translates the content \((lv \leftarrow \langle e \rangle)\) and puts it in a new reference \((\text{new}(lv))\). The \(\lambda^{dCG}\) rule \([\text{New}]\) (Figure 8) assigns the label of the translated content to the new reference, which also gets labeled with the original program counter label\(^{15}\), just as in the \(\lambda^{dFG}\) rule \([\text{New}]\) (Figure 4). In \(\lambda^{dFG}\), rule \([\text{Read}]\) reads from a reference \(n_{\ell \ell'}\) at security level \(\ell'\) that points to the \(\ell\)-labeled memory, and returns the content \(\Sigma(\ell)[n_{\ell \ell'}]\) at level \(\ell \sqcup \ell'\). Similarly, the translation creates a \(\text{toLabeled}(\cdot)\) block that executes to get a labeled reference \(lr = \text{Labeled} \ell' n_{\ell},\)

\(^{15}\)The nested block does not execute any \(\text{unlabel}(\cdot)\) nor \(\text{taint}(\cdot)\).
extracts the reference \( n_ℓ (r ← \text{unlabel}(lr)) \) tainting the program counter label with \( ℓ' \), and then reads the reference’s content further tainting the program counter label with \( ℓ \) as well. The code that translates and updates a reference consists of two \texttt{toLabeled()} blocks. The first block is responsible for the update: it extracts the labeled reference and the labeled new content \((lr \text{ and } lv \text{ resp.}), \) extracts the reference from the labeled value \((r ← \text{unlabel}(lr)) \) and updates it \((r := lv)\). The second block, \texttt{toLabeled(return())}, returns unit at security level \( pc \), i.e., \texttt{Labeled pc ()}, similar to the \( \lambda_{dFG} \) rule \[\text{[Write]}\]. The translation of \texttt{labelOfRef(e)} extracts the reference and projects its label via the \( \lambda_{dCG} \) primitive \texttt{labelOfRef(·)}, which additionally taints the program counter with the label itself, similar to the \( \lambda_{dFG} \) rule \[\text{[LABELOfRef]}\].

### 4.1 Correctness

In this section, we establish some desirable properties of the \( \lambda_{dFG} \)-to-\( \lambda_{dCG} \) translation defined above. These properties include type and semantics preservation as well as recovery of non-interference—a meta criterion that rules out a class of semantically correct (semantics preserving), yet elusive translations that do not preserve the meaning of security labels [Barthe et al. 2007; Rajani and Garg 2018].

We start by showing that the program translation preserves typing. The type translation for typing contexts \( Γ \) is pointwise, i.e., \( ⟨⟨Γ⟩⟩ = λx.⟨⟨Γ(χ)⟩⟩ \).

**Lemma 4.1 (Type Preservation).** Given a well-typed \( \lambda_{dFG} \) expression, i.e., \( Γ ⊢ e : τ \), the translated \( \lambda_{dCG} \) expression is also well-typed, i.e., \( ⟨⟨Γ⟩⟩ ⊢ ⟨⟨e⟩⟩ : \text{LIO}⟨⟨τ⟩⟩ \).

**Proof.** By induction on the given typing derivation. \( \square \)

The main correctness criterion for the translation is semantics preservation. Intuitively, proving this theorem ensures that the program translation preserves the meaning of secure \( \lambda_{dFG} \) programs when translated and executed with \( \lambda_{dCG} \) semantics (under a translated environment). In the theorem below\(^{16}\), the translation of stores and memories is pointwise, i.e., \( ⟨⟨Σ⟩⟩ = ℓτ.⟨⟨Σ(ℓ)⟩⟩ \), and \( ⟨⟨[]⟩⟩ = [] \) and \( ⟨⟨r : M⟩⟩ = ⟨⟨r⟩⟩ : ⟨⟨M⟩⟩ \) for each \( ℓ \)-labeled memory \( M \). Furthermore, notice that in the translation, the initial and final program counter labels are the same. This establishes that the program translation preserves the flow-insensitive program counter label of \( \lambda_{dFG} \) (by means of primitive \texttt{toLabeled(·)}).

**Theorem 3 (Semantics Preservation of \( ⟨•⟩ \): \( \lambda_{dFG} → \lambda_{dCG} \)).** Given a well-typed \( \lambda_{dFG} \) program \( ⟨Σ, e⟩ \downarrow_{pc}^θ ⟨Σ', v⟩ \), then \( ⟨⟨Σ⟩⟩, pc, ⟨⟨e⟩⟩⟩ \downarrow_{θ}^Θ ⟨⟨Σ'⟩⟩, pc, ⟨⟨v⟩⟩⟩ \).

**Proof.** By induction on the given evaluation derivation using basic properties of the security lattice and of the translation function.

**Recovery of non-interference.** We conclude this section by constructing a proof of termination-insensitive non-interference for \( \lambda_{dFG} \) (Theorem 1) from the corresponding theorem for \( \lambda_{dCG} \) (Theorem 2), using the semantics preserving translation (Theorem 3), together with a property that the translation preserves \( L \)-equivalence as well (Lemmas 4.2 and 4.3). Doing so ensures that the meaning of labels is preserved by the translation [Barthe et al. 2007; Rajani and Garg 2018]. In the absence of such an artifact, one could devise a semantics-preserving translation that simply does not use the security features of the target language. While technically correct (i.e., semantics preserving), the translation would not be meaningful from the perspective of security.\(^{17}\) The following lemma

---

\(^{16}\) The proof of Theorem 3 requires the (often used) axiom of functional extensionality in our mechanized proofs.

\(^{17}\) Note that such bogus translations are also ruled out due to the need to preserve the outcome of any label introspection. Nonetheless, building this proof artifact increases our confidence in the robustness of our translation. In contrast, if the
shows that the translation of $\lambda^{dFG}$ initial configurations, defined as $\langle\langle c \rangle\rangle_{PC} = \langle\langle \Sigma, pc, \langle\langle e \rangle\rangle \rangle$ if $c = \langle\Sigma, e \rangle$, preserves $L$-equivalence by lifting $L$-equivalence from source to target and back.

**Lemma 4.2.** For all program counter labels $pc$, $c_1 \approx_L c_2$ if and only if $\langle\langle c_1 \rangle\rangle_{PC} \approx_L \langle\langle c_2 \rangle\rangle_{PC}$.

