Simple Contextual Information-Flow Control with Effects

ANONYMOUS AUTHOR(S)

Pure functional languages have the particularity of being able to provide information-flow security (IFC) as a library. These libraries (e.g., LIO, MAC, HLIO, etc.) use monads as their main abstraction for enforcing security. While sound, such a design decision can introduce restrictions at the time of programming that need to be solved by adding extra primitives, e.g., to support a functor or applicative-like functor structure when handling sensitive values. We argue that such approaches are unnecessarily conservative mainly because of the use of *a single security label* to secure both aspects of a monadic computation, namely its effects and result.

In this work, we explore a different point in the design space for IFC in pure languages. We present coIFC, a novel domain-specific language where IFC is tracked differently for the effect-free and effectful components. This language uses modalities for protecting the effect-free part of our language, and a graded monad for effectful components. The monad does not enforce security per se, but only collects the labels of the potential observers of the computation's effects. coIFC reduces the security of effectful IFC to a *single point* in the language, thus giving us clarity and simplicity in the design and implementation.

We provide a Haskell library implementing coIFC in less than 10 lines of code for the effect-free fragment and less than 30 for the effectful part. We demonstrate that our library is capable to encode previous libraries that enforce IFC statically. We also show that providing a functor or applicative-functor structure for sensitive values is a derived operation in our language. All of our security guarantees are mechanized in the Agda proof assistant.

Additional Key Words and Phrases: Modalities, Graded Monads, Information-Flow Control, Security Library, Haskell

ACM Reference Format:
Anonymous Author(s). 2017. Simple Contextual Information-Flow Control with Effects. In . ACM, New York, NY, USA, 28 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Information-flow control [17, 38] (IFC) is a promising technology to protect data confidentiality. Many IFC approaches are designed to prevent sensitive data from influencing what attackers can observe from a program's public behavior—a security policy known as non-interference [14]. In the past years, the use of pure functional languages for tackling IFC challenges and answering research questions has been proliferating, e.g., [31, 33, 34, 49]. From the practical point of view, pure functional languages can provide IFC security via libraries [24, 37, 42]—which is a minor task when compared to building compilers or interpreters from scratch [9, 16, 28, 41]. Some of these security libraries have demonstrated to be capable of building practical secure systems [13, 31].

The foundational work on the Dependency Core Calculus [1] (DCC) positions monads as a suitable abstraction for enforcing IFC on the simply-typed lambda calculus; and there are several implementations of DCC's ideas in Haskell [3, 4, 37, 44]—a language known to embrace monadic domain-specific languages. Intuitively, DCC implements security through a family T of monads indexed by security labels. The type $T_\ell\ a$ denotes data of type a which is only accessible to observers

ICFP2021, July 1997, City, State, Country

ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

who have the privilege of reading data at least as sensitive as the security label ℓ. This idea naturally extends to secure effects which can be represented in a pure way (like state or exceptions), but it becomes less evident how to use it to secure arbitrary effects, e.g., like those provided by the opaque *IO*-monad. Hirsch and Cecchetti [19] describe a translation from a version of DCC with a *program counter* [11]—a label used to rule out data leakage arising from effects in imperative languages—to an effect-free setting based on *productors* [43].

Other monadic approaches to IFC adopt principles from operating systems [6] to provide security in pure functional languages. For instance, Russo [36] introduces a family *MAC* of monads indexed by security labels. The type $MAC\ \ell\ a$ represents an effectful computation where the label ℓ indicates that the result of type a is only accessible to observers who have the privilege of reading data at least as sensitive as ℓ—analogously to DCC. However, the label ℓ is also used to restrict leakeage through effects. Specifically, a computation of type $MAC\ \ell\ a$ can only read sources labeled ℓ_i if they are at most as sensitive as ℓ (*no-read-up principle*). Similarly, a computation of type $MAC\ \ell\ a$ can only write to sinks labeled ℓ_o if they are at least as sensitive as ℓ (*no-write-down principle*). The choice of using a single label to restrict both aspects of a computation, i.e., both its effects and result, is exhibited by a number of IFC libraries, e.g., SecLib [37], LIO [42], HLIO[10], LWeb [31], and Lifty [33]. While sound, such a design decision imposes constraints at the time of programming that need to be resolved by adding more primitives. For instance, the work by Vassena et al. [46] adds a functor and applicative-like structure to sensitive values handled by *MAC*, which took the authors to re-do the security proofs from previous work [47].

This work explores a different point in the design space of enforcing IFC in pure languages. We design a security library that is simple, sound, and implementable in a way that *each security label talks only about a single aspect of the computation*, i.e., either a value like an integer or a boolean, or an effect like writing to a file. For that we have taken inspiration from work on IFC for the simply-typed lambda calculus using modalities [26, 40] and other contextual approaches like coeffects [12]. It is known that approaches based on modalities and coeffects are challenging to implement as a library since they are likely to require access to *contextual information* that is usually not available to the library implementer, e.g., which variables are currently in scope. This work shows how to overcome this limitation in the IFC setting.

This article introduces coIFC, a language where IFC is tracked differently for the effect-free and effectful components. By construction, coIFC separates effect-free and effectful terms and each label talks only about a single aspect of the computation. For the effect-free part we utilize a family *Labeled* of modal types. The type $Labeled_\ell\ \tau$ makes sure that the value of type τ is only accessible to observers who have the privilege of reading data at least as sensitive as ℓ. We stress that $Labeled_\ell\ \tau$ is a modal type rather than a monadic one and that implies having a different *interface* to manipulate sensitive values. We later show that the modal operators for protecting or inspecting sensitive values are flexible enough to encode DCC; and we provide a Haskell implementation that corroborates it.

For representing computations coIFC uses a graded monad $Eff_{\ell s}\ \tau$ which tracks in its type a set of security labels ℓs. The monad does *not* enforce security however, it only collects the labels of the potential observers of the computation's effects. For instance, when the effect is printing the set contains the labels of the channels the computation writes to; when the effect is writing to memory it accounts for the security labels of the memory cells the computation writes to. In pure languages—like Haskell—there is a strict separation between the time a computation is built and when it is executed. Enforcing security amounts to only allow computations to run whose effects do not violate the security policy.

In this work, we reduce the problem of ensuring that an effectful sensitive computation does not leak through effects to simply checking that the computation's label is compatible with the effects' label. To illustrate this point, the type $\mathit{Labeled}_{\ell}\ (\mathit{Eff}_{\ell s}\ \mathit{Unit})$ denotes a program which builds an effectful computation depending on sensitive information at level ℓ. A program with this type, if naively allowed to execute, could leak information marked with sensitivity ℓ into public sinks if ℓs contains the label of a less sensitive observer than ℓ. To enforce security, we could forbid running any program of type $\mathit{Labeled}_{\ell}\ (\mathit{Eff}_{\ell s}\ \tau)$, but this would be overly restrictive. Instead, if every label in ℓs is at least as sensitive as ℓ then we *definitely* know that the effect will not leak; and allowing the program to run is safe. The relation between the sensitivity of the data as indicated by the modality $\mathit{Labeled}_{\ell}\ \tau$ and of the effects' observers is the key and the design of coIFC takes full advantage of it. Thus, coIFC reduces the security of effectful IFC to a single point in the language thereby giving us clarity and simplicity in the design and implementation.

Along with our informal argumentation for why coIFC is secure, we have mechanized proofs in the Agda proof assistant about the strong security guarantees that the programs in the language satisfy, namely termination-insensitive non-interference (TINI). Our proofs are based on the technique of logical relations and consist of around 1500 lines of Agda code.

Finally, we present an implementation of coIFC as a Haskell library. The conciseness of our implementation illustrates the elegance and simplicity of our approach: less than 10 lines of code for the effect-free fragment and less than 30 for the effectful part. Our library is at least as expressive as previous work on libraries for IFC in Haskell. We show implementations of SecLib [37], DCC [1] (in its alternative presentation SDCC [3]) and MAC [36]) in terms of coIFC's interface. Furthermore, in order to implement our IFC modal operators as a library, we use a novel encoding of contextual information as capabilities via higher-rank polymorphism [22].

The summary of the technical contributions of this paper are:

- ▶ A language design where effect-free parts are protected by modal types (Section 2).
- ▶ A language design where effectful components are modelled by a graded monad. We show how to handle both write (Section 3) and read effects (Section 4). Further, we present a single primitive capable of enforcing security in effectful computations (Section 3).
- ▶ Security gurantees and proofs of TINI based on logical relations (Section 5).
- ▶ A Haskell implementation of coIFC using a novel encoding of contextual information as capabilities together with evidence that coIFC can encode existing monadic security libraries as well as derive operations for a flexible treatment of sensitive values (Section 6).
- ▶ Mechanized proofs of all our security guarantees (approx. 1500 lines of Agda code submitted as accompanying material).

2 EFFECT-FREE LANGUAGE

In this section we present λ-coIFC, an effect-free language with a modal type-system for information-flow control. We split the presentation in two parts; a base programming language which is a variant of the STLC with call-by-name semantics; and a security type-system that enforces IFC. This incremental presentation arises from the consideration that programs already make sense (semantically speaking) on their own before enforcing security. This statement is specially true for static IFC enforcements, since all the relevant security type annotations and security checks *should not* have any runtime impact on programs.

The effect-free fragment of the language is a non-strict variant of the sealing calculus [40]. For simplicity, we consider *Bool* and *Unit* as the only ground types. The base language, STLC, is given by the set of types, typing contexts, well-typed terms and a small-step reduction relation which explains the operational behavior of programs—see Figure 1. The typing judgements and semantics for STLC are well understood and therefore we omit their explanation here. Instead, we focus on the IFC security type-system.

$$
\begin{array}{rl}
\text{Ground Types} & \sigma \ ::= \ \mathit{Bool} \mid \mathit{Unit} \\
\text{Types} & \tau, \tau_1, \tau_2 \ ::= \ \sigma \mid \tau_1 \rightarrow \tau_2 \\
\text{Typing Contexts} & \Gamma \ ::= \ \cdot \mid \Gamma, x : \tau
\end{array}
$$

$\boxed{\Gamma \vdash M : \tau}$

$$
\textsc{Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}
\qquad
\textsc{Lam}\ \frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \rightarrow \tau_2}
\qquad
\textsc{App}\ \frac{\Gamma \vdash M : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash M\ N : \tau_2}
$$

$$
\textsc{Unit}\ \frac{}{\Gamma \vdash () : \mathit{Unit}}
\qquad
\textsc{True}\ \frac{}{\Gamma \vdash \mathit{true} : \mathit{Bool}}
\qquad
\textsc{False}\ \frac{}{\Gamma \vdash \mathit{false} : \mathit{Bool}}
\qquad
\textsc{If}\ \frac{\Gamma \vdash M : \mathit{Bool} \qquad \Gamma \vdash N_1 : \tau \qquad \Gamma \vdash N_2 : \tau}{\Gamma \vdash \mathit{ifte}(M, N_1, N_2) : \tau}
$$

$\boxed{\Gamma \vdash^{v} V : \tau}$

$$
\textsc{Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash^{v} x : \tau}
\qquad
\textsc{Lam}\ \frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash^{v} \lambda x.M : \tau_1 \rightarrow \tau_2}
\qquad
\textsc{True}\ \frac{}{\Gamma \vdash^{v} \mathit{true} : \mathit{Bool}}
\qquad
\textsc{False}\ \frac{}{\Gamma \vdash^{v} \mathit{false} : \mathit{Bool}}
\qquad
\textsc{Unit}\ \frac{}{\Gamma \vdash () : \mathit{Unit}}
$$

$\boxed{M \longrightarrow N}$

$$
\textsc{App}\ \frac{M \longrightarrow M'}{M\ N \longrightarrow M'\ N}
\qquad
\textsc{Lam}\ \frac{}{(\lambda x.M)\ N \longrightarrow M[N/x]}
\qquad
\textsc{If}\ \frac{M \longrightarrow M'}{\mathit{ifte}(M, N_1, N_2) \longrightarrow \mathit{ifte}(M', N_1, N_2)}
$$

$$
\textsc{If-True}\ \frac{}{\mathit{ifte}(\mathit{true}, N_1, N_2) \longrightarrow N_1}
\qquad
\textsc{If-False}\ \frac{}{\mathit{ifte}(\mathit{false}, N_1, N_2) \longrightarrow N_2}
$$

Fig. 1. Types, terms and small-step operational semantics for STLC (excerpts).

In the rest of this document, we assume L to be a set of security labels equipped with a preorder structure $\mathcal{L} = (L, \sqsubseteq)$; we also denote concrete labels using letters ℓ, ℓ', etc.

λ-coIFC enforces IFC through a type discipline. To introduce the type-system, we start by defining the grammar for security types τ^s and a relation $[\![\tau^s]\!] = \tau$ to be read as τ^s refines type τ with a security annotation—see Figure 2. The security types reflect those in the base language except for a new type former, *Labeled*, which serves to annotate a piece of a program with a security label. *Labeled* takes two arguments, a security label ℓ from L and a security type τ^s.

Further, in Figure 2 we define the typing derivations for the security IFC type-system where the typing rules are defined over well-typed terms in the base language. Judgements in the security type-system are of the form $\pi\ ;\ \Gamma^s \vdash^s M^s : \tau^s$, where $[\![\tau^s]\!] = \tau$ and $[\![\Gamma^s]\!] = \Gamma$—we ignore π for the moment. It is necessary that M is a well-typed term in the base calculus, i.e., $\Gamma \vdash M : \tau$ holds.

For easier reading of examples, we decorate rules [Label] and [Unlabel] with terms, and interchangeably use M to refer both to the underlying term without decorations, i.e., $\Gamma \vdash M : \tau$, and the typing derivation, i.e., $\pi\ ;\ \Gamma^s \vdash^s M^s : \tau^s$. Further, in the latter we usually omit the ^{s} superscript in the typing contexts and types when it is evident from the context.

The component π of the security typing judgement is a finite set of security labels, drawn from L, i.e, $\pi \subseteq_{fin} L$, and it represents the security labels of all the data that the term can depend upon. The set of labels indexing the context are analogous protections contexts from previous work [40, 44]. To illustrate the intuition behind π, we assume the classical two-point lattice for IFC as a preorder, i.e., $\mathcal{LH} = (\{L, H\}, \sqsubseteq_{\mathcal{LH}})$ where $H \not\sqsubseteq_{\mathcal{LH}} L$ is the only disallowed flow. In this scenario, $\pi \overset{\text{def}}{=} \{H\}$ means that the program can unlabel, i.e., can depend on, terms tagged with security type $\mathit{Labeled}_L\ \tau$ or $\mathit{Labeled}_H\ \tau$. Instead, if $\pi \overset{\text{def}}{=} \{L\}$, then only terms of type $\mathit{Labeled}_L\ \tau$ can be unlabeled, ensuring the flows of information are secure, i.e., they follow the policy ascribed by $\mathcal{LH}$. Labels in π act as a kind of *type-level key* whose possession permits access to sensitive information.

The IFC typing rules for the STLC fragment are rather standard: they simply propagate the set of labels π to their premises, e.g., rules [App], [Lam] and [If]. Observe, that using rule [If] to branch on a secret boolean requires the secret to be explicitly unlabeled to be of the correct type, i.e., Bool^s. Rules [Unlabel] and [Label] are the most interesting ones since they are responsible for enforcing that information flows to the appropriate places. These are the only rules which interact in a non-trivial way with the set of security labels π.

The rule [Unlabel] allows unlabeling a term with security type $\mathit{Labeled}_\ell\ \tau$ if π contains a security label which is compatible with the label ℓ in the term's type. Compatible means the policy allows the flow of information. Formally, this is encoded by the relation $\pi \geq \ell \overset{\text{def}}{=} \exists \ell_H.\ \ell_H \in \pi \wedge \ell \sqsubseteq \ell_H$, which we read as π covers ℓ. Intuitively, a key ℓ_H can be used to open a term of type $\mathit{Labeled}_\ell\ \tau$ if information with sensitivity ℓ can flow into entities with sensitivity ℓ_H, which the security policy allows just in case $\ell \sqsubseteq \ell_H$. If we were to assume a lattice structure on L, then, the only possible readers of a term typed with a set of labels π would be any ℓ that is at least as sensitive as *the least upper bound of all the labels in* π, i.e., $\sqcup(\pi) \sqsubseteq \ell$[1].

The rule [Label] serves a double purpose: it marks terms as being of $\mathit{Labeled}_\ell\ \tau$ type; and it extends the set of security labels in the premise with label ℓ. This means that the derivation of the premise can utilize the rule [Unlabel] on any label that can flow to ℓ. From the IFC perspective, [Label] is responsible to make sure that M is built from values of at most sensitivity ℓ or any other label $\ell' \in \pi$. The set π in the judgement $\pi\ ;\ \Gamma \vdash^s M : \tau$ represents an over-approximation on the number of enclosing *label*-terms to M. In case that the set is initially non-empty, then the term can depend on information that can flow to any of those levels, even containing sub-terms whose evaluation cannot be observed. Yet those sub-terms cannot influence the observable behaviour of the program.

As an example, consider a two-point preorder with labels ℓ_1 and ℓ_2 such that the only permitted flows are the reflexive ones, i.e., $\ell_1 \sqsubseteq \ell_1$ and $\ell_2 \sqsubseteq \ell_2$. A program $\cdot \vdash M : \mathit{Bool} \longrightarrow \mathit{Bool}$ typed in the security type system with $\{\ell_1\}\ ;\ \cdot \vdash^s M : \mathit{Labeled}_{\ell_1}\ \mathit{Bool}^s \longrightarrow^s \mathit{Bool}^s$ may use the rule [Unlabel] to make use of its argument to influence the result, e.g., it could be the identity function. On the other hand, if instead the term is typed as $\{\ell_1\}\ ;\ \cdot \vdash^s M : \mathit{Labeled}_{\ell_2}\ \mathit{Bool}^s \longrightarrow^s \mathit{Bool}^s$ then we are guaranteed that the resulting Bool cannot be influenced by the argument, i.e., it is equivalent to a constant program.

3 EFFECTFUL LANGUAGE WITH PRINTING EFFECTS

To secure effects in a language, it is enough to classify them as read and/or write effect and enforce the corresponding security checks—an insight that comes from operating systems research [6, 52] and has been recently adopted by security IFC libraries (e.g., [10, 36, 42]). In this light, we follow the same approach and start describing how to incorporate *write-effects* to λ-coIFC.

[1]The join operator $\sqcup$ is lifted to sets in the usual manner.

Security types

$$\begin{array}{lrl} \text{Ground Types} & \sigma^s & ::= \mathit{Bool}^s \mid \mathit{Unit}^s \\ \text{Types} & \tau^s, \tau_1^s, \tau_2^s & ::= \sigma^s \mid \tau_1^s \to^s \tau_2^s \\ & & \mid \mathit{Labeled}_\ell\, \tau^s \\ \text{Typing Contexts} & \Gamma^s & ::= \cdot \mid \Gamma^s, x : \tau^s \end{array}$$

Annotation relation

$$\begin{aligned} [\![\mathit{Unit}^s]\!] &= \mathit{Unit} \\ [\![\mathit{Bool}^s]\!] &= \mathit{Bool} \\ [\![\tau_1^s \to^s \tau_2^s]\!] &= [\![\tau_1^s]\!] \to [\![\tau_2^s]\!] \\ [\![\mathit{Labeled}_\ell\, \tau^s]\!] &= [\![\tau^s]\!] \\ [\![\cdot]\!] &= \cdot \\ [\![\Gamma^s, x : \tau^s]\!] &= [\![\Gamma^s]\!], x : [\![\tau^s]\!] \end{aligned}$$

Typing rules

$$\boxed{\pi \,;\, \Gamma^s \vdash^s M^s : \tau^s \text{ given } [\![\Gamma^s]\!] \vdash M : [\![\tau^s]\!]}$$

$$\textsc{Var}\ \frac{x : \tau \in \Gamma}{\pi \,;\, \Gamma \vdash^s x : \tau} \qquad \textsc{Lam}\ \frac{\pi \,;\, \Gamma, x : \tau_1 \vdash^s M : \tau_2}{\pi \,;\, \Gamma \vdash^s \lambda x.M : \tau_1 \to^s \tau_2} \qquad \textsc{App}\ \frac{\pi \,;\, \Gamma \vdash^s M : \tau_1 \to^s \tau_2 \qquad \pi \,;\, \Gamma \vdash^s N : \tau_1}{\pi \,;\, \Gamma \vdash^s M\, N : \tau_2}$$

$$\textsc{Unit}\ \frac{}{\pi \,;\, \Gamma \vdash^s () : \mathit{Unit}^s} \qquad \textsc{True}\ \frac{}{\pi \,;\, \Gamma \vdash^s \mathit{true} : \mathit{Bool}^s} \qquad \textsc{False}\ \frac{}{\pi \,;\, \Gamma \vdash^s \mathit{false} : \mathit{Bool}^s}$$

$$\textsc{If}\ \frac{\pi \,;\, \Gamma \vdash^s M : \mathit{Bool}^s \qquad \pi \,;\, \Gamma \vdash^s N_1 : \tau \qquad \pi \,;\, \Gamma \vdash^s N_2 : \tau}{\pi \,;\, \Gamma \vdash^s \mathit{ifte}(M, N_1, N_2) : \tau}$$

$$\textsc{Label}\ \frac{\pi \cup \{\ell\} \,;\, \Gamma \vdash^s M : \tau}{\pi \,;\, \Gamma \vdash^s \mathit{label}_\ell(M) : \mathit{Labeled}_\ell\, \tau} \qquad \textsc{Unlabel}\ \frac{\pi \,;\, \Gamma \vdash^s M : \mathit{Labeled}_\ell\, \tau \qquad \pi \geq \ell}{\pi \,;\, \Gamma \vdash^s \mathit{unlabel}_\ell(M) : \tau}$$

Fig. 2. Security type-system for λ-coIFC.

As in the previous section, we first introduce the syntax, typing rules, and semantics of the underlying language. As expected in a pure setting, programs which might produce effects, hereafter computations, are given a specific type former which enjoys a monadic structure [27]. In our case, $\mathit{Eff}\ \tau$ types computations that when executed on top of possibly producing effects also return a result of type τ. Figure 3 presents the extension to STLC which allows programs to perform printing effects via a primitive *print* (rule [Print]). Note that *print* is indexed by a channel and the channel is static information, i.e., there are a number of fixed channels and these are statically known. This permits linking a channel to the security label of the observers on that channel. The typing rules of the standard monadic operations, *return* and *bind*, are as expected—see rules [Return] and [Bind].

A monadic reduction relation specifies the operational semantics of computations. The relation is of the form $M \rightsquigarrow N, o$ and is to be read as: program $\cdot \vdash M : \mathit{Eff}\ \tau$ evaluates in one step to program $\cdot \vdash N : \mathit{Eff}\ \tau$ and produces output o. The output is a function from channels to lists of boolean values, $o : \mathit{Ch} \to \mathit{List}(\mathit{Bool})$. For a given channel ch, $o(ch)$ is the list of booleans printed to ch during the execution of the program. To combine outputs, we lift the monoid structure on $\mathit{List}(\mathit{Bool})$ to

$$\text{Types} \qquad \tau, \tau_1, \tau_2 \ ::= \ \ldots \mid \mathit{Eff}\ \tau$$

$$\boxed{\Gamma \vdash M : \tau}$$

$$\textsc{Return}\ \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathit{return}(M) : \mathit{Eff}\ \tau} \qquad \textsc{Bind}\ \frac{\Gamma \vdash M : \mathit{Eff}\ \tau_1 \qquad \Gamma \vdash N : \tau_1 \to \mathit{Eff}\ \tau_2}{\Gamma \vdash \mathit{bind}(M, N) : \mathit{Eff}\ \tau_2} \qquad \textsc{Print}\ \frac{\Gamma \vdash M : \mathit{Eff}\ \mathit{Bool}}{\Gamma \vdash \mathit{print}_{ch}(M) : \mathit{Eff}\ \mathit{Unit}}$$

$$\boxed{\Gamma \vdash^{v} V : \tau}$$

$$\textsc{ReturnV}\ \frac{\Gamma \vdash M : \tau}{\Gamma \vdash^{v} \mathit{return}(M) : \mathit{Eff}\ \tau} \qquad \textsc{BindV}\ \frac{\Gamma \vdash N : \mathit{Eff}\ \tau_1 \qquad \Gamma \vdash M : \tau \to \mathit{Eff}\ \tau_2}{\Gamma \vdash^{v} \mathit{bind}(N, M) : \mathit{Eff}\ \tau_2} \qquad \textsc{PrintV}\ \frac{\Gamma \vdash M : \mathit{Eff}\ \mathit{Bool}}{\Gamma \vdash^{v} \mathit{print}_{ch}(M) : \mathit{Eff}\ \mathit{Unit}}$$

$$\boxed{\Gamma \vdash^{cv} CV : \tau}$$

$$\textsc{Return}\ \frac{\Gamma \vdash M : \tau}{\Gamma \vdash^{cv} \mathit{return}(M) : \mathit{Eff}\ \tau}$$

$$\boxed{M \rightsquigarrow N, o}$$

$$\textsc{Pure}\ \frac{M \longrightarrow M'}{M \rightsquigarrow M', \epsilon} \qquad \textsc{Bind-Ctx}\ \frac{M' \rightsquigarrow M', o}{\mathit{bind}(M, N) \rightsquigarrow \mathit{bind}(M', N), o} \qquad \textsc{Bind-Ret}\ \frac{}{\mathit{bind}(\mathit{return}(N), M) \rightsquigarrow M\ N, \epsilon}$$

$$\textsc{Print-False}\ \frac{}{\mathit{print}_{ch}(\mathit{false}) \rightsquigarrow \mathit{return}(()), ch \mapsto [\mathit{false}]} \qquad \textsc{Print-True}\ \frac{}{\mathit{print}_{ch}(\mathit{true}) \rightsquigarrow \mathit{return}(()), ch \mapsto [\mathit{true}]}$$

$$\textsc{Print-Ctx}\ \frac{M \longrightarrow N}{\mathit{print}_{ch}(M) \rightsquigarrow \mathit{print}_{ch}(N), \epsilon}$$

$$\boxed{M \rightsquigarrow^{\star} N, o}$$

$$\textsc{Nil}\ \frac{}{M \rightsquigarrow^{\star} M, \epsilon} \qquad \textsc{Cons}\ \frac{M \rightsquigarrow M', o_1 \qquad M' \rightsquigarrow^{\star} N, o_2}{M \rightsquigarrow^{\star} N, o_1 \cdot o_2}$$

Fig. 3. Types, terms and small-step operational semantics for STLC with printing effects.

functions, and use ϵ to denote the unit element, i.e, $\epsilon = \lambda ch.[\,]$, and $\cdot$ to denote composition, i.e., $(\lambda ch.xs) \cdot (\lambda ch.ys) = \lambda ch.xs \mathbin{+\!\!+} ys$. We borrow notation from Haskell; symbol $[\,]$ stands for the empty list, and function $\mathbin{+\!\!+}$ for list concatenation.

Rules [Bind-Ctx] and [Bind-Ret] reduce the left argument of the bind. The judgement $\Gamma \vdash^{cv} \mathit{return}(M) : \mathit{Eff}\ \tau$ indicates that the term $\mathit{return}(M)$ is a computational value for any term M; the monadic semantics does not evaluate it further. This design choice will have implication on how we state the security property (explained below).