**Proof.** By definition of $L$-equivalence for initial configurations in both directions (Sections 2.2 and 3.2), using injectivity of the translation function, i.e., if $\langle\langle e_1 \rangle\rangle = \approx_L \langle\langle e_2 \rangle\rangle$ then $e_1 = \approx_L e_2$, in the if direction, and by mutually proving similar lemmas for all categories: for stores, i.e., $\Sigma_1 \approx_L \Sigma_2$ iff $\langle\langle \Sigma_1 \rangle\rangle = \approx_L \langle\langle \Sigma_2 \rangle\rangle$, for memories, i.e., $M_1 \approx_L M_2$ iff $\langle\langle M_1 \rangle\rangle = \approx_L \langle\langle M_2 \rangle\rangle$, for environments, i.e., $\theta_1 \approx_L \theta_2$ iff $\langle\langle \theta_1 \rangle\rangle = \approx_L \langle\langle \theta_2 \rangle\rangle$, for values, i.e., $v_1 \approx_L v_2$ iff $\langle\langle v_1 \rangle\rangle = \approx_L \langle\langle v_2 \rangle\rangle$, and for raw values, i.e., $r_1 \approx_L r_2$ iff $\langle\langle r_1 \rangle\rangle = \approx_L \langle\langle r_2 \rangle\rangle$.

The following lemma recovers $L$-equivalence of source final configurations by back-translating $L$-equivalence of target final configurations. We define the translation for $\lambda^{dFG}$ final configurations as $\langle\langle c \rangle\rangle_{PC} = \langle\langle \Sigma, pc, \langle\langle v \rangle\rangle \rangle$ if $c = \langle\Sigma, v \rangle$.

**Lemma 4.3.** Let $c_1 = \langle\Sigma_1, r_1^{f_1} \rangle$ and $c_2 = \langle\Sigma_2, r_2^{f_2} \rangle$ be $\lambda^{dFG}$ final configurations. For all program counter label $pc$, such that $pc \not\subseteq \ell_1$ and $pc \not\subseteq \ell_2$, if $\langle\langle c_1 \rangle\rangle_{PC} \approx_L \langle\langle c_2 \rangle\rangle_{PC}$ then $c_1 \approx_L c_2$.

**Proof.** By case analysis on the $L$-equivalence relation of the target final configurations, two cases follow. First, we recover $L$-equivalence of the source stores, i.e., $\Sigma_1 \approx_L \Sigma_2$, from that of the target stores, i.e., $\langle\langle \Sigma_1 \rangle\rangle \approx_L \langle\langle \Sigma_2 \rangle\rangle$ from $\langle\langle c_1 \rangle\rangle = \approx_L \langle\langle c_2 \rangle\rangle$ in both cases. Then, the program counter in the target configurations is either (i) above the attacker’s level [PC$_H$], i.e., $pc \not\subseteq L$, and the source values are $L$-equivalent, i.e., $r_1^{f_1} \approx_L r_2^{f_2}$ by rule [VALUE$_H$] applied to $\ell_1 \not\subseteq L$ and $\ell_2 \not\subseteq L$ (from $pc \not\subseteq L$ and, respectively, $pc \not\subseteq \ell_1$ and $pc \not\subseteq \ell_2$), or (ii) below the attacker’s level [PC$_L$], i.e., $pc \subseteq L$, then $\langle\langle r_1^{f_1} \rangle\rangle = \approx_L \langle\langle r_2^{f_2} \rangle\rangle$ and the source values are $L$-equivalent, i.e., $r_1^{f_1} \approx_L r_2^{f_2}$, by Lemma 4.2 for values.

**Theorem 4.** ($\lambda^{dFG}$-TIN via $\langle\langle \cdot \rangle\rangle$). If $c_1 \ll_p \theta_1 c_1, c_2 \ll_p \theta_2 c_2$, $\theta_1 \approx_L \theta_2$, and $c_1 \approx_L c_2$ then $c_1' \approx_L c_2'$.

**Proof.** We start by applying the fine to coarse grained program translation and the initial configurations and environments. By Theorem 3 (semantics preservation), we derive the corresponding $\lambda^{dCG}$ reductions, i.e., $\langle\langle c_1 \rangle\rangle_{PC} \ll_p \theta_1 \langle\langle c_1' \rangle\rangle_{PC}$ and $\langle\langle c_2 \rangle\rangle_{PC} \ll_p \theta_2 \langle\langle c_2' \rangle\rangle_{PC}$. Then, we lift $L$-equivalence of the initial configurations and environments from source to target, i.e., from $c_1 \approx_L c_2$ to $\langle\langle c_1 \rangle\rangle_{PC} \approx_L \langle\langle c_2 \rangle\rangle_{PC}$ and from $\theta_1 \approx_L \theta_2$ to $\langle\langle \theta_1 \rangle\rangle \approx_L \langle\langle \theta_2 \rangle\rangle$ (Lemma 4.2), and apply $\lambda^{dCG}$-TINI (Theorem 2) to obtain $L$-equivalence of the target final configurations, i.e., $\langle\langle c_1' \rangle\rangle_{PC} \approx_L \langle\langle c_2' \rangle\rangle_{PC}$. Finally, we recover $L$-equivalence of the final configurations from source to target, i.e., from $\langle\langle c_1' \rangle\rangle_{PC} \approx_L \langle\langle c_2' \rangle\rangle_{PC}$ to $\theta_1 \approx_L \theta_2$, via Lemma 4.3, applied to $c_1' = \langle\Sigma_1, r_1^{f_1} \rangle$ and $c_2' = \langle\Sigma_2, r_2^{f_2} \rangle$, and where $pc \subseteq \ell_1$ and $pc \subseteq \ell_2$ by Property 1 applied to the source reductions, i.e., $c_1 \ll_p \theta_1 c_1'$ and $c_2 \ll_p \theta_2 c_2'$.

## 5 COARSE- TO FINE-GRAINED PROGRAM TRANSLATION

We now show a verified program translation in the opposite direction—from the coarse-grained calculus $\lambda^{dCG}$ to the fine-grained calculus $\lambda^{dFG}$. The translation in this direction is more involved—a program in $\lambda^{dFG}$ contains strictly more information than its counterpart in $\lambda^{dCG}$, namely the extra intrinsic label annotations that tag every value. The challenge in constructing this translation is two-fold. On one hand, the translation must come up with labels for all values. However, it is not always possible to do this statically during the translation: Often, the labels depend on input values and arise at run-time with intermediate results since the $\lambda^{dFG}$ calculus is designed to compute and attach labels at run-time. On the other hand, the translation cannot conservatively enforce...
under-approximate the values of labels\textsuperscript{18}—$\lambda^{dCG}$ and $\lambda^{dFG}$ have label introspection so, in order to get semantics preservation, labels must be preserved precisely. Intuitively, we solve this impasse by crafting a program translation that (i) preserves the labels that can be inspected by $\lambda^{dCG}$ and (ii) lets the $\lambda^{dFG}$ semantics compute the remaining label annotations automatically—we account for those labels with a cross-language relation that represents semantic equivalence between $\lambda^{dFG}$ and $\lambda^{dCG}$ modulo extra annotations (Section 5.1). The fact that the source program in $\lambda^{dCG}$ cannot inspect those labels—they have no value counterpart in the source $\lambda^{dCG}$ program—facilitates this aspect of the translation. We elaborate more on the technical details later.

At a high level, an interesting aspect of the translation (that informally attests that it is indeed semantics-preserving) is that it encodes the flow-sensitive program counter of the source $\lambda^{dCG}$ program into the label annotation of the $\lambda^{dFG}$ value that results from executing the translated program. For example, if a $\lambda^{dCG}$ monadic expression starts with program counter label $pc$ and results in some value, say $\text{true}$, and final program counter $pc'$, then the translated $\lambda^{dFG}$ expression, starting with the same program counter label $pc$, computes the same value (modulo extra label annotations) at the same security level $pc'$, i.e., the value $\text{true}^{pc'}$. This encoding requires keeping the value of the program counter label in the source program synchronized with the program counter label in the target program, by loosening the fine-grained precision of $\lambda^{dFG}$ at run-time in a controlled way.

Types. The $\lambda^{dCG}$-to-$\lambda^{dFG}$ translation follows the same type-driven approach used in the other direction, starting from the function $\llbracket \cdot \rrbracket$ in Figure 14a, that translates a $\lambda^{dFG}$ type $\tau$ into the corresponding $\lambda^{dCG}$ type $\llbracket \tau \rrbracket$. The translation is defined by induction on $\tau$ and preserves all the type constructors standard types. Only the cases corresponding to $\lambda^{dCG}$-specific types are interesting. In particular, it converts explicitly labeled types, i.e., Labeled $\tau$, to a standard pair type in $\lambda^{dFG}$, i.e., $(\mathcal{L} \times \llbracket \tau \rrbracket)$, where the first component is the label and the second component the content of type $\tau$. Type LIO $\tau$ becomes a suspension in $\lambda^{dFG}$, i.e., the function type unit $\rightarrow \llbracket \tau \rrbracket$ that delays a computation and that can be forced by simply applying it to the unit value $()$.

Values. The translation of values follows the type translation, as shown in Figure 14b. Notice that the translation is indexed by the program counter label (the translation is written $\llbracket v \rrbracket^{pc}$).

\textsuperscript{18}In contrast, previous work on static type-based fine-to-coarse grained translation safely under-approximates the label annotations in types with $\bot$ [Rajani and Garg 2018]. The proof of type preservation of the translation recovers the actual labels via subtyping.

Fig. 14. Translation from $\lambda^{dCG}$ to $\lambda^{dFG}$ (part I).
which converts the $\lambda^{dCG}$ value $v$ in scope of a computation protected by security level $pc$ to the corresponding fully label-annotated $\lambda^{dFG}$ value. The translation is pretty straightforward and uses the program counter label to tag each value, following the $\lambda^{dCG}$ principle that the program counter label protects every value in scope that is not explicitly labeled. The translation converts a $\lambda^{dCG}$ function closure into a corresponding $\lambda^{dFG}$ function closure by translating the body of the function to a $\lambda^{dFG}$ expression (see below) and translating the environment pointwise, i.e., $\llbracket \theta \rrbracket^{pc} = \lambda x. [\llbracket \theta(x) \rrbracket]^{pc}$. A thunk value or a thunk closure, which denotes a suspended side-effectful computation, is also converted into a $\lambda^{dFG}$ function closure. Technically, the translation would need to introduce a fresh variable that would get bound to unit when the suspension gets forced. However, the argument to the suspension does not have any purpose, so we do not bother with giving a name to it and write $\lambda \_ \_ \ ([t])$ instead. (In our mechanized proofs we employ unnamed De Bruijn indexes and this issue does not arise.) The translation converts an explicitly labeled value Labeled $\ell$ $v$, into a labeled pair at security level $pc$, i.e., $(\ell^{\ell}, [v]^{\ell})^{pc}$. The pair consists of the label $\ell$ tagged with itself, and the value translated at a security level equal to the label annotation, i.e., $[\llbracket v \rrbracket]^{\ell}$. Notice that tagging the label with itself allows us to translate the $\lambda^{dCG}$ (label introspection) primitive $\text{labelOf}(\cdot)$ by simply projecting the first component, thus preserving the label and its security level across the translation.

**Expressions and Thunks.** The translation of pure expressions (Figure 15a) is trivial: it is homomorphic in all constructs, mirroring the type translation. The translation of a thunk expression $t$ builds a suspension explicitly with a $\lambda$-abstraction (the name of the variable is again irrelevant, thus we omit it as explained above), and carries on by translating the thunk itself according to the definition in Figure 15b. The thunk return becomes $[e]$, since return($\cdot$) does not have any side-effect. When two monadic computations are combined via bind($e_1, x, e_2$), the translation (i) converts the first computation to a suspension and forces it by applying unit ($\llbracket e_1 \rrbracket()$), (ii) binds the result to $x$ and passes it to the second computation\footnote{Syntax let $x = e_1$ in $e_2$ where $x$ is free in $e_2$ is a shorthand for $(\lambda x. e_2)\ e_1$.}, which is also converted, forced, and, importantly, iii) executed with a program counter label tainted with the security level of the result.

---

| $([], x) = ()$ | $[\text{return}(e)] = [e]$ |
| $[\ell] = \ell$ | $\text{bind}(e_1, x, e_2) = \llbracket e_1 \rrbracket [x]$ |
| $[x] = x$ | $\text{taint}([\text{labelOf}(x), [e_2]])$ |
| $[\lambda x. e] = \lambda x. [e]$ | $\text{unlabel}(e) = \llbracket e \rrbracket$ |
| $[e_1, e_2] = [e_1][e_2]$ | $\text{let } x = [e_1]() \text{ in } \text{taint}([\text{fst}(x), \text{snd}(x)])$ |
| $[(e_1, e_2)] = ([e_1], [e_2])$ | $\text{toLabeled}(e) = \llbracket e \rrbracket() \text{ in } \text{labelOf}(x, e)$ |
| $\text{fst}(e) = \text{fst}([e])$ | $\text{getLabel} = \text{getLabel}$ |
| $\text{snd}(e) = \text{snd}([e])$ | $\text{taint}(e) = \text{taint}([e], ()]$ |
| $\text{inl}(e) = \text{inl}([e])$ |   |
| $\text{inr}(e) = \text{inr}([e])$ |   |
| $\text{case} (e, x, e_1, x, e_2) = \text{case} ([e], x, [e_1], x, [e_2])$ |   |
| $[t] = \lambda \_\_\_ [t]$ |   |

(a) Expressions. | (b) Thunks.
of the first computation (\texttt{taint}(\texttt{labelOf}(e), \llbracket e_2 \rrbracket))]. Notice that \texttt{taint}(\cdot) is essential to ensure that the second computation executes with the program counter label set to the correct value—the precision of the fine-grained system would otherwise retain the initial lower program counter label according to rule [\texttt{App}] and the value of the program counter labels in the source and target programs would differ in the remaining execution.