Rules [Print-False] and [Print-True] output *false* and *true* in the output channel *ch*, respectively. The output $ch \mapsto [v]$ is the function that maps the channel *ch* to the list $[v]$ and every other

Security types

$$\text{Types} \quad \tau^s, \tau_1^s, \tau_2^s \ ::= \ \ldots \mid \mathit{Eff}_{\ell s}\ \tau^s$$

Annotation relation

$$[\![\mathit{Eff}_{\ell s}\ \tau^s]\!] = \mathit{Eff}\ [\![\tau^s]\!]$$

Typing rules

$$\boxed{\pi \ ; \ \Gamma^s \vdash^s M^s : \tau^s \text{ given } [\![\Gamma^s]\!] \vdash M : [\![\tau^s]\!]}$$

$$\textsc{Return}\ \frac{\pi \ ; \ \Gamma \vdash^s M : \tau}{\pi \ ; \ \Gamma \vdash^s \mathit{return}(M) : \mathit{Eff}_{\emptyset}\ \tau} \qquad \textsc{Bind}\ \frac{\pi \ ; \ \Gamma \vdash^s N : \mathit{Eff}_{\ell s_1}\ \tau_1 \qquad \pi \ ; \ \Gamma \vdash^s M : \tau_1 \rightarrow^s \mathit{Eff}_{\ell s_2}\ \tau_2}{\pi \ ; \ \Gamma \vdash^s \mathit{bind}(N, M) : \mathit{Eff}_{\ell s_1 \cup \ell s_1}\ \tau_2}$$

$$\textsc{Print}\ \frac{\pi \ ; \ \Gamma \vdash^s M : \mathit{Bool} \qquad \mathit{label}(ch) = \ell}{\pi \ ; \ \Gamma \vdash^s \mathit{print}_{ch}(M) : \mathit{Eff}_{\{\ell\}}\ \mathit{Unit}} \qquad \textsc{SubEff}\ \frac{\pi \ ; \ \Gamma \vdash^s M : \mathit{Eff}_{\ell s_1}\ \tau \qquad \ell s_1 \subseteq \ell s_2}{\pi \ ; \ \Gamma \vdash^s M : \mathit{Eff}_{\ell s_2}\ \tau}$$

$$\textsc{Distr}\ \frac{\pi \ ; \ \Gamma \vdash^s M : \mathit{Labeled}_{\ell}\ (\mathit{Eff}_{\ell s}\ \tau) \qquad \ell \sqsubseteq \ell s}{\pi \ ; \ \Gamma \vdash^s \mathit{distr}(M) : \mathit{Eff}_{\ell s}\ (\mathit{Labeled}_{\ell}\ \tau)}$$

Fig. 4. Security type-system for λ-coIFC^{print}(excerpts).

channel to the empty list. Rule [Print-Ctx] reduces $print_{ch}$'s argument. Observe that the rules for $print_{ch}$ make the primitive strict in its argument, i.e., the reduction first evaluates the argument to a boolean value, and then prints it to *ch*. The reflexive transitive closure of the monadic reduction relation—see rules [Nil] and [Cons]—is responsible to combine the outputs of multiple steps of the computation.

Rule [Pure] defines the interplay between the reduction relation for the effect-free and the effectful fragments of the language. It states that we can lift effect-free into monadic reductions and these produce the empty output ϵ. Terms which exclusively belong to the monadic part of the language, such as *return*, *bind* and *print*, are treated as values by the pure reduction relation from Figure 1—see rules [ReturnV], [BindV] and [PrintV].

Two reduction relations. While it might seem unnecessary to have two separated reduction relations, it is a natural form of expressing the semantics of a pure language with effects [50]. The pure reduction relation is responsible to evaluate pure programs, whilst the monadic reduction relation for computing the effects. Having two separated semantics enables us to precisely localize the parts of the program that might perform effects when executed. This choice of presentation helps to clarify one of the novel insights of this work: in a pure language, computations are first class objects and there exists a separation between building computations, i.e., values of type $\mathit{Eff}\ \tau$, and executing them. *In IFC terms, the construction of programs which can produce effects does not leak information, but its execution might!* In this light, our approach is more permissive than previous work by allowing attackers to build insecure computations as long as they do not attempt to run them.

Once laid down the basis of the extended underlying language, we are in position to turn our attention to security. For IFC purposes, we consider that each output channel has a dedicated security level ℓ which represents the lower bound of the security labels of the observers on the channel. Using $label(ch) = \ell$ we denote that the channel ch is at security label ℓ, i.e., observers at label ℓ' are allowed to see output on ch if $\ell \sqsubseteq \ell'$. The security type system indexes the *print* instruction with the channel and the security level of the channel where the write occurs. For ease of presentation, sometimes we conflate a channel with its security label since to reason about security it is irrelevant that there are several channels associated with the same label.

Unfortunately, by naively extending λ-coIFC with computations (programs of type $\mathit{Eff}\ \tau$) is not secure. To illustrate this point, the reader should consider a program p of type $\mathit{Labeled}_H\ (\mathit{Eff}\ \mathit{Unit})$; and ask themselves, is it secure to execute those effects? It is if $p = \mathit{label}_H(\mathit{print}_H(\mathit{unlabel}_H(s)))$ assuming $\cdot \vdash^s s : \mathit{Labeled}_H\ \mathit{Bool}$ is the secret we would like to protect, i.e., p outputs the contents of s into a secret channel. However, and assuming again the same s, it can also be the case that $p = \mathit{label}_H(\mathit{print}_L(\mathit{unlabel}_H(s)))$. In this case, p, when executed, sends a secret in an public channel, clearly in violation of the security policy! The problem arises because the type of p, $\mathit{Labeled}_H\ (\mathit{Eff}\ \mathit{Unit})$, is not expressive enough; it does not say anything about what effects the computation might perform—in this case printing effects—and most importantly who will be able to observe those effects. Without this information, and as just exemplified, only from its type we cannot tell if the program p is safe to execute. If we wanted to retain any sort of security guarantee in the presence of effects, thus, we would need to disallow executing any such program. However, this would be overly restrictive; we can do better and will do.

To address this, we need to reflect the security levels of the channels a computation prints to in its type so we can decide when it is safe to execute. We extend the security type-system with an *Eff*-type that is more precise; it carries a set of security labels. Programs typeable at security type $\mathit{Eff}_{\ell s}\ \tau$ are guaranteed not to print to any channel whose security label does not belong to the set ℓs. Figure 4 presents λ-coIFC^{print} and the new IFC typing rules that cover possibly effectful programs. We extend the security types with $\mathit{Eff}_{\ell s}\ \tau^s$, which essentially indexes the monad for printing effects with a set of labels ℓs. The set ℓs over-approximates the actual set of labels of the channels that the computation may print to.

Next, we explain the rules of the security type-system. Rule [Return] states that *return* produces no effect, thus its security monadic type has the empty set as index, $\mathit{Eff}_{\emptyset}\ \tau$. Rule [Bind] combines the sets of labels of its two arguments ($\mathit{Eff}_{\ell s_1 \cup \ell s_1}\ \tau_2$). The only instruction which actually performs an effect is *print*, thus, rule [Print] states that its security type is indexed by the label of the channel where the output will be written, ($\mathit{Eff}_{\{\ell\}}\ \mathit{Unit}^s$). Lastly, rule [Subeff] allows for subeffecting, this is casting the type of a term to include more security labels, ($\ell s_1 \subseteq \ell s_2$).

With the security types for the effectful language in place, we turn our attention to terms of type $\mathit{Labeled}_\ell\ (\mathit{Eff}_{\ell s}\ \tau^s)$. Observe that, in the underlying language, a program $\cdot \vdash^s p : \mathit{Labeled}_\ell\ (\mathit{Eff}_{\ell s}\ \tau^s)$ is nothing but a computation, i.e., $\cdot \vdash p : \mathit{Eff}\ \tau$. As the previous examples evidence, in terms of security there are two kinds of computations; those that leak secrets upon execution; and those which do not. Thus, in the general case we should forbid their execution. To increase the expressivity of our security type-system, we introduce a more selective primitive that only permits the execution of computations when security is not compromised. The idea is that programmers can construct computations, i.e., programs of type Eff_τ , at will inside labeled terms; the computations and the effects they produce depend on secret information. However, they are not guaranteed that these programs will ever get executed.

Rule [Distr] introduces a mechanism to make sometimes possible to execute the effects described inside a term of type $\mathit{Labeled}_\ell\ (\mathit{Eff}_{\ell s}\ \tau)$. For that, the rule permits the security-type of its premise to be swapped in its conclusion, i.e., $\mathit{Labeled}_\ell\ (\mathit{Eff}_\ell\ \tau)$ to $\mathit{Eff}_{\ell s}\ (\mathit{Labeled}_\ell\ \tau)$, always provided it is secure to execute the effects. Yet, the question remains: when it is secure to execute the effects?

The answer is, whenever the observers of the computations' effects , i.e., ℓs, are no less sensitive than the label of the decision to perform the effects, i.e., ℓ. The condition $\ell \sqsubseteq \ell s$ in the premise of the rule precisely captures this situation, i.e., $\forall \ell' \in \ell s.\ \ell \sqsubseteq \ell'$. After bringing the monadic type to the top level, we can be sure the effects of type $\mathit{Eff}_{\ell s}\ (\mathit{Labeled}_{\ell}\ \tau^{s})$ can be safely performed by the monadic operational semantics from Figure 3.

To illustrate rule [Distr] in action, we can revisit the previous examples. The program $p = \mathit{label}_H(\mathit{print}_H(\mathit{unlabel}_H(s)))$, again assuming the secret $s \vdash^{s} \mathit{Labeled}_H\ \mathit{Bool}^{s}$, is a well-typed program of type $\mathit{Eff}_{\{H\}}\ (\mathit{Labeled}_H\ \mathit{Bool}^{s})$ in the security type-system—therefore executable—by applying the rule.

$$\vdash^{s} p : \mathit{distr}(\mathit{label}_H(\mathit{print}_H(\mathit{unlabel}_H(s)))) : \mathit{Eff}_{\{H\}}\ (\mathit{Labeled}_H\ \mathit{Unit}^{s})$$

However, program $p = \mathit{label}_H(\mathit{print}_L(\mathit{unlabel}_H(s)))$ cannot be safely executed, since it leaks sensitive information into a public channel. Our type-system rightfully rejects p:

$$\nvdash^{s} p : \mathit{distr}(\mathit{label}_H(\mathit{print}_L(\mathit{unlabel}_H(s)))) : \mathit{Eff}_{\{H\}}\ (\mathit{Labeled}_H\ \mathit{Unit}^{s})$$

Observe that rule [Distr] executes the effects and protects the returned value using the same label as the decision to perform the effects. This is necessary because computations need not to perform effects at all. In a pure language, it is always possible to construct the no-op computation using the monadic primitive *return*.

Rule [Distr] shows the main novelty and elegance of our approach: the separation of effect-free and effectful computations, where each of them has different security labels (i.e., ℓ, and ℓs). Previous security libraries tried to separate effect-free and effectful computations but they often ended up using one security label for both [10, 31, 33, 36, 37, 42]. We argue that in a pure language having such separation is natural, since computations are both executable and first-class objects that can be manipulated. This also helps to clarify the relation between secure effect-free programs and computations as recent works points out [19].

4 EFFECTFUL LANGUAGE WITH MEMORY EFFECTS

We turn our attention to reading effects which are the counterpart to writing effects. In order to do so, we look at an effecful language with memory references.

The language is, again, an extension of STLC with a monadic type for computations and a type for memory references. References allow both writing effects—like channels in λ-coIFC^{print}—but also reading effects, i.e., reading the contents of a memory cell. The idea is that to ensure security, memory references are ascribed a security label, like channels did, and the security label of a memory location is used both to constraint the writes and ensure readings are secure. Following standard procedure, we start by introducing the syntax, typing rules, and semantics for the underlying language. Besides a monadic type for computations, the types also include $\mathit{Ref}\ \sigma$ for typed memory locations that hold terms of type σ. The typing judgement is extended with a store typing Σ, which is a mapping from store locations to ground types. Figure 5 presents the extension to STLC which allows programs to manipulate memory cells via two primitives, *read* and *write* (rules [Read] and [Write]). Moreover, the primitive $\mathit{ref}(l)$ (rule [Ref]) is not part of the surface syntax but rather an internal representation for locations of memory cells. The typing of the monadic operations is exactly as before.