Similarly, the translation of \texttt{unlabel}(e) first translates the labeled expression \(e\) (the translated expression does not need to be forced because it is not of a monadic type), binds its result to \(x\) and then projects the content in a context tainted with its label, as in \texttt{taint(fst}(x), \texttt{snd}(x)). This closely follows \(\lambda^{dCG}\)'s [\texttt{UNLABEL}] rule. The translation of \texttt{toLabeled}(e) forces the nested computation with \(\llbracket e \rrbracket\), binds its result to \(x\) and creates the pair \(\texttt{labelOf}(e, x)\), which corresponds to the labeled value obtained in \(\lambda^{dCG}\) via rule [\texttt{ToLabeled}]. Intuitively, the translation guarantees that the value of the final program counter label in the nested computation coincides with the security level of the translated result (bound to \(x\)). Therefore, the first component contains the correct label and it is furthermore at the right security level, because \texttt{labelOf}(\cdot) protects the projected label with the label itself in \(\lambda^{dFG}\). Primitive \texttt{labelOf}(e) simply projects the first component of the pair that encodes the labeled value in \(\lambda^{dFG}\) as explained above. Lastly, \texttt{getLabel} in \(\lambda^{dCG}\) maps directly to \texttt{getLabel} in \(\lambda^{dFG}\)—rule [\texttt{GetLABEL}] in \(\lambda^{dCG}\) simply returns the program counter label and does not raise its value, so it corresponds exactly to rule [\texttt{GetLABEL}] in \(\lambda^{dFG}\), which returns label \(pc\) at security level \(\ell\). Similarly, \texttt{taint}(e) translates to \texttt{taint(\llbracket e \rrbracket, \langle\rangle)}, since rule [\texttt{Taint}] in \(\lambda^{dCG}\) taints the program counter with the label that \(e\) evaluates to, say \(\ell\) and returns unit with program counter label equal to \(pc \sqcup \ell\), which corresponds to the result of the translated program, i.e., \(\langle\rangle)^{pc \sqcup \ell}\).

**References.** Figure 16 shows the translation of primitives that access the store via references. Since \(\lambda^{dCG}\)'s rule [\texttt{New}] in Figure 8 creates a new reference labeled with the label of the argument (which must be a labeled value), the translation converts \texttt{new}(e) to an expression that first binds \(\llbracket e \rrbracket\) to \(x\) and then creates a new reference with the same content as the source, i.e., \texttt{snd}(x), but tainted with the label in \(x\), i.e., \texttt{fst}(x). Notice that the use of \texttt{taint}(\cdot) is essential to ensure that \(\lambda^{dFG}\)'s rule [\texttt{New}] in Figure 4 assigns the correct label to the new reference. Due to its fine-grained precision, \(\lambda^{dFG}\) might have labeled the content with a different label that is less sensitive than the explicit label that coarsely approximates the security level in \(\lambda^{dCG}\). In contrast, updating a reference does not require any tainting—both \(\lambda^{dFG}\) and \(\lambda^{dCG}\) accept values less sensitive than the reference in rule [\texttt{Write}]. Thus, the translation \(e_1 := e_2\) simply updates the translated reference with the content of the labeled value projected from the translated pair, hence \(\llbracket e_1 := e_2 \rrbracket\) is \(\llbracket e_1 \rrbracket := \texttt{snd}(\llbracket e_2 \rrbracket)\). The translation of the primitives that read and query the label of a reference is trivial.

### 5.1 Cross-Language Semantic Equivalence up to Extra Annotations

When a \(\lambda^{dCG}\) program is translated to \(\lambda^{dFG}\) via the program translation described above, the \(\lambda^{dFG}\) result contains strictly more information than the original \(\lambda^{dCG}\) result. This happens because the semantics of \(\lambda^{dFG}\) tracks flows of information at fine granularity, in contrast with \(\lambda^{dCG}\), which instead coarsely approximates the security level of all values in scope of a computation with the

\[
\begin{align*}
\llbracket \texttt{new}(e) \rrbracket &= \texttt{let} x = \llbracket e \rrbracket \texttt{in} \\
&\quad \texttt{new(taint(fst}(x), \texttt{snd}(x))) \\
\llbracket \texttt{e_1 := e_2} \rrbracket &= \llbracket e_1 \rrbracket := \texttt{snd}(\llbracket e_2 \rrbracket) \\
\llbracket \texttt{! e} \rrbracket &= \llbracket x \rrbracket \texttt{=} \llbracket e \rrbracket \\
\llbracket \texttt{labelOfRef}(e) \rrbracket &= \texttt{labelOfRef}(\llbracket e \rrbracket)
\end{align*}
\]
program counter label. When translating a $\lambda^{dCG}$ program, we can capture this condition precisely for input values $\theta$ by homogeneously tagging all standard (unlabeled) values with the initial program counter label, i.e., $[\theta]^{pc}$. However, a $\lambda^{dCG}$ program handles, creates and mixes unlabeled data that originated at different security levels at run-time, e.g., when a secret is unlabeled and combined with previously public (unlabeled) data. Crucially, when the translated program executes, the fine-grained semantics of $\lambda^{dCG}$ tracks those flows of information precisely and thus new labels appear (these labels do not correspond to the label of any labeled value in the source value nor to the program counter label). Intuitively, this implies that the $\lambda^{dFG}$ result will not be homogeneously labeled with the final program counter label of the $\lambda^{dCG}$ computation, i.e., if a $\lambda^{dCG}$ program terminates with value $v$ and program counter label $pc'$, the translated $\lambda^{dFG}$ program does not necessarily result in $[v]^{pc'}$.

**Example.** Consider the $\lambda^{dCG}$ program $⟨\Sigma, L, \text{taint}(H); \text{return}(x)⟩ \downarrow^{x→\text{true}} ⟨\Sigma, H, \text{true}⟩$, which returns $\text{true} = \text{inl}(l)$ and the store $\Sigma$ unchanged, after tainting the program counter label with $H$. Let $e$ be the expression obtained by applying the program translation from Figure 15 to the example program:

\[
e = \lambda_\_.
\]

\[
\text{let } y = \text{taint}(H, ()) \text{ in }
\]

\[
\text{taint(labelOf}(y), x)
\]

Interestingly, when we force the program $e$ and execute it starting from program counter label equal to $L$, and an input environment translated according to the initial program counter label ($L$ in this case), i.e., $x→[\text{true}]^L = \text{inl}([()])^L = \text{true}^L$, we do not obtain the translated result homogeneously labeled with $H$:

\[
⟨[\Sigma], e(())⟩ \downarrow^{x→\text{true}^L} ⟨[\Sigma], \text{true}^H⟩ = ⟨[\Sigma], \text{inl}(())^H⟩ ≠ ⟨[\Sigma], \text{inl}([()])^H⟩ = ⟨[\Sigma], [\text{true}]^H⟩
\]

In particular, $\lambda^{dFG}$ preserves the public label tag on data nested inside the left injection, i.e., $()$ in $\text{inl}([()])^H$ above. This happens because $\lambda^{dFG}$’s rule $[\text{Var}]$ taints only the outer label of the value $\text{true}^L$ when it looks up variable $x$ in program counter label $H$.

**Solution.** In order to recover a notion of semantics preservation, we introduce a key contribution of this work, a cross-language binary relation that associates values of the two calculi that, in the scope of a computation at a given security level, are semantically equivalent up to the extra
annoyances present in the $\lambda^{dFG}$ value.\textsuperscript{20} Technically, we use this equivalence in the semantics preservation theorem in Section 5.2, which existentially quantifies over the result of the translated $\lambda^{dFG}$ program, but guarantees that it is semantically equivalent to the result obtained in the source program.

Concretely, for a $\lambda^{dFG}$ value $v_1$ and a $\lambda^{dCG}$ value $v_2$, we write $v_1 \approx_{pc} v_2$ if the label annotations (including those nested inside compound values) of $v_1$ are no more sensitive than label $pc$ and its raw value corresponds to $v_2$. Figure 17 formalizes this intuition by means of two mutually inductive relations, one for $\lambda^{dFG}$ values and one for $\lambda^{dFG}$ raw values. In particular, rule [VALUE] relates $\lambda^{dFG}$ value $r_1 \ell_1$ and $\lambda^{dCG}$ value $v_2$ at security level $pc$ if the label annotation on the raw value $r_1$ flows to the program counter label, i.e., $\ell_1 \subseteq pc$, and if the raw value is in relation with the standard value, i.e., $r_1 \approx_{pc} v_2$. The relation between raw values and standard values relates only semantically equivalent values, i.e., it is syntactic equality for ground types ([UNIT, LABEL, REF]), requires the same injection for values of the sum type ([INL, INR]) and requires the components to related to pairs ([PAIR]).

Rules [FUN] (resp. [THUNK]) relates function (resp. thunk) closures only when environments are related pointwise, i.e., $\theta_1 \approx_{pc} \theta_2$ iff $Dom(\theta_1) \equiv Dom(\theta_2)$ and $\forall x. \theta_1(x) \approx_{pc} \theta_2(x)$, and the $\lambda^{dFG}$ function body $x.[e]$ (resp. thunk body $\_.[f]$) is obtained from the $\lambda^{dCG}$ function body $e$ (resp. thunk $t$) via the program translation defined above. Lastly, rule [LABELED] relates a $\lambda^{dCG}$ labeled value Labeled $\ell v_1$ to a pair $(\ell', v_2)$, consisting of the label $\ell$ protected by itself in the first component and value $v_2$ related with the content $v_1$ at security level $\ell$ ($v_1 \approx_{\ell} v_2$) in the second component. This rule follows the principle of LIO that for explicitly labeled values, the label annotation represents an upper bound on the sensitivity of the content. Stores are related pointwise, i.e., $\Sigma_1 \approx \Sigma_2$ iff $\Sigma_1(\ell) \approx \Sigma_2(\ell)$ for $\ell \in L$, and $\ell$-labeled memories relate their contents respectively at security level $\ell$, i.e., $[] \approx []$ and $(r_1 : M_1) \approx (r_2 : M_2)$ iff $r_1 \approx_{\ell} r_2$ and $M_1 \approx M_2$ for $\lambda^{dFG}$ and $\lambda^{dCG}$ memories $M_1$, $M_2$ : Memory $\ell$. Lastly, we lift the relation to initial and final configurations.

Definition 1 (Equivalence of Configurations). For all initial and final configurations:

- $\langle \Sigma_1, [e]() \rangle \approx \langle \Sigma_2, pc, e \rangle$ iff $\Sigma_1 \approx \Sigma_2$,
- $\langle \Sigma_1, [t] \rangle \approx \langle \Sigma_2, pc, t \rangle$ iff $\Sigma_1 \approx \Sigma_2$,
- $\langle \Sigma_1, r^{pc} \rangle \approx \langle \Sigma_2, pc, v \rangle$ iff $\Sigma_1 \approx \Sigma_2$ and $r \approx_{pc} v$.

For initial configurations, the relation requires the $\lambda^{dFG}$ code to be obtained from the $\lambda^{dCG}$ expression (resp. thunk) via the program translation function $[\cdot]$ defined above (similar to rules [FUN] and [THUNK] in Figure 17). Furthermore, in the first case (expressions), the relation additionally forces the translated suspension $[e]$ by applying it to $()$, so that when the $\lambda^{dFG}$ security monitor executes the translated program, it obtains the result that corresponds to the $\lambda^{dCG}$ monadic program $e$. The third definition relates final configurations whenever the stores are related and the security level of the final $\lambda^{dFG}$ result corresponds to the program counter label $pc$ of the final $\lambda^{dCG}$ configuration, and the final $\lambda^{dCG}$ result corresponds to the $\lambda^{dFG}$ result up to extra annotations at security level $pc$, i.e., $r \approx_{pc} v$.

Before showing semantics preservation, we prove some basic properties of the equivalence that will be useful later. The following property allows instantiating the semantics preservation theorem with the $\lambda^{dCG}$ initial configuration. The translation for initial configurations is per-component, i.e., $[\langle \Sigma, pc, t \rangle] = [\langle \Sigma \rangle, [t]]$ and forcing for suspensions, i.e., $[\langle \Sigma, pc, e \rangle] = [\langle \Sigma \rangle, [e]]$, pointwise for stores, i.e., $[\Sigma] = \ell. [\Sigma(\ell)]$, and memories, i.e., $[] = []$ and $[v : M] = [v] : [M]$ for $\ell$-labeled memory $M$.

\textsuperscript{20}This relation is conceptually similar to the logical relation developed by Rajani and Garg [2018] for their translations with static IPC enforcement, but technically different in the treatment of labeled values.

Property 3 (Reflexivity). For all $\lambda^{dCG}$ initial configurations $c$, $\llbracket c \rrbracket \vdash c$.

Proof. The proof is by induction and relies on analogous properties for all syntactic categories: for stores, $\llbracket \Sigma \rrbracket \vdash \Sigma$, for memories, $\llbracket M \rrbracket \vdash M$, for environments $\llbracket \theta \rrbracket^{pc} \vdash^{pc} \theta$, for values $\llbracket v \rrbracket^{pc} \vdash^{pc} v$, for any label $pc$.

The next property guarantees that values and environments remain in the relation when the program counter label rises.

Property 4 (Weakening). For all labels $pc$ and $pc'$ such that $pc \sqsubseteq pc'$, and for all $\lambda^{dFG}$ raw values $r_1$, values $v_1$ and environments $\theta_1$, and $\lambda^{dCG}$ values $v_2$ and environments $\theta_2$:

- If $r_1 \downarrow^{pc} v_2$ then $r_1 \downarrow^{pc'} v_2$
- If $v_1 \downarrow^{pc} v_2$ then $v_1 \downarrow^{pc'} v_2$
- If $\theta_1 \downarrow^{pc} \theta_2$ then $\theta_1 \downarrow^{pc'} \theta_2$

Proof. By mutual induction on the cross-language equivalence relation.