A monadic reduction relation of the form $\theta_1, M \rightsquigarrow \theta_2, N$ specifies the operational semantics of computations. This is to be read as: program $\Sigma\,, \cdot \vdash M : \mathit{Eff}\ \tau$ paired with store θ_1 evaluates in one step to program $\Sigma\,, \cdot \vdash N : \mathit{Eff}\ \tau$ and store θ_2. Observe, the input and output stores are of store typing Σ, i.e., $\theta_1 : \Sigma$ and $\theta_2 : \Sigma$. In order to avoid unnecessary extra complexity, we make two assumptions: stores only holds terms—since we are in a call-by-name language—of ground type,

$$
\begin{array}{rl}
\text{Types} & \tau, \tau_1, \tau_2 \ ::= \ \ldots \mid \mathit{Eff}\,\tau \mid \mathit{Ref}\,\sigma \\
\text{Locations} & l, l_1, \ldots \\
\text{Store Typings} & \Sigma ::= \ \cdot \mid l : \sigma, \Sigma
\end{array}
$$

$$\boxed{\Sigma\,,\Gamma \vdash M : \tau}$$

$$
\textsc{Read}\ \frac{\Sigma\,,\Gamma \vdash M : \mathit{Ref}\,\sigma}{\Sigma\,,\Gamma \vdash \mathit{read}(M) : \mathit{Eff}\,\sigma}
\qquad
\textsc{Write}\ \frac{\Sigma\,,\Gamma \vdash M : \mathit{Ref}\,\sigma \qquad \Sigma\,,\Gamma \vdash N : \sigma}{\Sigma\,,\Gamma \vdash \mathit{write}(M, N) : \mathit{Eff}\ \mathit{Unit}}
\qquad
\textsc{Ref}\ \frac{l : \sigma \in \Sigma}{\Sigma\,,\Gamma \vdash \mathit{ref}(l) : \mathit{Ref}\,\sigma}
$$

$$\boxed{\Gamma \vdash^{v} V : \tau}$$

$$
\textsc{ReadV}\ \frac{\Sigma\,,\Gamma \vdash M : \mathit{Ref}\,\sigma}{\Sigma\,,\Gamma \vdash^{v} \mathit{read}(M) : \mathit{Eff}\,\sigma}
\qquad
\textsc{WriteV}\ \frac{\Sigma\,,\Gamma \vdash M : \mathit{Ref}\,\sigma \qquad \Sigma\,,\Gamma \vdash N : \sigma}{\Sigma\,,\Gamma \vdash^{v} \mathit{write}(M, N) : \mathit{Eff}\ \mathit{Unit}}
\qquad
\textsc{RefV}\ \frac{l : \sigma \in \Sigma}{\Sigma\,,\Gamma \vdash^{v} \mathit{ref}(l) : \mathit{Ref}\,\sigma}
$$

$$\boxed{\theta_1, M \rightsquigarrow \theta_2, N}$$

$$
\textsc{Read}\ \frac{\theta(l) = M}{\theta, \mathit{read}(\mathit{ref}(l)) \rightsquigarrow \theta, \mathit{return}(M)}
\qquad
\textsc{Read-Ctx}\ \frac{M \longrightarrow N}{\theta, \mathit{read}(M) \rightsquigarrow \theta, \mathit{read}(N)}
$$

$$
\textsc{Write}\ \frac{\theta_2 = \theta_1[l \mapsto M]}{\theta_1, \mathit{write}(\mathit{ref}(l), M) \rightsquigarrow \theta_2, \mathit{return}(())}
\qquad
\textsc{Write-Ctx}\ \frac{M_1 \longrightarrow M_2}{\theta, \mathit{write}(M_1, N) \rightsquigarrow \theta, \mathit{write}(M_2, N)}
$$

$$\boxed{\theta_1, M \rightsquigarrow^{\star} \theta_2, N}$$

$$
\textsc{Nil}\ \frac{}{\theta, M \rightsquigarrow^{\star} \theta, M}
\qquad
\textsc{Cons}\ \frac{\theta_1, M \rightsquigarrow \theta_2, M' \qquad \theta_2, M' \rightsquigarrow^{\star} \theta_3, N}{\theta_1, M \rightsquigarrow^{\star} \theta_3, N}
$$

Fig. 5. Extending STLC with read and write effects.

i.e., *Bool* or *Unit*; and the size of the store is fixed during execution. The reason for the former is that higher-order stores allows to express monadic fixpoint operator, thus non-termination. The reason for the latter is that the well-typed formalization that we work on becomes complicated and we prioritize simplicity of presentation—recent work [15] shows it is possible to lift these two simplifications and still obtain NI guarantees.

We follow the same approach as before and extend the security type-system of λ-coIFC with reading and writing primitives. Since terms are well-typed with respect to a store typing, we extend the grammar of security types with security store typings. These equip each location of the store given by the store typing Σ with a security type for its contents and a security label of the location. The latter serves to prevent implicit flows of information through the store [51]. Rules [Return], [Bind], [SubEff] and [Distr] are analogous to those from λ-coIFC^{print}, cf. Figure 4 thus omitted.

Next, we introduce typing rules for the new primitives in Figure 5. Rule [Ref] states that typing a location at security type $\mathit{Ref}_{\ell}\,\sigma$ amounts to checking in the security typing of the store that location l has the correct type and label ℓ. Writing on the store is an effect, thus the typing rule for

Security types

Store Typings $\Sigma^s ::= \cdot \mid l :_\ell \sigma^s, \Sigma^s$

Annotation relation

$$[\![\cdot]\!] = \cdot$$

$$[\![l :_\ell \sigma^s, \Sigma^s]\!] = l : [\![\sigma^s]\!], [\![\Sigma^s]\!]$$

Typing rules

$$\boxed{\pi \; ; \; \Sigma^s \, , \, \Gamma^s \vdash^s M : \tau^s \text{ given } [\![\Sigma^s]\!] \; ; \; [\![\Gamma^s]\!] \vdash M : [\![\tau^s]\!]}$$

Write

$$\frac{\pi \; ; \; \Sigma \, , \Gamma \vdash^s M : \mathit{Ref}_\ell \; \sigma \qquad \Sigma \, , \Gamma \vdash^s N : \sigma}{\pi \; ; \; \Sigma \, , \Gamma \vdash^s \mathit{write}(M, N) : \mathit{Eff}_{\{\ell\}} \; \mathit{Unit}^s}$$

Read

$$\frac{\pi \; ; \; \Sigma \, , \Gamma \vdash^s M : \mathit{Ref}_\ell \; \sigma}{\pi \; ; \; \Sigma \, , \Gamma \vdash^s \mathit{read}(M) : \mathit{Eff}_\emptyset \; (\mathit{Labeled}_\ell \; \sigma)}$$

Ref

$$\frac{l :_\ell \sigma \in \Sigma}{\pi \; ; \; \Sigma \, , \Gamma \vdash \mathit{ref}(l) : \mathit{Ref}_\ell \; \sigma}$$

Fig. 6. Security-type system for λ-coIFC^{mem}(excerpts).

write ([Write]) reflects the writing effect it on its type using the label of the reference (ℓ) as the annotation in its monadic security type, i.e., $\mathit{Eff}_{\{\ell\}} \; \sigma$.

Most peculiar is the typing rule for *read* (rule [Read]) for its type diverges from usual presentation of coarse-grained libraries to IFC (e.g., [33, 36, 42] with exception of *SLIO* in [35]). In coarse-grained libraries, the monadic type keeps track of one label which represents and upper bound of all the labels that could have influenced the computation after unlabeling them. In a nutshell, the monad uses *a single label to rule them all*. Reading from a memory reference, if the flow is permitted, amounts to incorporate the contents of the memory cell directly into the context. The coarse label protects the whole context and in particular the newly incorporated term.

In contrast, λ-coIFC^{mem} takes advantage of purity again by forcing the flow of information from memory to the program to be explicit. Rule [Read]—see Figure 6—does so by wrapping the result of the *read* in a *Labeled* value so subsequent parts of the computation that depend on it have to explicitly remove the label by unlabeling. Further, the typing rule states that the set of labels attached to the monadic type is empty; in our model reads to the store do leave footprints that an attacker can use as a side-channel [21] to recover information about our secrets.

5 SECURITY GUARANTEES

In this section, we formally state the security conditions of the languages presented in previous sections. Further, we explain the proof technique of logical relations which we utilize to show that the type-system is sound, i.e., terms with a typing derivation are secure. Lastly, we detail how each soundess proof follows as a corollary of the fundamental theorem of the appropiate logical relation.

5.1 Effect-free Language

In this section, we formally state our security condition for λ-coIFC: *termination-insensitive non-interference* (TINI). In words, for any given program in λ-coIFC with a hole for a secret, and two executions of it obtained by replacing the secret by any two values, then the public output will be the same. We note that λ-coIFC is strongly normalizing, and assuming termination in TINI

$$
\begin{aligned}
&\mathcal{R}_{\mathcal{V}}[\![Unit^s]\!]_{\pi_{Atk}}(M_1, M_2) = M_1 \equiv M_2\\
&\mathcal{R}_{\mathcal{V}}[\![Bool^s]\!]_{\pi_{Atk}}(M_1, M_2) = M_1 \equiv M_2\\
&\mathcal{R}_{\mathcal{V}}[\![\tau_1 \to^s \tau_2]\!]_{\pi_{Atk}}(M_1, M_2) = \forall(N_1, N_2 : \cdot \vdash \tau_1).\mathcal{R}_{\mathcal{E}}[\![E]\!]_{\tau_1}\pi_{Atk}(N_1, N_2) \Rightarrow \mathcal{R}_{\mathcal{E}}[\![\tau_2]\!]_{\pi_{Atk}}(M_1\ N_1, M_2\ N_2)\\
&\mathcal{R}_{\mathcal{V}}[\![Labeled_{\ell}\ \tau]\!]_{\pi_{Atk}}(M_1, M_2) = \pi_{Atk} \geq \ell \Rightarrow \mathcal{R}_{\mathcal{V}}[\![\tau]\!]_{\pi_{Atk}}(M_1, M_2)\\
\\
&\mathcal{R}_{\mathcal{E}}[\![\tau]\!]_{\ell_{Atk}}(M_1, M_2) = \forall(V_1, V_2 : \cdot \vdash \tau).M_1 \longrightarrow^* V_1 \land M_2 \longrightarrow^* V_2 \Rightarrow \mathcal{R}_{\mathcal{V}}[\![A]\!]_{\pi_{Atk}}(V_1, V_2)\\
\\
&\mathcal{R}_{\mathcal{S}}[\![S]\!]_{\pi_{Atk}}(\cdot, \cdot) = \top\\
&\mathcal{R}_{\mathcal{S}}[\![\Gamma, x : \tau]\!]_{\pi_{Atk}}((\gamma_1, M_1), (\gamma_2, M_2)) = \mathcal{R}_{\mathcal{E}}[\![\tau]\!]_{\pi_{Atk}}(M_1, M_2) \land \mathcal{R}_{\mathcal{S}}[\![\Gamma]\!]_{\pi_{Atk}}(\gamma_1, \gamma_2)\\
\\
&\mathcal{R}_{\mathcal{T}}[\![\Gamma \vdash^s \tau]\!]_{\pi_{Atk}}(M_1, M_2) = \forall(\gamma_1, \gamma_2 : \cdot \Vdash \Gamma).\mathcal{R}_{\mathcal{S}}[\![\Gamma]\!]_{\pi_{Atk}}(\gamma_1, \gamma_2) \Rightarrow \mathcal{R}_{\mathcal{E}}[\![\tau]\!]_{\pi_{Atk}}(\gamma_1(M_1), \gamma_2(M_2))
\end{aligned}
$$

Fig. 7. Logical relation for λ-coIFC.

seems redundant. We remark that it is straightforward to consider fixpoint operators from a IFC-perspective while assuming termination (e.g., as done in [37, 39, 41]) and therefore we omit it here for simplicity and easy of presentation.

In what follows, we keep the formulation of TINI concrete by restricting the secret input and public output to be both of boolean type. This simplifies the security condition since it is enough to check for syntactic equality of the outputs, i.e., boolean values. We assume the presence of an attacker at security level ℓ_{Atk} and secrets with sensitivity ℓ_H. We consider public outputs as those outputs observable by the attacker.

Definition 1 (TINI for λ-coIFC). Assume an attacker with security label ℓ_{Atk}, and a label ℓ_H for secrets such that $\ell_H \not\sqsubseteq \ell_{Atk}$. Given a program M, two terms $\cdot \vdash N_1, N_2 : Bool$, and values V_1, V_2 such that,

- $\{\ell_{Atk}\}\ ;\ s : Labeled_{\ell_H}\ Bool^s \vdash^s M : Labeled_{\ell_{Atk}}\ Bool^s$
- $M[N_1/s] \longrightarrow^* V_1 \land M[N_2/s] \longrightarrow^* V_2$,

then it holds that $V_1 \equiv V_2$.

A more general formulation of TINI is possible, for instance, where the output of the program M is at function type. This naturally follows from the logical relation we use to prove that λ-coIFC satisfies Definition 1. We refer the interested reader to our Agda mechanization for further details. In the rest of this subsection we explain the details of the logical relation.

5.1.1 Soundness of the IFC Type-System for λ-coIFC. In order to prove that the language satisfies TINI, we define a logical relation (LR) by structural induction on the security types. As usual in proofs based on logical relations, the proof of TINI for λ-coIFC follows as a corollary of the fundamental theorem of the LR (described below).

Logical relation for proving TINI. We split the LR in two parts, depending on whether the term is already evaluated, i.e., $\mathcal{R}_{\mathcal{V}}[\![\_]\!]_{\_}$, or not, i.e., $\mathcal{R}_{\mathcal{E}}[\![\_]\!]_{\_}$. At each type the LR stipulates what observations an attacker can perform on programs. For this, the LR is parametrized by a finite set of labels π_{Atk} which represents the attacker's observation power, namely, the attacker can observe any data labeled at ℓ—or below ℓ—such that $\ell \in \pi_{Atk}$.

Before diving into the details of our LRs, we remark that the operational behavior of programs typed in the security type system of Figure 2 is based on the operational reduction of the underlying programs in the base calculus—cf. Figure 1. This implies that the terms decorating the rules [Label] and [Unlabel], i.e., *label* and *unlabel*, do not carry computational meaning; they are operationally irrelevant and do not influence reduction.

The LR for λ-coIFC is described in Figure 7. Starting with the LR for values ($\mathcal{R}_{\mathcal{V}}[\![\_]\!]\_$), we have that for types $Unit^s$ or $Bool^s$ two values are related whenever they are the same syntactic term ($M_1 \equiv M_2$). At the function type $\tau_1 \rightarrow^s \tau_2$, two functions are related whenever they map related inputs[2] ($\mathcal{R}_{\mathcal{E}}[\![E]\!]_{\pi_{Atk}} \tau_1(N_1, N_2)$) to related outputs ($\mathcal{R}_{\mathcal{E}}[\![\tau_2]\!]_{\pi_{Atk}}(M_1\ N_1, M_2\ N_2)$). Finally, if the security type is $Labeled_\ell\ \tau$ for some ℓ and τ, then depending on the attacker observation power we might require the underlying values to be related. If the attacker can observe the labeled term, as in the premise of the implication ($\pi_{Atk} \geq \ell$), then the two values have to be related at type τ. As before, $\pi_{Atk} \geq \ell$ means that ℓ flows to some label in π_{Atk}, i.e., π_{Atk} covers ℓ. In case the attacker cannot observe the label ℓ, both M_1 and M_2 can assume any value.

When it comes to the LR for expressions ($\mathcal{R}_{\mathcal{E}}[\![\_]\!]\_$), we have that, for an attacker with observational power π_{Atk}, two expressions are related at type τ ($\mathcal{R}_{\mathcal{E}}[\![E]\!]_\tau \pi_{Atk}(M_1, M_2)$), whenever both expressions evaluate to two corresponding values and those values are related by the LR for values. Definitions $\mathcal{R}_{\mathcal{V}}[\![\_]\!]\_$ and $\mathcal{R}_{\mathcal{E}}[\![\_]\!]\_$ only work on closed terms. In order to prove the fundamental theorem of the LR we need to lift the latter to closing substitutions, i.e., $\mathcal{R}_{\mathcal{S}}[\![\_]\!]\_$. Two empty substitutions, $(\cdot, \cdot)$, are trivially related, denoted by $\top$, and two non-empty substitutions (γ_1, M_1) and (γ_2, M_2) are related whenever they are pointwise related, i.e., $\mathcal{R}_{\mathcal{E}}[\![\tau]\!]_{\pi_{Atk}}(M_1, M_2) \wedge \mathcal{R}_{\mathcal{S}}[\![\Gamma]\!]_{\pi_{Atk}}(\gamma_1, \gamma_2)$. Finally, the LR for arbitrary open terms, written $\mathcal{R}_{\mathcal{T}}[\![\_]\!]\_$, states that two open terms are related when for any two related closing substitutions, the substituted terms are related by the LR for expression.

The fundamental theorem of the LR states that any open term which can be typed in the security type system of λ-coIFC is related to itself.

Theorem 5.1 (Fundamental theorem of the LR for λ-coIFC). *For any attacker's observational power π_{Atk}, and program $\pi_{Atk}\ ;\ \Gamma \vdash^s M : \tau$ with type τ in typing context Γ, it holds that $\mathcal{T}[\![\Gamma \vdash^s \tau]\!]_{\pi_{Atk}}(M, M)$.*

Proof. By induction on the typing derivation. □

The security property TINI is derived as a corollary of the fundamental theorem.

Corollary 5.2. *Assume an attacker with security label ℓ_{Atk}, and label ℓ_H for secrets, and a program M as described by Definition 1, then M satisfies TINI.*

Proof. Let $\cdot \vdash N_1\ N_2 : Bool$ be the secrets to be substituted in M. Since $\ell_H \not\sqsubseteq \ell_{Atk}$, we have that the secrets are related, i.e., $\mathcal{R}_{\mathcal{E}}[\![Labeled_{\ell_H}\ Bool^s]\!]_{\{\ell_{Atk}\}}(N_1, N_2)$, and thus we have two closing substitutions $\gamma_1 = \{s \mapsto N_1\}$ and $\gamma_2 = \{s \mapsto N_2\}$ which are also related. By the fundamental theorem, the term M is related to itself which means that $\mathcal{R}_{\mathcal{E}}[\![Labeled_{\ell_{Atk}}\ Bool^s]\!]_{\{\ell_{Atk}\}}(\gamma_1(M), \gamma_2(M))$. Unfolding the definitions of $\mathcal{R}_{\mathcal{E}}[\![\_]\!]\_$ and $\mathcal{R}_{\mathcal{V}}[\![\_]\!]\_$, we obtain, $\forall (V_1 V_2 : \cdot \vdash \tau). \gamma_1(M) \longrightarrow^* V_1 \wedge \gamma_2(M) \longrightarrow^* V_2 \Rightarrow V_1 \equiv V_2$ □

The fundamental theorem of the LR permit us to state and prove more general versions of the non-interference theorem. It does not restrict the program nor its result to be of boolean type. For example, the program could return values of function type which independent of secrets are guaranteed to have the same extensional behaviour. In a different direction, the LR let us generalize the secret that M depends on to several secrets intermixed with public dependencies. In such case the public parts should be the same across all executions while the secrets can vary.

[2]The $\forall$ (forall) quantifies over well-typed terms of type τ_1, i.e., $(N_1, N_2 : \cdot \vdash \tau_1)$ means $\cdot \vdash N_1 : \tau_1$ and $\cdot \vdash N_2 : \tau_1$.

5.2 Effectful Language with Printing Effects

Next, we proceed to show the security condition that holds for λ-coIFC^{print} in the presence of effects. To do so, we have to take into account the resulting value of a program and the effects the program performs, and show that an attacker cannot learn information about the program's secrets through either. Therefore, the first step is to define when are two outputs, i.e., the outputs of the same program with different secrets, are indistinguishable to an observer of a certain security label.

Definition 2 (Output indistinguishability). Two outputs o_1 and o_2 are indistinguishable to an observer at level ℓ, denoted by $o_1 \simeq_\ell o_2$, when $\forall ch.\ label(ch) \sqsubseteq \ell \Rightarrow o_1(ch) \ \equiv o_2(ch)$.

In words, the definition above says that any channel ch observable at level ℓ ($label(ch) \sqsubseteq \ell$) contains the same list of boolean values—the prints the program performs on the channel—in both o_1 and o_2. The outputs do not contain observable differences to observers at level ℓ.

Next, we define the security condition for λ-coIFC^{print}, where we demand that a program with substituted secrets produces indistinguishable outputs with regard to an attacker at level ℓ_{Atk}.

Definition 3 (TINI for λ-coIFC^{print}). Assume an attacker with security label ℓ_{Atk}, and a label ℓ_H for secrets such that $\ell_H \not\sqsubseteq \ell_{Atk}$ and a set of security labels ℓs.

Given a program $s : Bool \vdash M : Eff\ Bool$, two terms $\cdot \vdash N_1, N_2 : Bool$, two terms $\cdot \vdash M_1', M_2' : Bool$ and two outputs o_1, o_2, such that

- $\{\ell_{Atk}\}\ ;\ s : Labeled_{\ell_H}\ Bool \vdash_s M : Eff_{\ell s}\ Bool$
- $M[N_1/s] \rightsquigarrow^* return(M_1'),\ o_1 \wedge M[N_2/s] \rightsquigarrow^* return(M_2'),\ o_2,$

then it holds that $o_1 \simeq_{\ell_{Atk}} o_2$.

The security condition disregards the boolean terms produced which result after the programs execution and only takes into account the produced output. In a non-strict language, possibly also in strict languages with explicit monads, the monadic operational semantics does not force evaluation under the monad, i.e., the final value of a computation is $\cdot \vdash return(M) : Eff\ \tau$ where $\cdot \vdash M : \tau$ is an arbitrary program of type τ, not a value.

Definition 3 does not require that the returned terms (M_1' and M_2') reduce to indistinguishable values. An alternative formulation of TINI could demand that if both terms M_1' and M_2' terminate, then they reduce to equal values. Our Agda formalization shows that these two apparently different formulations of Termination-Insensitive Non-Interference are actually equivalent.

5.2.1 *Soundness of the IFC Type-System for λ-coIFC^{print}.*

In order to prove that the language with printing effects is secure, i.e., satisfies *TINI*, we employ again the method of logical relations.

Writing-scope logical predicate. When considering effects, we need to first capture the *writing scope* of the effects that a monadic program might produce, i.e., what are the security labels of the channels that a computation possibly writes to. The security type-system already refines the type of computations by adding a set of security labels. What is needed now is to show that the type-system is sound in this regard; a program $\cdot \vdash p : Eff\ \tau$ with security type $\cdot \vdash^s p : Eff_{\ell s}\ \tau$ is guaranteed not to print on any channel whose security label does not belong to ℓs. Formally, $\forall N, o.\ P \rightsquigarrow^* return(N), o \Rightarrow \forall ch.\ label(ch) \notin \ell s \Rightarrow o(ch) = \epsilon$.

In order to prove soundness, as previous work on *LRs* for security type-systems [35], we use a logical predicate. Figure 8 introduces the logical predicate (LP)—unary logical relation—which defines for each type what effects a program might produce. Similarly as in the previous section, the LP is expressed for values ($\mathcal{W}_\mathcal{V}[\![\_]\!]$), expressions ($\mathcal{W}_\mathcal{E}[\![\_]\!]$), closing substitutions ($\mathcal{W}_\mathcal{S}[\![\_]\!]$), and open terms ($\mathcal{W}_\mathcal{T}[\![\_]\!]$). For base types, $\mathcal{W}_\mathcal{V}[\![\_]\!]$ simply holds. For function types, $\mathcal{W}_\mathcal{V}[\![M : \tau_1 \rightarrow^s \tau_2]\!]$ holds if it maps expressions that satisfy the expression predicate ($\mathcal{W}_\mathcal{E}[\![\tau_1]\!](n)$) to expressions that

$$
\begin{aligned}
&\mathcal{W}_{\mathcal{V}}[\![Unit^s]\!](M) = \top\\
&\mathcal{W}_{\mathcal{V}}[\![Bool^s]\!](M) = \top\\
&\mathcal{W}_{\mathcal{V}}[\![\tau_1 \to^s \tau_2]\!](M) = \forall (N : \cdot \vdash \tau_1).\ \mathcal{W}_{\mathcal{E}}[\![\tau_1]\!](N) \Rightarrow \mathcal{W}_{\mathcal{E}}[\![\tau_2]\!](M\ N)\\
&\mathcal{W}_{\mathcal{V}}[\![Labeled_\ell\ \tau]\!](M) = \mathcal{W}_{\mathcal{V}}[\![\tau]\!](M)\\
&\mathcal{W}_{\mathcal{V}}[\![Eff_{\ell s}\ \tau]\!](M) =\\
&\qquad\qquad \forall N, o.\ M \rightsquigarrow^\star return(N), o \Rightarrow \mathcal{W}_{\mathcal{E}}[\![\tau]\!](N) \wedge (\forall ch.\ label(ch) \notin \ell s \Rightarrow o(ch) \equiv [\,])\\
\\
&\mathcal{W}_{\mathcal{E}}[\![\tau]\!](M) = \forall (V : \cdot \vdash_s \tau).\ M \longrightarrow^* V \Rightarrow \mathcal{W}_{\mathcal{V}}[\![\tau]\!](N)\\
\\
&\mathcal{W}_{\mathcal{S}}[\![\cdot]\!](\cdot) = \top\\
&\mathcal{W}_{\mathcal{S}}[\![\Gamma, x : \tau]\!](\gamma, M) = \mathcal{W}_{\mathcal{E}}[\![\tau]\!](M) \wedge \mathcal{W}_{\mathcal{S}}[\![\Gamma]\!](\gamma)\\
\\
&\mathcal{W}_{\mathcal{T}}[\![\Gamma \vdash_s \tau]\!](M) = \forall (\gamma : \cdot \Vdash \Gamma).\ \mathcal{W}_{\mathcal{S}}[\![\Gamma]\!] \Rightarrow \mathcal{W}_{\mathcal{E}}[\![\tau]\!](\gamma(M))
\end{aligned}
$$

Fig. 8. Logical predicate capturing the writing scope of effects.

also satisfy it ($\mathcal{W}_{\mathcal{E}}[\![\tau_2]\!](M\ N)$). The interesting case is the definition of $\mathcal{W}_{\mathcal{V}}[\![Eff_{\ell s}\ \tau]\!](M)$, which demands that the outputs produced by M only target those channels whose label is present in its security type, i.e., in the set ℓs. The lifting of the LP for substitutions and open terms is standard and therefore we do not describe them any further.

The fundamental theorem of the LP guarantees that the writing scope of a monadic computation is bounded by its security type: the computation may print to any channel whose label is stated in its type and no more.

Theorem 5.3 (Fundamental theorem of the writing-scope predicate, $\mathcal{W}_{\mathcal{T}}[\![\_]\!].$). *For any program $\Gamma \vdash^s M : \tau$, it holds that $\mathcal{W}_{\mathcal{T}}[\![\Gamma \vdash^s \tau]\!](M)$.*

Proof. By induction on typing derivations. □

By capturing the scope of writing effects (outputs), we obtain an approximation about where information could flow via effects. Now we are in a position to reason about TINI for the effectful language.

Logical relation. Figure 9 presents the indistinguishability relation for λ-coIFC^{print} as a LR. Structurally, the indistinguishability relation is similar to the one defined for λ-coIFC—cf. Figure 7—except that now it includes a case when the security type is the monadic type for effects ($\mathcal{R}_{\mathcal{V}}[\![Eff_{\ell s}\ \tau]\!]_{\ell_{Atk}}$). It is worth noting however, that the LR has adjustments in both the function ($\mathcal{R}_{\mathcal{V}}[\![\tau_1 \to^s \tau_2]\!]_{\ell_{Atk}}$) and labeled types ($\mathcal{R}_{\mathcal{V}}[\![Labeled_\ell\ \tau]\!]_{\ell_{Atk}}$), where it mentions the writing scope predicate—recall Figure 8.

The LR at function type has now the additional requirement that both the arguments and the produced result satisfy the writing-scope predicate ($\mathcal{W}_{\mathcal{E}}[\![\tau_2]\!](N_1)$, $\mathcal{W}_{\mathcal{E}}[\![\tau_2]\!](N_2)$, $\mathcal{W}_{\mathcal{E}}[\![\tau_1]\!](M_1\ N_1)$, $\mathcal{W}_{\mathcal{E}}[\![\tau_1]\!](M_2\ N_2)$). Intuitively, this extra assumptions are needed to strengthen the induction hypothesis and prove the fundamental theorem of the LR. Unlike previous work, [34], the binary logical relation does not imply the writing-scope predicate for its components, namely it is not the case that $\mathcal{R}_{\mathcal{E}}[\![\tau]\!]_{\ell_{Atk}}(M_1, M_2) \Rightarrow \mathcal{W}_{\mathcal{E}}[\![\tau]\!](M_1) \wedge \mathcal{W}_{\mathcal{E}}[\![\tau]\!](M_2)$. Intuitively, the reason is that in the call-by-name setting variables are substituted for arbitrary terms and not just for values.

$$
\begin{aligned}
&\mathcal{R}_{\mathcal{V}}[\![\mathit{Unit}^s]\!]_{\ell_{Atk}}(M_1, M_2) = M_1 \equiv M_2\\
&\mathcal{R}_{\mathcal{V}}[\![\mathit{Bool}^s]\!]_{\ell_{Atk}}(M_1, M_2) = M_1 \equiv M_2\\
&\mathcal{R}_{\mathcal{V}}[\![\tau_1 \rightarrow^s \tau_2]\!]_{\ell_{Atk}}(M_1, M_2) = \forall (N_1, N_2 : \cdot \vdash_s \tau_1).\ \mathcal{R}_{\mathcal{E}}[\![\tau_1]\!]_{\ell_{Atk}}(N_1, N_2) \wedge \mathcal{W}_{\mathcal{E}}[\![\tau_2]\!](N_1) \wedge \mathcal{W}_{\mathcal{E}}[\![\tau_2]\!](N_2)\\
&\qquad\qquad \Rightarrow \mathcal{R}_{\mathcal{E}}[\![\tau_2]\!]_{\ell_{Atk}}(M_1\ N_1, M_2\ N_2) \wedge \mathcal{W}_{\mathcal{E}}[\![\tau_1]\!](M_1\ N_1) \wedge \mathcal{W}_{\mathcal{E}}[\![\tau_1]\!](M_2\ N_2)\\
&\mathcal{R}_{\mathcal{V}}[\![\mathit{Labeled}_\ell\ \tau]\!]_{\ell_{Atk}}(M_1, M_2) = (\ell \sqsubseteq \ell_{Atk} \Rightarrow \mathcal{R}_{\mathcal{V}}[\![\tau]\!]_{\ell_{Atk}}(M_1, M_2))\\
&\qquad\qquad \wedge (\ell \not\sqsubseteq \ell_{Atk} \Rightarrow \mathcal{W}_{\mathcal{V}}[\![\tau]\!](M_1) \wedge \mathcal{W}_{\mathcal{V}}[\![\tau]\!](M_2))\\
&\mathcal{R}_{\mathcal{V}}[\![\mathit{Eff}_{\ell s}\ \tau]\!]_{\ell_{Atk}}(M_1, M_2) = \forall (N_1, N_2 : \cdot \vdash \tau), o_1, o_2.\ M_1 \rightsquigarrow^{\star} \mathit{return}(N_1), o_1 \wedge M_2 \rightsquigarrow^{\star} \mathit{return}(N_2), o_2\\
&\qquad\qquad \Rightarrow \mathcal{R}_{\mathcal{E}}[\![\tau]\!]_{\ell_{Atk}}(N_1, N_2) \wedge o_1 \simeq_{\ell_{Atk}} o_2\\
&\qquad\qquad \wedge (\forall ch.\ \mathit{label}(ch) \notin \ell s \Rightarrow o_1(ch) \equiv [])\\
&\qquad\qquad \wedge (\forall ch.\ \mathit{label}(ch) \notin \ell s \Rightarrow o_2(ch) \equiv [])\\
&\mathcal{R}_{\mathcal{E}}[\![\tau]\!]_{\ell_{Atk}}(M_1, M_2) = \forall (V_1, V_2 : \cdot \vdash_s \tau).\ M_1 \longrightarrow^{*} V_1 \wedge M_2 \longrightarrow^{*} V_2 \Rightarrow \mathcal{R}_{\mathcal{V}}[\![\tau]\!]_{\ell_{Atk}}(V_1, V_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{R}_{\mathcal{S}}[\![\cdot]\!]_{\ell_{Atk}}(\cdot, \cdot) = \top\\
&\mathcal{R}_{\mathcal{S}}[\![\Gamma, x : \tau]\!]_{\ell_{Atk}}((\gamma_1, M_1), (\gamma_2, M_2)) = \mathcal{R}_{\mathcal{E}}[\![\tau]\!]_{\ell_{Atk}}(M_1, M_2) \wedge \mathcal{R}_{\mathcal{S}}[\![S]\!]_{\ell_{Atk}}(\gamma_1, \gamma_2)
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{R}_{\mathcal{T}}[\![\Gamma \vdash_s \tau]\!]_{\ell_{Atk}}(M_1, M_2) = \forall (\gamma_1, \gamma_2 : \cdot \Vdash \Gamma).\ \mathcal{R}_{\mathcal{S}}[\![\Gamma]\!]_{\ell_{Atk}}(\gamma_1, \gamma_2) \wedge \mathcal{W}_{\mathcal{S}}[\![\Gamma]\!](\gamma_1) \wedge \mathcal{W}_{\mathcal{S}}[\![\Gamma]\!](\gamma_2)\\
&\qquad\qquad \Rightarrow \mathcal{R}_{\mathcal{E}}[\![\tau]\!]_{\ell_{Atk}}(\gamma_1(M_1), \gamma_2(M_2))
\end{aligned}
$$

Fig. 9. Logical relation for λ-coIFC^{print}.

The LR for the $\mathit{Labeled}_\ell\ \tau$ type has two cases. As before, if the label flows to the attacker, then the two values have to be related at type τ ($\mathcal{R}_{\mathcal{V}}[\![\tau]\!]_{\ell_{Atk}}(M_1, M_2)$). Otherwise, the terms are required to satisfy the writing-scope predicate. In this case, the writing predicate ensures that the underlying terms do not produce effects which may compromise security.

The relation for monadic values states that if the monadic reduction of both computations terminate ($M_1 \rightsquigarrow^{\star} \mathit{return}(N_1), o_1$ and $M_2 \rightsquigarrow^{\star} \mathit{return}(N_2), o_2$) then the resulting terms are related ($\mathcal{R}_{\mathcal{E}}[\![\tau]\!]_{\ell_{Atk}}(N_1, N_2)$) and the outputs are indistinguishable ($o_1 \simeq_{\ell_{Atk}} o_2$). Moreover,the LR ensures that neither computation produces output on those channels not explicitly mentioned in its type analogously to the writing-scope predicate. The lifting to substitutions is as expected; however, the lifting to open terms requires that the closing substitutions to fulfil the writing-scope predicate. Again, it is needed to strengthen the induction hypothesis. Below, we state the fundamental theorem of the LR for λ-coIFC^{print}.

Theorem 5.4 (Fundamental theorem of $\mathcal{R}_{\mathcal{T}}[\![\_]\!]\_$). *For any program $\Gamma \vdash_s M : \tau$, it holds that $\mathcal{R}_{\mathcal{T}}[\![\Gamma \vdash_s \tau]\!]_{\ell_{Atk}}(M, M)$.*

Proof. By induction on the typing derivation. □

The security property TINI is derived as a corollary of the fundamental theorem.

Corollary 5.5. *Assume an attacker with security label ℓ_{Atk}, a label ℓ_H for secrets, and a program M as described by Definition 3, then M satisfies TINI.*

$$\mathcal{I}_{\mathcal{G}}[\![Unit^s]\!](M_1, M_2)$$
$$= \forall (V_1 V_2 : \cdot \vdash_s Unit).\ M_1 \longrightarrow^* V_1 \wedge M_2 \longrightarrow^* V_2 \Rightarrow V_1 \equiv V_2$$
$$\mathcal{I}_{\mathcal{G}}[\![Bool^s]\!](M_1, M_2) = M_1 \equiv M_2$$
$$= \forall (V_1 V_2 : \cdot \vdash_s Bool).\ M_1 \longrightarrow^* V_1 \wedge M_2 \longrightarrow^* V_2 \Rightarrow V_1 \equiv V_2$$

$$\mathcal{I}_{\mathcal{M}}[\![\Sigma^s]\!]_{\ell_{Atk}}(\theta_1, \theta_2) = \forall l, \ell, \sigma^s.\ (l :_\ell \sigma^s \in \Sigma^s) \wedge (\ell \sqsubseteq \ell_{Atk}) \Rightarrow \mathcal{I}_{\mathcal{G}}[\![\sigma^s]\!]\ell_{Atk}(\theta_1(l), \theta_2(l))$$

Fig. 10. Indistinguishability of ground programs and stores for λ-coIFC^{mem}.

Proof. Let $\cdot \vdash N_1\ N_2 : Bool$ be the secrets to be substituted in M. Since $\ell_H \not\sqsubseteq \ell_{Atk}$, we have that the secrets are related, i.e., $\mathcal{R}_{\mathcal{E}}[\![Labeled_{\ell_H}\ Bool^s]\!]_{\ell_{Atk}}(N_1, N_2)$, and thus we have two closing substitutions $\gamma_1 = \{s \mapsto N_1\}$ and $\gamma_2 = \{s \mapsto N_2\}$ which are also related. By the fundamental theorem, the term M is related to itself which means that $\mathcal{R}_{\mathcal{E}}[\![Eff_{\ell s}\ Bool^s]\!]_{\ell_{Atk}}(\gamma_1(M), \gamma_2(M))$. By assumption we have that $M[N_1/s] \rightsquigarrow^* return(M_1'),\ o_1$ and $M[N_2/s] \rightsquigarrow^* return(M_2'),\ o_2$. From this we obtain that there exists two intermediate programs, M_1'', M_2'' such that $M[N_1/s] \longrightarrow M_1'' \wedge M_1'' \rightsquigarrow^* return(M_1'),\ o_1$ and $M[N_2/s] \longrightarrow M_2'' \wedge M_2'' \rightsquigarrow^* return(M_2'),\ o_2$. We apply $\mathcal{R}_{\mathcal{E}}[\![Eff_{\ell s}\ Bool^s]\!]_{\ell_{Atk}}(M[N_1/s], M[N_2/s])$ to the two pure reductions which gives us that $\mathcal{R}_{\mathcal{V}}[\![Eff_{\ell s}\ Bool^s]\!]_{\ell_{Atk}}(M_1'', M_2'')$, which we apply again to the monadic reductions, obtaining that $o_1 \simeq_{\ell_{Atk}} o_2$. □

5.3 Effectful Language with Memory Effects

The type-system λ-coIFC^{mem}satisfies an analogous security guarantee as TINI for λ-coIFC^{print}—cf. Definition 3. The main difference is that in λ-coIFC^{mem}, we have to take into account the stores, since monadic reduction is only sensible when the program is paired with a store that provides terms for the references. Note that differently from call-by-value, in a call-by-name language terms need not be fully evaluated before storing them in memory. Thus, it is not enough to say that two stores are indistinguishable if they contain syntactically the same terms in positions transparent to the attacker; we need a more refined notion. For that, we lift the security condition TINI to terms in the store, namely, assuming termination of the terms pairwise stored in the memories the resulting values are equal. In Figure 10 we define indistinguishability of stores.

The indistinguishability relation states that two stores are pointwise related at security store typing Σ when both stores agree on the contents—they are indistinguishable by $\mathcal{I}_{\mathcal{G}}[\![\_]\!]$—up to locations that the attacker can not observe ($\ell \sqsubseteq \ell_{Atk}$). Note that $\mathcal{I}_{\mathcal{G}}[\![\_]\!]\_$ is an instance of the logical relation for expressions specialized at ground types.

Next, we state the notion of non-interference for the language with memory.

Definition 4 (TINI for λ-coIFC^{mem}). Assume an attacker with security label ℓ_{Atk}, and a label ℓ_H for secrets such that $\ell_H \not\sqsubseteq \ell_{Atk}$ and a set of security labels ℓs. Also assume a security store typing Σ. Given a program $\Sigma\ ,\ s : Bool \vdash M : Eff\ Bool$, two terms $\Sigma\ , \cdot \vdash N_1, N_2 : Bool$, two initial stores $\theta_1, \theta_2 : \Sigma$, two terms $\Sigma\ , \cdot \vdash M_1', M_2'$ and two final stores $\theta_1', \theta_2' : \Sigma$, such that

- $\{\ell_{Atk}\}\ ;\ \Sigma\ ,\ s : Labeled_{\ell_H}\ Bool \vdash_s M : Eff_{\ell s}\ Bool$
- $\mathcal{I}_{\mathcal{M}}[\![\Sigma]\!]_{\ell_{Atk}}(\theta_1, \theta_2)$
- $\theta_1, M[N_1/s] \rightsquigarrow^* \theta_1', return(M_1') \wedge \theta_2, M[N_2/s] \rightsquigarrow^* \theta_2', return(M_2')$,

then it holds that $\mathcal{I}_{\mathcal{M}}[\![\Sigma]\!]_{\ell_{Atk}}(\theta_1', \theta_2')$.

5.3.1 Soundness of the IFC Type-System for λ-coIFC^{mem}. In order to show that λ-coIFC^{mem} satisfies TINI, we again employ the method of logical relations. In the rest of this section we outline the notable differences between the LR for λ-coIFC^{print} and λ-coIFC^{mem}. We refer the reader to the Agda formalization for further details.

First, we necessitate of a similar logical predicate to writing-scope to show that a computation only may modify those memory cells whose label, i.e., as given by the store typing, appear in the set of labels of its security monadic type. Formally, given a security store typing Σ, a computation $\Sigma \, , \, \cdot \vdash^{s} M : \mathit{Eff}_{\ell s}\, \tau$ and two concrete stores of that type, $\theta_1, \theta_2 : \Sigma$ the predicate states that $\forall l.\mathit{label}(l) \notin \ell s \implies \theta_1(l) \equiv \theta_2(l)$. In words, locations whose labels do not belong to ℓs contain syntactically equal terms in both stores. The logical predicate is also lifted to the contents of the store.

Next, the logical indistinguishability relation from Figure 9 has to be adapted to take into consideration that terms of monadic type take related stores to related stores upon execution. The LR for stores is exactly like the indistinguishability relation from Figure 10. Restricting the contents of stores to terms of ground types avoids stepping into circularity problems in the definition which heavily complicates our development.

A further adaptation occurs in the case of the value relation for terms of monadic type $\mathcal{R}_{\mathcal{V}}[\![\mathit{Eff}_{\ell s}\, \tau]\!]_{\ell_{Atk}}$. In this case, the computation takes any two initial logically related stores to related values and related final stores. Moreover, the initial stores should satisfy the writing-scope predicate, and in return the final stores also satisfy it.

Then the proof of Definition 4 falls as a corollary, in the same vein as the previous two languages, of the fundamental theorem of the logical relation.

Theorem 5.6. *The language λ-coIFC^{print} satisfies TINI as in Definition 4.*

Proof. Follows directly as a corollary of the fundamental theorem of the LR. □

The details can be found in the accompanying Agda formalization.

6 IMPLEMENTATION

In this section, we outline the Haskell implementation of λ-coIFC and λ-coIFC^{print}—the code for handling references follows analogously. The challenging aspect of implementing the calculus from Figures 2 and 4 in Haskell is how to handle the security labels in π. Recall that the symbol π in the type judgement $\pi \, ; \cdot \vdash M : \tau$ represents the set of security labels—intuitively *type-level keys*—that program M has in context. Security labels in the calculus are non first-class objects, i.e., the calculus does not have a type of security labels neither introduction nor elimination rules; accordingly, the implementation should concur.

The set of security labels can also be understood as reflecting the *capabilities* the program can use to open labeled data. In this light, our approach provides an implementation which considers type-level keys—i.e., security labels—as capabilities [20] which should be explicitly exercised. This design decision aligns with the fact that capability-based systems are enough to enforce IFC (e.g., [7, 25]). Another challenge in the implementation is to provide *unforgeable capabilities*. For that we will resort to a combination of Haskell's module system, to hide internals of the implementation from users, and a known trick of second-order polymorphism. Our implementation is simple and elegant, and we believe that it could be also applied to other capability-based system—an interesting direction for future work.

```
1  module CoIFC
2    (Key (), Label (..), FlowsTo (), Labeled (Labeled), label, unlabel, ...)
3  where
4    -- Enumeration of security labels for the two-point lattice
5  data Label = H | L
6    -- Preorder relation as a type-class
7  class FlowsTo (l :: Label) (l′ :: Label)
8    -- Instances for the preorder
9  instance FlowsTo l l
10 instance FlowsTo L H
11   -- Type-level keys
12 data Key l s = Key
13 data Labeled l a = Labeled (∀ s . Key l s → a)
14   -- Labeling terms
15 label :: (∀ s . Key l s → a) → Labeled l a
16 label = Labeled
17   -- Unlabeling terms
18 unlabel :: FlowsTo l′ l ⇒ Key l s → Labeled l′ a → a
19 unlabel k@Key (Labeled f) = f Key
```

Fig. 11. Implementation of λ-coIFC for the two-point security lattice.

6.1 Implementation of λ-coIFC

Figure 11 shows the whole implementation of λ-coIFC in Haskell. Without loss of generality, the implementation assumes a two-point lattice. Similar as previous work (e.g., [4, 10, 36], we represent labels as types of kind *Label* (line 5, and the use of the GHC extension *DataKinds*) and the preorder relation encoded via a type class (lines 7–10).

λ-coIFC treats the set of labels π abstract, and the only rules which modify the set of security labels are [Label] and [Unlabel]—cf. Figure 2. This coincides with the idea that capabilities cannot be either created out of thin air nor deconstructed. Thus, our implementation needs to mimic this behavior within Haskell. We propose to apply the ST monad trick [22] as a solution to not allow keys be treated as first-class objects.

Line 12 introduces a new datatype *Key* which is parametrized by a type l, of kind *Label*. It stands for a label in the security lattice and a phantom type s. Then, line 13 introduces the *Labeled* datatype, which is simply a wrapper over the function space between the type *Key* and an arbitrary type a. In order to ensure IFC safety, it is important that the constructors of the datatype *Key* to be kept abstract from the library user—observe *Key* () in the export list of the module (line 2). Otherwise, anyone—including the attacker—could extract the underlying term of type a from *secret* :: *Labeled l a* by applying the destructor *unLabeled* to the constructor of the key datatype *Key*, i.e., writing *unLabeled secret Key*. In fact, anyone with access to the constructor *Key* has authority to forge any capability. The ∀ (forall) quantifier on the right hand side of *Labeled* prevents keys from being stored within *Labeled* values—see line 13. The idea is that any new key bound by the function will have a unique type variable which would not coincide with any type variable stated on the type, i.e., the program $p = \mathit{Labeled}\ (\lambda k \rightarrow k)$ does not typecheck.

Function *label* (line 15 and 16) is a wrapper over the constructor of the *Labeled* type. The combinator *unlabel* (lines 18–19) allows to unlabel a term of *Labeled* type in case we have a *Key* (i.e., capability) for a label l' which flows to l ($\mathit{FlowsTo}\ l'\ l$). Similar to [37], this function is strict in its argument (*k@Key*) to eliminate the possibility that a forged key like *undefined* :: *Key* can be effectively used to leak secrets. We remark that the security condition TINI —recall Definitions 1 and 3—rules out the execution of those programs where *undefined* is evaluated, i.e., halts the execution with error.

The implementation shown so far consists of the *trusted computing base* (TCB) for our security library. From now on, users of the library can derive functionalities from the interface of the library. For instance, programmers can show that *Labeled l* is a *Functor*, an *Applicative* and a *Monad* for any label l as follows:

```
instance Functor (Labeled l) where
  fmap f x = label (λk → f (unlabel k x))
instance Applicative (Labeled l) where
  pure x = label (λk → x)
  f <*> a = label (λk → (unlabel k f) (unlabel k a))
instance Monad (Labeled l) where
  return = pure
  m >>= f = label (λk → unlabel k (f $ unlabel k m))
```

Observe that these instances can be implemented *only* with the exposed interface, which we take as a sign of the elegance and generality of our approach. We have also implemented the security libraries for writting effect-free programs *Sec* [37] and DCC [1] (in an alternative but equivalent formulation proposed by Algehed [3]).

6.2 Implementation of λ-coIFC^{print}

Figure 12 shows the extension to the implementation in Figure 11 to obtain λ-coIFC^{print}. In order to implement printing effects, we introduce a datatype that *Eff* which is parametrized with a type-level list of the security labels of the channels where the computation might print—see line 5. The type *Eff* is a graded monad over the set of security labels—just like in the calculus. The label information about effects is critical to implement the *distr* rule from Figure 4, i.e., to know what observers an computation has.

Datatype *Eff* wraps *IO*-actions and uses type-level lists (*cs* of kind [*Label*]) to keep track of the channel where the computation can write to. We can simply implement the return and bind of the graded monad, see lines 24–27, where we borrow type level sets from [30] to represent the labels indexing the graded monad. The type of return indicates that it does not produce effects, i.e., the index is the empty set—line 20. The type of bind is annotated with the union of the sets of the first argument and the continuation—line 26. Lastly, we present the subeffecting as the identity function—line 29–30.

The printing effect is embodied by function *printEff* (line 10–11) which builds a computation that produces output at security level l when executed. Argument of type *SLabel l* is simply to provide an argument to indicate on which channel l, of kind *Label*, the output will get produced it—recall that functions' arguments in Haskell are of kind $*$, l has kind *Label*, and *SLabel l* has kind $*$.

```
module CoIFC
  (..., Eff (), FlowsToSet (), pureEff, appEff, returnEff, bindEff, distr, subeff, printEff)
where
 ...
newtype Eff (ls :: [Label]) a = Eff {runEff :: IO a}
  -- Printing
data SLabel :: Label → ∗where
  SH :: SLabel H
  SL :: SLabel L
printEff :: (Show a) ⇒ SLabel l → a → Eff [l] ()
printEff l x = Eff (print (header l) ≫ print x)
  where header :: SLabel l → String
    header SH = "Channel H:"
    header SL = "Channel L:"
  -- Functor Eff
instance Functor (Eff ls) where
  fmap f (Eff io) = Eff (fmap f io)
  -- Applicative
pureEff :: a → Eff [ ] a
pureEff = returnEff
appEff :: Eff ls (a → b) → Eff ls a → Eff (Union ls ls′) b
appEff (Eff ioff) (Eff ioa) = Eff $ ioff <∗> ioa
  -- Monad
returnEff :: a → Eff [ ] a
returnEff a = Eff (return a)
bindEff :: Eff ls₁ a → (a → Eff ls₂ b) → Eff (Union ls₁ ls₂) b
bindEff (Eff m) f = Eff (m ≫= runEff ∘ f)
  -- Sub-typing
subeff :: Subset ls₁ ls₂ ⇒ Eff ls₁ a → Eff ls₂ a
subeff (Eff m) = Eff m
  -- Distr
type family FlowsToSet (l :: Label) (ls :: [Label]) :: Constraint where
  FlowsToSet l₁ [ ]        = ()
  FlowsToSet l₁ (l₂ : ls) = (FlowsTo l₁ l₂, FlowsToSet l₁ ls)
distr :: FlowsToSet l ls ⇒ Labeled l (Eff ls a) → Eff ls (Labeled l a)
distr (Labeled f) = Eff $ fmap (λa → label (const a)) (runEff (f Key))
```

Fig. 12. Implementation of λ-coIFC^{print}.

Finally, the implementation of the primitive *distr* is straightforward—see line 35-36. Type constraint *FlowsToSet l ls* checks that $l \sqsubseteq l'$ for every channel l' in *ls*. The implementation uses a *Key* for label *l* (*eff Key*) to "unlabel" the *IO*-action *effects io* :: *IO a*. The final step is to "label" the result of the *IO*-action with label *l* (*fmap* ($\lambda a \rightarrow$ *label* (*const a*) *io*)).

6.3 Implementing Existing Libraries for IFC

The strength of our type-system as a basis for IFC libraries in Haskell arises from being able to use it as a low level implementation language for already existing libraries such as SecLib [37], DCC [1] and MAC [48].

In the rest of the section we briefly explain their implementations with a focus on the non-standard combinators. To clarify the presentation assume the interface of λ-coIFC is in scope and qualified as *CoIFC* when it is not clear from the context. For a more complete account of DCC's and SecLib's implementations, we refer the reader to the Appendices A and B.

SecLib. SecLib [37] is one of the pioneers of static IFC as a library in the context of a general purpose language such as Haskell. Its main feature is a family of security monads, *Sec*, indexed by labels from the security lattice, each equipped with ≫= (bind) and *return*. *Sec*'s special ingredient is a combinator *up*, which allows coercing values value from a lower security into a higher levels in the security monad. On the left column, the reader can find the important types and combinators exported from the *Sec* library, on the right their implementation using λ-coIFCas a library:

```
type Sec l a                                  type Sec l a = CoIFC.Labeled l a
instance Functor (Sec l) where                ...
instance Monad (Sec l) where                  ...
up :: FlowsTo l l' ⇒ Sec l a → Sec l' a       up :: FlowsTo l l' ⇒ Sec l a → Sec l' a
                                              up lv = label (λk → unlabel k lv)
```

Dependency Core Calculus. DCC [1] is a calculus design to subsume several dependency analysis. Although direct implementations of DCC in Haskell exist [4], we opt for an alterative presentation, S(implified)DCC, due to [3].

The main difference between SDDC and DCC, is that the former avoids the non-standard bind and the *protected at* relation disapears favoring a different set of combinators. Like DCC, SDCC sports a family of monads, one for each security label with *fmap*, *return* and ≫=. Moreover, SDC interface exposes two combinators *up* and *com*. The former serves to relabel values to higher security types and the latter alows to commute the labels of a value with double monadic type. The left column displays SDCC's interface, on the right column its implementation using λ-coIFC.

```
type T l a                                    type T l a = CoIFC.Labeled l a
instance Functor (Sec l) where                ...
instance Monad (Sec l) where                  ...
up :: FlowsTo l l' ⇒ T l a → T l' a           up :: FlowsTo l l' ⇒ Sec l a → Sec l' a
                                              up lv = label (λk → unlabel k lv)
com :: T l (T l' a) → T l' (T l a)            com :: T l (T l' a) → T l' (T l a)
                                              com lv = label (λkl' → label (λkl →
                                                            unlabel kl' (unlabel kl lv)))
```

MAC. MAC [48] is a full-featured library for effectful IFC in Haskell. At its core, MAC defines two types, one for labeled pure values, *Labeled l a* and a another, *MAC l a*, for secure computations. *MAC l a* is a monad for each security label *l*, where the purpose of the label is to, a) protect the data that it is in the context of the computation; and b) restrict what effects the computation is allowed to perform. *MAC*s key is the interaction between labeled values and computations. This is achieved through two primitives *label* and *unlabel*. Unlike λ-coIFC, in order to label a value one needs to be in a computation, there is no notion of a effect-free (non monadic) secure programs.

Below we hint its implementation. In the left column, we show MAC's interface and in the right column its implementation in terms of effectful λ-coIFC.

$\mathbf{type}\ \mathit{Labeled}\ l\ a$	$\mathbf{type}\ \mathit{Labeled}\ l\ a = \mathit{CoIFC.Labeled}\ l\ a$
$\mathbf{type}\ \mathit{MAC}\ l\ a$	$\mathbf{type}\ \mathit{MAC}\ l\ a = \forall\ ls\ .\ \mathit{FlowsToSet}\ l\ ls \Rightarrow \mathit{Labeled}\ l\ (\mathit{Eff}\ ls\ a)$
$\mathit{label} :: \mathit{FlowsTo}\ l\ l' \Rightarrow a \to \mathit{MAC}\ l\ (\mathit{Labeled}\ l'\ a)$	$\mathit{label} :: \mathit{FlowsTo}\ l_1\ l_2 \Rightarrow a \to \mathit{MAC}\ l_1\ (\mathit{Labeled}\ l_2\ a)$ $\mathit{label}\ a = \mathit{label}\ (\lambda k \to (\mathit{returnEff}\ (\mathit{label}\ (\lambda k' \to a))))$
$\mathit{unlabel} :: \mathit{FlowsTo}\ l\ l' \Rightarrow \mathit{Labeled}\ l\ a \to \mathit{MAC}\ l'\ a$	$\mathit{unlabel} :: \mathit{FlowsTo}\ l_1\ l_2 \Rightarrow \mathit{Labeled}\ l_1\ a \to \mathit{MAC}\ l_2\ a$ $\mathit{unlabel}\ lv = \mathit{label}\ (\lambda k \to \mathit{returnEff}\ (\mathit{unlabel}\ k\ lv))$

7 RELATED WORK

Modal logic and programming languages. The pure fragment of our language, i.e., λ-coIFC, has been inspired by several works on applying modal logic to language-based security. The work by Shikuma and Igarashi [40] presents the *sealing calculus*, which captures the same principles behind our label_ℓ and $\mathit{unlabel}_\ell$ annotations but in a call-by-value setting. The sealing calculus is equivalent to DCC [1] and we therefore expect that λ-coIFC enjoys of the same expressivity. The work by Miyamoto and Igarashi [26] gives an informal connection between a classical type-system for IFC and a certain modal logic. Their type-system is very different from ours in that a typing judgement has two separate contexts, inspired by work on dual-context calculi for modal logic, which serve to keep track of globally and locally valid assumptions. Globally valid assumptions are tagged with security labels which represents the sensitivity of the variable. Further, the judgement is also indexed by a label which somehow stands for the *current* security level, and serves to constraint when variables from the global context can be used in terms. The authors also give semantics for their label_ℓ term which has a computation cost (i.e., a subsitution). In contrast, label_ℓ in λ-coIFC has no runtime representation, and therefore no cost at runtime. Recently, the work by Abel and Bernardy [2] presents a unify treatment of modalities in typed lambda calculi. The authors essentially present a side-effect free lambda calculus parametrized on a modality with certain mathematical structure, and show many PL analyses, including IFC, can be obtained as an instantiation of their framework. As we have shown, our type-system has a rather straightforward implementation in Haskell, in contrast is not very clear how would one implement theirs since it would require a fine-grained control over the variables in the context which is difficult to achieve for a shallow embedding.

Coeffects type-systems. Recently, there is a line of work that suggests using coeffect type systems to enforce information-flow control. The work by Petricek et al. [32] develops a calculus to capture different granulatiry demands on contexts, i.e., flat whole-context coeffetcs (like implicit parameters [23]) or structural per-variable ones (like usage or data access patterns). The work by Gaboardi et al. [12] expand and use graded comonads and monads to combine coeffects and effects. The authors describe *distributivity laws* for the graded comonads and monads for situations similar to what our primitive *distr* addresses. The article suggests IFC as an application domain where the coeffect system captures the IFC constraints and the effect system gives semantics to effects, concretely non-determinism. The distributive laws, then explain how both are combined. However, their work does not state neither proves a concrete security property for their calculus. Their interest lays in constructing a categorical model with a flavour of the usual erasure for IFC languages. Different from it, our work does not use comonads as the underlying structure for IFC and considers general reading and writing effects. Granule is a recent programming language [29] based on *graded modal types* to impose usage constraints, and encompassing coeffect types and graded monads for effects. This work also hints about IFC as an application domain by changing the interpretation of the coffect grades in a pure setting.

IFC for pure and effectful languages. The work by Austin et al. [5] analyzes dynamic IFC for an imperative language in terms of a IFC-aware pure lambda calculus. This work relies on adding security annotations to lambda terms' evaluation contexts as well as that insecure effects can be rolled back, i.e., writing to a public location of memory gets ignored if a secret is available in scope. In a similar spirit, Hirsch and Cecchetti [19] develop a formal framework based on productors and a type-and-effects systems to characterize how effectful languages can be translated into a pure subset—an idea that informally hint by Algehed and Russo [4]. In contrast, our approach is conceived to separate side-effect free and side-effectful parts by design, where we use a modal operator for the side-effect free terms and a graded monad for the side-effectful ones.

Logical relations for NI. Both Heintze and Riecke [18], and Zdancewic and Myers [51] use logical relation to prove non-interference for a simply-typed security lambda calculus. Tse and Zdancewic [45] apply logical relation to prove soundness of a translation from DCC [1] to System F. Unfortunately, the translation is not sound [40]. Bowman and Ahmed [8] use existential types (expressed as universal ones) to provide a valid DCC to System F_ω translation—their soundness theorem is also based on logical relations. Different from the cited work so far, Rajani and Garg [34] introduce logical relation to prove non-interference for a language with references. In this line of work, Gregersen et al. [15] extend the use of logical relation to prove NI for languages with predicative polymorphism. Different from [15, 34], we consider only first-order references for simplicity. Otherwise, we should have had to utilize a step-indexed Krypke-style logical-relations model, which would have introduced technical complications that are orthogonal to the main contribution of our work.

8 CONCLUSIONS

In this paper, we shed light on a point in the design space of information-flow control libraries in the context of pure languages with effects. To achieve so, we have introduced several type-systems for IFC, λ-coIFC, λ-coIFC^{print} and λ-coIFC^{mem}, which build atop a pure call-by-name programming language with modal types and explicit effects of a monadic type. We show that to secure computations is enough to constraint what computations we allow to run rather than restricting which ones are allowed to be programed. After all, in a pure language, computations are first-class objects! Such an insight translates into a single primitive which controls when computations built depending on labeled information can be executed. We hope that coIFC revitalizes the importance of modalities for IFC research and opens the door to a new design space for security libraries.

REFERENCES

M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. 1999. A Core Calculus of Dependency. In *ACM Symp. on Principles of Programming Languages*. 147–160.

Andreas Abel and Jean-Philippe Bernardy. 2020. A unified view of modalities in type systems. *Proc. ACM Program. Lang.* 4, ICFP (2020).

Maximilian Algehed. 2018. A Perspective on the Dependency Core Calculus. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*. 24–28.

Maximilian Algehed and Alejandro Russo. 2017. Encoding DCC in Haskell. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, PLAS@CCS*.

Thomas H. Austin, Cormac Flanagan, and Martín Abadi. 2012. A Functional View of Imperative Information Flow. In *Programming Languages and Systems - 10th Asian Symposium, APLAS 2012*.

David E. Bell and L. La Padula. 1976. *Secure Computer System: Unified Exposition and Multics Interpretation*. Technical Report MTR-2997, Rev. 1. MITRE Corporation, Bedford, MA.

Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld. 2011. Capabilities for information flow. In *Proceedings of the 2011 Workshop on Programming Languages and Analysis for Security, PLAS 2011*.

William J. Bowman and Amal Ahmed. 2015. Noninterference for Free. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/2784731.2784733

Niklas Broberg, Bart van Delft, and David Sands. 2013. Paragon for Practical Programming with Information-Flow Control.. In *APLAS (LNCS, Vol. 8301)*. Springer, 217–232.

P. Buiras, D. Vytiniotis, and A. Russo. 2015. HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell. In *ACM SIGPLAN International Conference on Functional Programming*. ACM.

Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (1976), 236–243.

Marco Gaboardi, Shin-ya Katsumata, Dominic A. Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining effects and coeffects via grading. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP*.

Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. 2012. Hails: Protecting Data Privacy in Untrusted Web Applications. In *10th Symposium on Operating Systems Design and Implementation*.

J.A. Goguen and J. Meseguer. 1982. Security policies and security models. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society.

Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. 2021. Mechanized logical relations for termination-insensitive noninterference. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29.

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

D. Hedin and A. Sabelfeld. 2011. A Perspective on Information-Flow Control. In *2011 Marktoberdorf Summer School*. IOS Press.

N. Heintze and J. G. Riecke. 1998. The SLam calculus: programming with secrecy and integrity. In *ACM Symp. on Principles of Programming Languages*. 365–377.

Andrew K. Hirsch and Ethan Cecchetti. 2021. Giving Semantics to Program-Counter Labels via Secure Effects. *Proc. ACM Program. Lang.* 5, POPL (Jan. 2021).

Richard Y. Kain and Carl E. Landwehr. 1986. On Access Checking in Capability-Based Systems. In *IEEE Symp. on Security and Privacy*. 95–101.

B. W. Lampson. 1973. A Note on the Confinement Problem. *Commun. ACM* 16, 10 (1973).

John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*.

Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark Shields. 2000. Implicit Parameters: Dynamic Scoping with Static Types. In *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

P. Li and S. Zdancewic. 2006. Encoding Information Flow in Haskell. In *IEEE Workshop on Computer Security Foundations*. IEEE Computer Society.

Mark Miller, Ka-Ping Yee, Jonathan Shapiro, and Combex Inc. 2003. *Capability Myths Demolished*. Technical Report. Johns Hopkins University Systems Research Laboratory.

Kenji Miyamoto and Atsushi Igarashi. 2004. A modal foundation for secure information flow. In *In Proceedings of IEEE Foundations of Computer Security (FCS)*. 187–203.

Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. 2001. Jif: Java Information Flow. (2001). http://www.cs.cornell.edu/jif.

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.* 3, ICFP (2019), 110:1–110:30.

Dominic Orchard and Tomas Petricek. 2014. Embedding effect systems in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell.* 13–24.

James Parker, Niki Vazou, and Michael Hicks. 2019. LWeb: information flow security for multi-tier web applications. *Proc. ACM Program. Lang.* 3, POPL (2019), 75:1–75:30. https://doi.org/10.1145/3290388

Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. 2014. Coeffects: a calculus of context-dependent computation. In *Proceedings of the 19th ACM SIGPLAN International conference on Functional programming.* 123–135.

Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. 2020. Liquid information flow control. *Proc. ACM Program. Lang.* 4, ICFP (2020), 105:1–105:30. https://doi.org/10.1145/3408987

Vineet Rajani and Deepak Garg. 2020a. On the expressiveness and semantics of information flow types. *J. Comput. Secur.* 28, 1 (2020), 129–156. https://doi.org/10.3233/JCS-191382

Vineet Rajani and Deepak Garg. 2020b. On the expressiveness and semantics of information flow types. *Journal of Computer Security* 28, 1 (2020), 129–156.

Alejandro Russo. 2015. Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell. In *ACM SIGPLAN International Conference on Functional Programming (ICFP).* ACM.

A. Russo, K. Claessen, and J. Hughes. 2008. A library for light-weight information-flow security in Haskell. In *ACM SIGPLAN symposium on Haskell.* ACM.

A. Sabelfeld and A. C. Myers. 2003. Language-Based Information-Flow Security. *IEEE J. Selected Areas in Communications* 21, 1 (2003), 5–19.

Daniel Schoepe, Daniel Hedin, and Andrei Sabelfeld. 2014. SeLINQ: tracking information across application-database boundaries. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014.*

Naokata Shikuma and Atsushi Igarashi. 2006. Proving Noninterference by a Fully Complete Translation to the Simply Typed λ-Calculus. In *Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues (ASIAN'06).* Springer-Verlag, Berlin, Heidelberg.

V. Simonet. 2003. The Flow Caml System. (2003). Software release at http://cristal.inria.fr/~simonet/soft/flowcaml/.

D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. 2011. Flexible Dynamic Information Flow Control in Haskell. In *ACM SIGPLAN Haskell symposium.*

Ross Tate. 2013. The sequential semantics of producer effect systems. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 15–26.

Stephen Tse and Steve Zdancewic. 2004a. Translating Dependency into Parametricity. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming* (Snow Bird, UT, USA) *(ICFP '04).* Association for Computing Machinery, New York, NY, USA, 115–125. https://doi.org/10.1145/1016850.1016868

S. Tse and S. Zdancewic. 2004b. Translating dependency into parametricity. In *ACM SIGPLAN International Conference on Functional Programming.* ACM.

Marco Vassena, Pablo Buiras, Lucas Waye, and Alejandro Russo. 2016. Flexible Manipulation of Labeled Values for Information-Flow Control Libraries. In *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I.* 538–557.

Marco Vassena and Alejandro Russo. 2016. On Formalizing Information-Flow Control Libraries. In *Proc. of ACM Workshop on Programming Languages and Analysis for Security (PLAS '16).* ACM.

Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. 2018. MAC A verified static information-flow control library. *Journal of Logical and Algebraic Methods in Programming* 95 (2018), 148–180. https://doi.org/10.1016/j.jlamp.2017.12.003

Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, and Deian Stefan. 2019. From fine- to coarse-grained dynamic information flow control and back. *Proc. ACM Program. Lang.* 3, POPL (2019).

Philip Wadler and Peter Thiemann. 2003. The marriage of effects and monads. *ACM Transactions on Computational Logic (TOCL)* 4, 1 (2003), 1–32.

Stephan Arthur Zdancewic and Andrew Myers. 2002. *Programming Languages for Information Security.* Ph.D. Dissertation. USA.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making information flow explicit in HiStar. In *USENIX Symp. on Operating Systems Design and Implementation.* USENIX.

A IMPLEMENTATION OF THE SECLIB LIBRARY (PURE FRAGMENT)

```
module SecLib
  (Sec, return, (>>=), fmap, up, FlowsTo, Label (..))
where
import CoIFC
import Prelude hiding (fmap, (>>=), return)
type Sec l a = Labeled l a
return :: a → Sec l a
return a = label (λk → a)
fmap :: (a → b) → Sec l a → Sec l b
fmap f lv = label (λk → f (unlabel k lv))
(>>=) :: Sec l a → (a → Sec l b) → Sec l b
lv >>= f = label (λk → unlabel k (f (unlabel k lv)))
up :: FlowsTo l l′ ⇒ Sec l a → Sec l′ a
up lv = label (λk → unlabel k lv)
```

B IMPLEMENTATION OF THE (SIMPLIFIED) DEPENDENCY CORE CALCULUS

```
module SDCC
    (T, eta, mu, mapT, up, com, FlowsTo, Label (..))
where
import CoIFC
type T l a = Labeled l a
eta :: a → T l a
eta a = label (λk → a)
mapT :: (a → b) → T l a → T l b
mapT f lv = label (λk → f (unlabel k lv)
mu :: T l (T l a) → T l a
mu lv = label (λk → unlabel k (unlabel k lv))
up :: FlowsTo l l′ ⇒ T l a → T l′ a
up lv = label (λk → unlabel k lv)
com :: T l (T l′ a) → T l′ (T l a)
com lv = label (λkl′ → label (λkl → unlabel kl′ (unlabel kl lv)))
```