5.2 Correctness

With the help of the cross-language relation defined above, we can now state and prove that the $\lambda^{dCG}$-to-$\lambda^{dFG}$ translation is correct, i.e., it satisfies a semantics-preservation theorem analogous to that proved for the translation in the opposite direction. At a high level, the theorem ensures that the translation preserves the meaning of a secure terminating $\lambda^{dCG}$ program when executed under $\lambda^{dFG}$ semantics, with the same program counter label and translated input values. Since the translated $\lambda^{dFG}$ program computes strictly more information than the original $\lambda^{dCG}$ program, the theorem existentially quantify over the $\lambda^{dFG}$ result, but insists that it is semantically equivalent to the original $\lambda^{dCG}$ result at a security level equal to the final value of the program counter label, using the cross-language relation just defined.

We start by proving that the program translation preserves typing.

Lemma 5.1 (Type Preservation). If $\Gamma \vdash e : \tau$ then $\llbracket \Gamma \rrbracket \vdash \llbracket e \rrbracket : \llbracket \tau \rrbracket$.

Proof. By straightforward induction on the typing judgment.

Next, we prove semantics preservation of $\lambda^{dCG}$ pure reductions. Since these reductions do not perform any security-relevant operation (they do not read or write state), they can be executed with any program counter label in $\lambda^{dFG}$ and do not change the state in $\lambda^{dFG}$.

Lemma 5.2 ($\llbracket \cdot \rrbracket : \lambda^{dCG} \rightarrow \lambda^{dFG}$ preserves Pure Semantics). If $e \uparrow^\theta v$ then for any program counter label $pc$, $\lambda^{dFG}$ store $\Sigma$, environment $\theta'$ such that $\theta' \downarrow^{pc} \theta$, there exists a raw value $r$, such that $\langle \Sigma, \llbracket e \rrbracket \rangle \uparrow^{{pc}} (\Sigma, r^{pc})$ and $r \downarrow^{pc} v$.

Proof. By induction on the given evaluation derivation and using basic properties of the lattice.

Notice that the lemma holds for any input target environment $\theta'$ in relation with the source environment $\theta$ at security level $pc$ rather than just for the translated environment $\llbracket \theta \rrbracket^{pc}$. Intuitively, we needed to generalize the lemma so that the induction principle is strong enough to discharge cases where (i) we need to prove reductions that use an existentially quantified environment, e.g., [App] and (ii) when some intermediate result at a security level other than $pc$ gets added to the environment, so the environment is no longer homogenously labeled with $pc$. While the second condition does not arise in pure reductions, it does occur in the reduction of monadic expressions considered in the following theorem.

Theorem 5 ($\llbracket \cdot \rrbracket : \lambda^{dCG} \rightarrow \lambda^{dFG}$ preserves Thunk and Forcing Semantics).
• Let \( c_2 = (\Sigma, pc, t) \) be an initial \( \lambda^{dCG} \) configuration. If \( c_2 \Downarrow^{\theta_1} c_2' \), then for all \( \lambda^{dFG} \) environments \( \theta_1 \) and initial configurations \( c_1 \) such that \( \theta_1 \downarrow_{pc} \theta_2 \) and \( c_1 \downarrow_{r} c_2 \), there exists a final configuration \( c_1' \), such that \( c_1 \Downarrow^{\theta_1} c_1' \) and \( c_1' \downarrow_{pc} c_2' \).

• Let \( c_2 = (\Sigma, pc, e) \) be an initial \( \lambda^{dCG} \) configuration. If \( c_2 \Downarrow^{\theta_1} c_2' \), then for all \( \lambda^{dFG} \) environments \( \theta_1 \) and initial configurations \( c_1 \) such that \( \theta_1 \downarrow_{pc} \theta_2 \) and \( c_1 \downarrow_{r} c_2 \), there exists a final configuration \( c_1' \), such that \( c_1 \Downarrow^{\theta_1} c_1' \) and \( c_1' \downarrow_{pc} c_2' \).

**Proof (Sketch).** By mutual induction on the given derivations, using Lemma 5.2 for pure reductions and Properties 2 and 4 in cases [Bind, ToLabeled, Unlabel, Read], basic properties of the lattice and of the translation function (for operations on the store).

We finally instantiate the semantics-preservation theorem with the translation of the input values and the initial stores at security level \( pc \).

**COROLLARY 1 (Correctness).** Let \( c_2 = (\Sigma, pc, e) \), if \( c_2 \Downarrow^{\theta} c_2' \), then there exists a final \( \lambda^{dFG} \) configuration \( c_1' \) such that \( \llbracket c_2 \rrbracket \Downarrow^{\theta_{pc}} c_1' \) and \( c_1' \downarrow_{pc} c_2' \).

**Proof.** By Property 3 and Theorem 5.

Notice that the flow-sensitive program counter of the source \( \lambda^{dCG} \) program gets encoded in the security level of the result of the \( \lambda^{dFG} \) translated program. For example, if \( (\Sigma_2, pc, e) \Downarrow^{\theta} (\Sigma, pc', v) \) then, by Corollary 1 and unrolling Definition 1, there exists a raw value \( r \) at security level \( pc' \) and a store \( \Sigma_1 \), such that \( (\llbracket \Sigma_2 \rrbracket, r) \Downarrow^{\theta_{pc'}} (\Sigma_1, r) \), \( r \downarrow_{pc} v \) and \( \Sigma_1 \downarrow_{pc} \Sigma_2 \).

**Recovery of non-interference.** Similarly to our presentation of Theorem 4 for the translation in the opposite direction, we conclude this section with a sanity check—recovering a proof of termination-insensitive non-interference (TIN) for \( \lambda^{dCG} \) through the program translation defined above, semantics preservation (Corollary 1), \( \lambda^{dFG} \) non-interference (Theorem 1), together with a property that the translation preserves \( L \)-equivalence as well (Lemmas 5.3, 5.4 and 5.5). By reproving non-interference of the source language from the target language, we show that our program translation is authentic.

The following lemma ensures that the translation of initial configurations preserves \( L \)-equivalence.

**LEMMA 5.3.** If \( c_1 \approx_L c_2 \), then \( \llbracket c_1 \rrbracket \approx_L \llbracket c_2 \rrbracket \).

**Proof.** By induction on the \( L \)-equivalence judgment and proving similar lemmas for values, i.e., if \( v_1 \approx_L v_2 \) then \( \llbracket v_1 \rrbracket_{pc} \approx_L \llbracket v_2 \rrbracket_{pc} \), for environments, i.e., if \( \theta_1 \approx_L \theta_2 \) then \( \llbracket \theta_1 \rrbracket_{pc} \approx_L \llbracket \theta_2 \rrbracket_{pc} \), for any label \( pc \) for memories, i.e., if \( M_1 \approx_L M_2 \) then \( \llbracket M_1 \rrbracket \approx_L \llbracket M_2 \rrbracket \), and for stores, i.e., if \( \Sigma_1 \approx_L \Sigma_2 \) then \( \llbracket \Sigma_1 \rrbracket \approx_L \llbracket \Sigma_2 \rrbracket \).

The following lemmas recovers \( \lambda^{dCG} \) \( L \)-equivalence from \( \lambda^{dFG} \) \( L \)-equivalence using the cross-language equivalence relation for all the syntactic categories.

**LEMMA 5.4.** For all public program counter labels \( pc \subseteq L \), for all \( \lambda^{dFG} \) values \( v_1, v_2 \), raw values \( r_1, r_2 \), environments \( \theta_1, \theta_2 \), memories \( M_1, M_2 \), stores \( \Sigma_1, \Sigma_2 \), and corresponding \( \lambda^{dCG} \) values \( v'_1, v'_2 \) and environments \( \theta'_1, \theta'_2 \), memories \( M'_1, M'_2 \), stores \( \Sigma'_1, \Sigma'_2 \): 

- If \( v_1 \approx_L v_2 \) and \( v_2 \downarrow_{pc} v'_2 \), then \( v'_1 \approx_L v'_2 \).
- If \( r_1 \approx_L r_2 \) and \( r_2 \downarrow_{pc} v'_2 \), then \( v'_1 \approx_L v'_2 \).
- If \( \theta_1 \approx_L \theta_2 \) and \( \theta_2 \downarrow_{pc} \theta'_2 \), then \( \theta'_1 \approx_L \theta'_2 \).
- If \( M_1 \approx_L M_2 \) and \( M_2 \downarrow_{pc} M'_2 \), then \( M'_1 \approx_L M'_2 \).
- If \( \Sigma_1 \approx_L \Sigma_2 \) and \( \Sigma_2 \downarrow_{pc} \Sigma'_2 \), then \( \Sigma'_1 \approx_L \Sigma'_2 \).
Proof. The lemmas are proved mutually, by induction on the $L$-equivalence relation and the cross-language equivalence relations and using injectivity of the translation function $\llbracket \cdot \rrbracket$ for closure values.  

The next lemma lifts the previous lemma final configurations.

**Lemma 5.5.** Let $c_1$ and $c_2$ be $\lambda^{dFG}$ final configurations, let $c'_1$ and $c'_2$ be $\lambda^{dCG}$ final configurations. If $c_1 \approx_L c_2$, $c_1 \approx c'_1$ and $c_2 \approx c'_2$, then $c'_1 \approx_L c'_2$.

**Proof.** Let $c_1 = (\Sigma_1, v_1)$, $c_2 = (\Sigma_2, v_2)$, $c'_1 = (\Sigma'_1, pc_1, v'_1)$, $c'_2 = (\Sigma'_2, pc_2, v'_2)$. From $L$-equivalence of $\lambda^{dFG}$ final configurations, it follows $L$-equivalence for the stores and the values, i.e., $\Sigma_1 \approx_L \Sigma_2$ and $v_1 \approx_L v_2$ from $c_1 \approx_L c_2$ (Section 2.2). Similarly, from cross-language equivalence of final $\lambda^{dFG}$ and $\lambda^{dCG}$ configurations, it follows cross-language equivalence of their components, i.e., respectively $\Sigma_1 \approx \Sigma'_1$ and $v_1 \approx_{pc_1} v'_1$ from $c_1 \approx c'_1$, and $\Sigma_2 \approx \Sigma'_2$ and $v_2 \approx_{pc_2} v'_2$ from $c_2 \approx c'_2$ (Definition 1).

First, we show that the $\lambda^{dCG}$ stores are $L$-equivalent, i.e., $\Sigma'_1 \approx_L \Sigma'_2$ by Lemma 5.4 for stores, then two cases follow by case split on $v_1 \approx_L v_2$. Either (i) both label annotations on the values are not observable ($\text{Value}_H$), then the program counter labels are also not observable ($\text{pc}_1 \not\in L$ and $\text{pc}_2 \not\in L$ from $v_1 \approx_{pc_1} v'_1$ and $v_2 \approx_{pc_2} v'_2$) and $c'_1 \approx_L c'_2$ by rule $(\text{PC}_L)$ or (ii) the label annotations are equal and observable by the attacker ($\text{Value}_L$), i.e., $\text{pc}_1 \equiv \text{pc}_2 \equiv L$, then $v'_1 \approx_L v'_2$ by Lemma 5.4 for values and $c'_1 \approx_L c'_2$ by rule $(\text{PC}_L)$.

**Theorem 6 ($\lambda^{dCG}$-TINI via $\llbracket \cdot \rrbracket$).** If $c_1 \llbracket \theta_1 \rrbracket c'_1, c_2 \llbracket \theta_2 \rrbracket c'_2, \theta_1 \approx_L \theta_2$ and $c_1 \approx_L c_2$, then $c'_1 \approx_L c'_2$.

**Proof.** First, we apply the translation $\llbracket \cdot \rrbracket : \lambda^{dCG} \rightarrow \lambda^{dFG}$ to the initial configurations $c_1$ and $c_2$ and the respective environments $\theta_1$ and $\theta_2$. Let $pc$ be the initial program counter label common to configurations $c_1$ and $c_2$ (it is the same because $c_1 \approx_L c_2$). Corollary 1 (Correctness) then ensures that there exist two $\lambda^{dFG}$ configurations $c_1'$ and $c_2'$, such that $\llbracket c_1 \rrbracket \llbracket \theta_1 \rrbracket^{pc} c_1'$ and $\llbracket c_1' \rrbracket \approx_L \llbracket c_2 \rrbracket^{pc}$ and $\llbracket c_2 \rrbracket^{pc} c_2'$ and $\llbracket c_2' \rrbracket \approx_L \llbracket c_1' \rrbracket^{pc}$. We then lift $L$-equivalence of source configurations and environments to $L$-equivalence in the target language via Lemma 5.3, i.e., $\llbracket \theta_1 \rrbracket^{pc} \approx_L \llbracket \theta_2 \rrbracket^{pc}$ and $\llbracket c_1 \rrbracket \approx_L \llbracket c_2 \rrbracket$, and apply Theorem 1 ($\lambda^{dFG}$-TINI) to the reductions i.e., $\llbracket c_1 \rrbracket \llbracket \theta_1 \rrbracket^{pc} c_1'$ and $\llbracket c_2 \rrbracket^{pc} c_2'$, which gives $L$-equivalence of the resulting configurations, i.e., $c_1' \approx_L c_2'$. Then, we apply Lemma 5.5 to $c_1' \approx_L c_2'$, $c_1' \approx L c_2'$, and $c_2' \approx L c_2'$ and recover $L$-equivalence for the source configurations, i.e., $c'_1 \approx_L c'_2$.

6 RELATED WORK

Systematic study of the relative expressiveness of fine- and coarse-grained information flow control (IFC) systems has started only recently. Rajani et al. [2017] initiated this study in the context of static coarse- and fine-grained IFC, enforced via type systems. In more recent work, Rajani and Garg [2018] show that a fine-grained IFC type system, which they call FG, and two variants of a coarse-grained IFC type system, which they call CG, are equally expressive. Their approach is based on type-directed translations, which are type- and semantics-preserving. For proofs, they develop logical relations models of FG and the two variants of CG, as well as cross-language logical relations.

Our work and some of our techniques are directly inspired by their work, but we examine dynamic IFC systems based on runtime monitors. As a result, our technical development is completely different. In particular, in our work we handle label introspection, which has no counterpart in the

---

21Technically, the function $\llbracket \cdot \rrbracket$ presented in Section 5 is not injective. For example, consider the type translation function from Figure 14a: $\text{Labeled unit} = \mathcal{L} \times \text{unit} = \llbracket \mathcal{L} \times \text{unit} \rrbracket$ but $\text{Labeled unit} \neq \mathcal{L} \times \text{unit}$, and $\text{LIO unit} = \text{unit} \rightarrow \text{unit} = \llbracket \text{unit} \rightarrow \text{unit} \rrbracket$ but $\text{LIO unit} \neq \text{unit} \rightarrow \text{unit}$. We make the translation injective by (i) adding a wrapper type $\text{Id}_\tau$ to $\lambda^{dFG}$, together with constructor $\text{Id}(e)$, a deconstructor $\text{unId}(e)$ and raw value $\text{Id}(v)$, and (ii) tagging security-relevant types and terms with the wrapper, i.e., $\llbracket \text{Labeled}_\tau \rrbracket = \text{Id}(\mathcal{L} \times [\tau])$ and $\text{LIO}_\tau = \text{Id}(\text{unit} \rightarrow [\tau])$. Adapting the translations in both directions is tedious but straightforward; we refer the interested reader to our mechanized proofs for details.
earlier work on static IFC systems, and which also requires significant care in translations. Our dynamic setting also necessitated the use of tainting operators in both the fine-grained and the coarse-grained systems.

Our coarse-grained system $\lambda^{dCG}$ is the dynamic analogue of the second variant of Rajani and Garg [2018]'s CG type system. This variant is described only briefly in their paper (in Section 4, paragraph "Original HLIO") but covered extensively in Part-II of the paper’s appendix. Rajani and Garg [2018] argue that translating their fine-grained system FG to this variant of CG is very difficult and requires significant use of parametric label polymorphism. The astute reader may wonder why we do not encounter the same difficulty in translating our fine-grained system $\lambda^{dFG}$ to $\lambda^{dCG}$. The reason for this is that our fine-grained system $\lambda^{dFG}$ is not a direct dynamic analogue of Rajani and Garg [2018]'s FG. In $\lambda^{dFG}$, a value constructed in a context with program counter label $pc$ automatically receives the security label $pc$. In contrast, in Rajani and Garg [2018]'s FG, all introduction rules create values (statically) labeled $\bot$. Hence, leaving aside the static-vs-dynamic difference, FG’s labels are more precise than $\lambda^{dFG}$’s, and this makes Rajani and Garg [2018]'s FG to CG translation more difficult than our $\lambda^{dFG}$ to $\lambda^{dCG}$ translation. In fact, in earlier work, Rajani et al. [2017] introduced a different type system called FG−, a static analogue of $\lambda^{dFG}$ that labels all constructed values with $pc$ (statically), and noted that translating it to the second variant of CG is much easier (in the static setting).

Coarse-grained dynamic IFC systems are prevalent in security research in operating systems [Efstathopoulos et al. 2005; Krohn et al. 2007; Zeldovich et al. 2006]. These ideas have also been successfully applied to other domains, e.g., the web [Bauer et al. 2015; Giffin et al. 2012; Stefan et al. 2014; Yip et al. 2009], mobile applications [Jia et al. 2013; Nadkarni et al. 2016], and IoT [Fernandes et al. 2016]. LIO is a domain-specific language embedded in Haskell that rephrases OS-like IFC enforcement into a language-based setting [Stefan et al. 2012, 2011]. Heule et al. [2015] introduce a general framework for coarse-grained IFC in any programming language in which external effects can be controlled. Laminar [Roy et al. 2009] unifies mechanisms for IFC in programming languages and operating systems, resulting in a mix of dynamic fine- and coarse-grained enforcement.

In general, dynamic fine-grained IFC systems often do not support label introspection. LIO [Stefan et al. 2017, 2011] and Breeze [Hritcu et al. 2013] are notable exceptions. Breeze is conceptually similar to our $\lambda^{dFG}$ except for the $\text{taint}()$ primitive. Different from our $\lambda^{dFG}$, there are dynamic fine-grained IFC systems in which labels of references are flow-sensitive [Austin and Flanagan 2009, 2010; Bichhawat et al. 2014; Hedin et al. 2014]. This design choice, however, allows label changes to be exploited as a covert channel for information leaks [Austin and Flanagan 2009, 2010; Russo and Sabelfeld 2010]. There are many approaches to preventing such leaks—from using static analysis techniques [Sabelfeld and Myers 2006], to disallowing label upgrades depending on sensitive data (i.e., no-sensitive-upgrades [Austin and Flanagan 2009; Zdancewic 2002]), to avoiding branching on data whose labels have been upgraded (i.e., permissive-upgrades [Austin and Flanagan 2010]). Extending our results to a fine-grained dynamic IFC system with flow-sensitive references is an interesting direction for future work.

7 CONCLUSION

We formally established a connection between dynamic fine- and coarse-grained enforcement for IFC, showing that both are equally expressive under reasonable assumptions. Indeed, this work provides a systematic way to bridging the gap between a wide range of dynamic IFC techniques often proposed by the programming languages (fine-grained) and operating systems (coarse-grained) communities. As consequence, this allows future designs of dynamic IFC to choose a coarse-grained
approach, which is easier to implement and use, without giving up on the precision of fine-grained IFC.

ACKNOWLEDGMENTS

We thank the anonymous POPL and POPL AEC reviewers for the insightful comments. This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023) and WebSec (Ref. RIT17-0011) as well as the Swedish research agency Vetenskapsrådet. Vineet Rajani was partly funded through the Collaborative Research Center “Methods and Tools for Understanding and Controlling Privacy” (SFB 1223) of the DFG, project “Programming Principles and Abstractions for Privacy.” This work was also supported in part by a gift from Cisco and the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES


