Pure Information-Flow Control with Effects Made Simple

No Author Given

No Institute Given

Abstract. Information-flow control (IFC) is a promising technology to protect data confidentiality. The foundational work on the Dependency Core Calculus (DCC) positions monads as a suitable abstraction for enforcing IFC. Pure functional languages with effects, like Haskell, can provide IFC as a library (MAC, LIO etc.), a minor task compared to implementing compilers for IFC from scratch.
Previous works on IFC as a library introduce ad hoc primitives to type programs whose effects do not depend on the sensitive data in context. In this work, we start afresh and ask ourselves: what would we need to extend an effect-free language for IFC (e.g., DCC) with secure effects? The answer turns out to be elegant and simple. In a pure language with effects there is a natural place where information flows from sensitive data to effects need to be restricted, and when effects are tracked in a more fine-grained fashion than Haskell's IO monad (e.g., with a graded monad) then a *single* primitive is enough to allow the secure flows!
To support our insight, we present and prove secure several IFC enforcement mechanisms based on extensions of the Sealing Calculus (SC) with effects using a graded monad. Effects that depend on sensitive data are secured through a novel primitive distr. All of our security guarantees are mechanized in the Agda proof assistant. Moreover, we provide an implementation of these mechanisms as a new Haskell library for IFC.

1 Introduction

Information-flow control (IFC) [35, 13] 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 property known as noninterference [11]. IFC mechanisms specify the sensitivity of data via *labels*, and then enforce security by controlling that the flows of information abide by the security policy. In the simplest scenario, there are two labels (alternatively, security levels or sensitivities) `public` and `secret` and the security policy specifies that every flow is allowed except from `secret` to `public`—i.e., flows from more to less sensitivity are forbidden. In static approaches to IFC, the sensitivity of data is known a priori, e.g., by specifying it using types, and the enforcement statically decides, e.g., during type checking, whether the program will leak information upon execution. To protect confidentiality of data, IFC mechanisms need to protect against two kinds of potentially malicious flows: 1) *explicit flows*, when `public` data directly depends on `secret` data; and

2) *implicit flows*, when the control flow and `public` outputs of the program are indirectly influenced by `secret` data, e.g., due to branching on a `secret`.

In recent years, the use of pure functional languages has been proliferating for tackling different IFC challenges (e.g., Vassena et al. [44], Parker, Vazou, and Hicks [29], Polikarpova et al. [31], and Rajani and Garg [32]). From a pragmatic perspective, pure functional languages can provide IFC security via libraries [22, 34, 39], which is less demanding than building compilers from scratch (e.g., Simonet [38] and Myers et al. [26]).

When it comes to controlling information flows, pure languages stand in a privileged position thanks to their abstraction facilities and strong encapsulation of effects. For instance, the popular Dependency Core Calculus (DCC) [1] utilizes the abstract type $\mathsf{T}_{\texttt{secret}}\, A$ to label pieces of data of type A with sensitivity `secret` and then the type system ensures that data can only be eliminated into—or influence—data of equal or higher sensitivity. DCC's security guarantees ensure that programs without effects are secure.

A different strand of work aims to provide security in a pure language by restricting the interplay between sensitive data and `public` effects (e.g., LIO [39], MAC [33] and HLIO [9]). In a pure language like Haskell, the *only* programs that can produce effects and thereby interact with the external world have to be of type $\mathsf{IO}\, A$, for some type A. In this light, in order to protect against implicit flows through effects it is enough to control which programs of IO type are safe to execute. To realize this idea, the MAC library, for example, replaces the IO monad with a custom monad MAC_l of computations that is indexed by a type-level label l. The label l has two purposes: (i) akin to DCC, it is an *upper bound* on the sensitivity of the information "going into" the monad, as well as (ii) it is a *lower bound* on the observers' effects—restricting information "leaving" the monad. Concretely, a computation of type $\mathsf{MAC}_{\texttt{public}}\, \mathsf{Bool}$ *cannot* branch on `secret` values but can perform `public` and `secret` effects; in contrast, a computation of type $\mathsf{MAC}_{\texttt{secret}}\, \mathsf{Bool}$ can branch on `secret` data but *cannot* perform `public` effects. In this manner, MAC prevents explicit and implicit flows by construction.

However, not everything in the garden is rosy. The label l in the MAC monad MAC_l does too many things at once. This leads to situations where the programmer needs to go through some contortions or use ad hoc primitives like join[1]: $\mathsf{MAC}_l\, \mathsf{Unit} \Rightarrow \mathsf{MAC}_{l'}\, \mathsf{Unit}$ (restricted to $l' \sqsubseteq l$) [41]. To illustrate this point, we extend the two-point security policy with two incomparable labels `Alice` and `Bob` such that `public` $\sqsubseteq$ `Alice` $\sqsubseteq$ `secret` and `public` $\sqsubseteq$ `Bob` $\sqsubseteq$ `secret` *but* neither `Alice` $\sqsubseteq$ `Bob` nor `Bob` $\sqsubseteq$ `Alice` (the relation $\sqsubseteq$ specifies the permitted flows). With that in mind, let us consider the following program in MAC which receives an `Alice`-sensitive Bool, i.e., *alice_sec* : $\mathsf{MAC}_{\texttt{Alice}}\, \mathsf{Bool}$,[2] and prints a

[1] The type of join is more general but this simplified form suffices for our purposes.

[2] To the reader familiar with MAC: our point applies equally if one uses the $\mathsf{Labeled}_l$ type to protect sensitive Booleans.

string on the `Alice`-observable channel $\mathtt{Ch}_{\mathtt{Alice}}$:

$$\begin{aligned} &\mathit{prog}_1 : \mathsf{MAC}_{\mathtt{Alice}}\,\mathsf{Bool} \Rightarrow \mathsf{MAC}_{\mathtt{Alice}}\,\mathsf{Unit} \\ &\mathit{prog}_1\ \mathit{alice\_sec} = \mathit{alice\_sec} \gg\!= \lambda b.\ \mathsf{if}\ b\ \mathsf{then}\ \mathsf{print}_{\mathtt{Ch}_{\mathtt{Alice}}}(\texttt{"Alice is here!"}) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \mathsf{return}(\mathsf{unit}) \end{aligned}$$

In the above program, the information flows according to the security policy (from the `Alice` value to `Alice`'s channel)—i.e., the program is secure. Consider now a different program that combines prog_1 with printing on the channel $\mathtt{Ch}_{\mathtt{Bob}}$:

$$\begin{aligned} &\mathit{prog}_2 : \mathsf{MAC}_{\mathtt{Alice}}\mathsf{Bool} \Rightarrow \mathsf{MAC}\ \boxed{?}\ \mathsf{Unit} \\ &\mathit{prog}_2\ \mathit{alice\_sec} = \mathit{prog}_1\ \mathit{alice\_sec} \gg \mathsf{print}_{\mathtt{Ch}_{\mathtt{Bob}}}(\texttt{"Hi Bob"}) \end{aligned}$$

Clearly, prog_2 is still secure since the decision to print to `Bob`'s channel and what is printed does not depend on the contents of the `Alice`-sensitive value *alice_sec*. What label then should replace $\boxed{?}$ in its type? First, the types of the computations on both sides of the bind ($\gg$) have to match. By (i) and (ii) the type of $\mathit{prog}_1\ \mathit{sec}$ has to be $\mathsf{MAC}_{\mathtt{Alice}}\,\mathsf{Unit}$, and by (ii) $\mathsf{print}_{\mathtt{Ch}_{\mathtt{Bob}}}(\texttt{"Hi Bob"})$ has to be of type $\mathsf{MAC}_{\mathtt{Bob}}\,\mathsf{Unit}$. There is a type mismatch! Since the label `public` is a lower bound of `Alice` and `Bob`, we can use MAC's join to fix the program:

$$\begin{aligned} &\mathit{prog}_2 : \mathsf{MAC}_{\mathtt{Alice}}\,\mathsf{Bool} \Rightarrow \mathsf{MAC}_{\mathtt{public}}\,\mathsf{Unit} \\ &\mathit{prog}_2\ \mathit{alice\_sec} = \mathsf{join}(\mathit{prog}_1\ \mathit{alice\_sec}) \gg \mathsf{join}(\mathsf{print}_{\mathtt{Ch}_{\mathtt{Bob}}}(\texttt{"Hi Bob"})) \end{aligned}$$

Now, something unexpected happened. To assign a type to prog_2, which only mentions labels `Alice` and `Bob`, we *have to* resort to another label `public`. What is worse, in the type of prog_2 only the label `Alice` appears and does so in an argument position which means we know nothing about the program's possible effects. The design decision of indexing the monad MAC by a *single* label, while enabling a simple implementation of IFC as a library (cf. [33]), requires the application of ad hoc primitives (like join) to mitigate over-approximations of how information flows in and out of computations.

In this work, we take a step back and ask ourselves: what would it take to allow arbitrary effects in a pure language with an already existing mechanism for effect-free IFC (e.g., DCC)? The answer turns out to be elegant and simple. We observe that enforcing IFC in a pure language with effects can be reduced to the *single* point—which we will explain below—where effects and sensitive data interact. Based on this observation, we present a novel IFC mechanism which is arguably simpler than existing IFC libraries, allows to assign more natural types to programs; and overcomes the programming contortions discussed above.

To briefly present our idea, let us assume that effect-free IFC is achieved using a DCC-style abstract type $\mathsf{T}_l\ A$, and we have at our disposal a more refined type constructor Eff of effectful programs akin to IO but annotated with concrete information about *observable* effects. For example, Eff could be annotated with the set of channels where the program might print, the set of exceptions the program might raise, or the set of memory references the program might modify. Moreover, we assume the security policy to specify the sensitivity of each effect.

Consider, for instance, a scenario with two output channels, namely $\mathtt{Ch}_{\mathtt{public}}$ and $\mathtt{Ch}_{\mathtt{secret}}$, that are assigned sensitivities `public` and `secret`, respectively.

In this scenario, a program $prog_3 : \mathsf{T}_{\mathtt{Alice}}\,(\mathsf{Eff}_{\{\mathtt{Ch}_{\mathtt{Bob}}\}}\,\mathsf{Unit})$ is a computation that might print to `Bob`'s channel $\mathtt{Ch}_{\mathtt{Bob}}$ depending on `Alice`-sensitive data. This is where sensitive data and effects interact! Assuming $alice\_sec : \mathsf{T}_{\mathtt{Alice}}\,\mathsf{Bool}$ is in scope, $prog_3$, for instance, could have the following implementation:[3]

$$
\begin{aligned}
&prog_3 : \mathsf{T}_{\mathtt{Alice}}\,(\mathsf{Eff}_{\{\mathtt{Ch}_{\mathtt{Bob}}\}}\,\mathsf{Unit})\\
&prog_3 = \mathsf{bind}\; b = alice\_sec\\
&\qquad \mathsf{in}\; \mathsf{return}_{\mathtt{Alice}}(\mathsf{if}\; b\; \mathsf{then}\; \mathsf{print}_{\mathtt{Ch}_{\mathtt{Bob}}}(\texttt{"So it was true, huh"})\\
&\qquad\qquad\qquad \mathsf{else}\; \mathsf{return}(\mathsf{unit}))
\end{aligned}
$$

In order to run the effects of the inner computation of type $\mathsf{Eff}_{\{\mathtt{Bob}\}}\,\mathsf{Unit}$ we have to "extract" it first from the $\mathsf{T}_{\mathtt{Alice}}$ value—recall that the language is pure. In general, extracting anything from $\mathsf{T}_{\mathtt{Alice}}$ is prohibited as it would render the IFC enforcement unsound: in $prog_3$, the computation of type $\mathsf{Eff}_{\{\mathtt{Ch}_{\mathtt{Bob}}\}}\,\mathsf{Unit}$ would become executable, and if the `Alice`-sensitive Boolean $alice\_sec$ is true, it will print `"So it was true, huh"` on `Bob`'s channel $\mathtt{Ch}_{\mathtt{Bob}}$—a flow that violates the security policy. On the contrary, let us consider a program of a different type, $prog_4 : \mathsf{T}_{\mathtt{Alice}}\,(\mathsf{Eff}_{\{\mathtt{Ch}_{\mathtt{secret}}\}}\,\mathsf{Unit})$. In this program, the computation is *definitely* safe to run since the only possible effects it might produce are printing to the `secret` channel $\mathtt{Ch}_{\mathtt{secret}}$ and the decision on what to print depends on data of at most sensitivity `Alice`—a flow permitted by the security policy. Securing effectful programs then amounts to allowing $prog_4$ to extract the computation of type $\mathsf{Eff}_{\{\mathtt{Ch}_{\mathtt{secret}}\}}\,\mathsf{Unit}$ from $\mathsf{T}_{\mathtt{Alice}}$ and run its observable effects while forbidding $prog_3$ from doing so. To achieve this, we introduce a novel primitive distr which systematically permits computations to be extracted, and hence, executed only when they are known to depend on data less sensitive than their observers. With distr we can turn $prog_4$ into an executable program: $\mathsf{distr}(prog_4) : \mathsf{Eff}_{\{\mathtt{Ch}_{\mathtt{secret}}\}}\,\mathsf{Unit}$. We have reduced the enforcement of IFC in pure languages with effects to a single primitive; this gives us *modularity*, *clarity* and *simplicity* in the language design and its possible implementations. Thanks to distr, we can express $prog_2$ with a more natural type:

$$
\begin{aligned}
&prog_2' : \mathsf{T}_{\mathtt{Alice}}\,\mathsf{Bool} \Rightarrow \mathsf{Eff}_{\{\mathtt{Ch}_{\mathtt{Alice}},\mathtt{Ch}_{\mathtt{Bob}}\}}\,\mathsf{Unit}\\
&prog_2'\; alice\_sec = \mathsf{distr}(\mathsf{bind}\; b = alice\_sec\\
&\qquad\qquad \mathsf{in}\; \mathsf{return}(\mathsf{if}\; b\; \mathsf{then}\; \mathsf{print}_{\mathtt{Ch}_{\mathtt{Alice}}}(\texttt{"Alice is here!"})\\
&\qquad\qquad\qquad\qquad \mathsf{else}\; \mathsf{return}(\mathsf{unit})))\\
&\qquad \gg \mathsf{print}_{\mathtt{Ch}_{\mathtt{Bob}}}(\texttt{"Hi Bob"})
\end{aligned}
$$

Our contributions In this work, we show that IFC in the context of pure languages with effects can be achieved through the combination of the following features: 1. an enforcement for effect-free IFC, 2. a type for tracking observable effects in a fine-grained fashion, and 3. a primitive distr which selectively permits to execute effectful computations which depend on sensitive data. We present our idea through an IFC enforcement mechanism for the simply-typed lambda-calculus (STLC). The mechanism is based on the Sealing Calculus (SC) [37], which is more expressive than DCC (cf. [40, 37]), although we could have chosen DCC instead. We then extend the language and IFC enforcement mechanism

[3] bind is DCC's eliminator for the type T_l.

in two different directions with printing and global store effects in the form of graded monads [18]. In these extensions, we include the distr primitive which allows to type check more effecful programs that are secure. Along with our informal argumentation for why our idea is secure for different instantiations of effects, we have mechanized proofs in the Agda proof assistant [2] about the strong security guarantees that the programs in the languages satisfy, namely termination-insensitive noninterference (TINI). Our proofs are based on the technique of logical relations [23]. Finally, we realize our idea in the form of a new Haskell library which we call SCLib. The conciseness of our implementation illustrates the elegance and simplicity of our insight: less than 10 lines of code for the effect-free fragment and less than 30 for the part with effects. In order to implement the effect-free IFC mechanism we also present a novel implementation of SC using an encoding of contextual information as type-level capabilities via higher-rank polymorphism [20]. Our library is at least as expressive as previous work on libraries for IFC in Haskell, which we evidence by showing implementations of SecLib [34], DCC (in its presentation by Algehed [4]) and MAC in terms of SCLib's interface.

In summary, the technical contributions of this paper are:

- A reinterpretation of the Sealing Calculus as an IFC enforcement mechanism for STLC (Section 2)
- Two extensions of STLC and SC for enforcing IFC in pure languages with effects via graded monads. As examples, we consider printing (Section 3.1) and global store (Section 3.2) effects
- We present distr, a single primitive that can control the interaction of sensitive data and effects
- Security guarantees and proofs of TINI based on logical relations for all the enforcements (Section 4)
- A Haskell implementation using a novel encoding of contextual information as capabilities together with evidence that SCLib can encode existing monadic security libraries (Section 5)
- Mechanized proofs of all our security guarantees (approx. 1500 lines of Agda code submitted as accompanying material)

2 Effect-free Information-Flow Control

In this section, we briefly recall the Sealing Calculus (SC) and explain its role as an IFC enforcement mechanism for programs written in the STLC.

SC utilizes an abstract type $\mathsf{S}_l\, A$ for protecting sensitive data.[4] A value of type $\mathsf{S}_l\, A$ is "sealed" at sensitivity l in the sense that it is only available to observers with sensitivity at least as high as l. Values of type $\mathsf{S}_l\, A$ are introduced and eliminated using the primitives seal_l and unseal_l. SC enforces IFC by restricting in which *contexts* a sealed value can be "unsealed". For example, the `Alice`-sensitive Boolean $sec :: \mathsf{S}_{\texttt{Alice}}\, \mathsf{Bool}$ can only be unsealed in contexts of at least sensitivity `Alice`.

[4] In the original presentation, the authors use the notation $[A]^l$ instead.

Let us illustrate SC with a program that receives two Booleans with sensitivities `Alice` and `Bob`, respectively, and computes their conjunction with sensitivity `secret` (and : Bool × Bool ⇒ Bool implements conjunction):

$$and' : \mathsf{S}_{\texttt{Alice}}\,\mathsf{Bool} \Rightarrow \mathsf{S}_{\texttt{Bob}}\,\mathsf{Bool} \Rightarrow \mathsf{S}_{\texttt{secret}}\,\mathsf{Bool}$$
$$and' = \lambda(sb_1\ sb_2.\,\mathsf{seal}_{\texttt{secret}}(\mathsf{and}(\mathsf{unseal}_{\texttt{Alice}}(sb_1), \mathsf{unseal}_{\texttt{Bob}}(sb_2))))$$

In the above program, the term $\mathsf{seal}_{\texttt{secret}}$ provides the context in which sb_1 and sb_2 can be unsealed: the term $\mathsf{unseal}_{\texttt{Alice}}(sb_1)$, for instance, is only well-typed because its label `Alice` flows to `secret`, which is the highest label.

We take a variant of the STLC as the programming language on which we want to enforce IFC: it has call-by-name semantics and Unit and Bool as the only primitive types. We work directly with intrinsically well-typed terms, and thus, for us a typing derivation $\Gamma \vdash t : a$ *is* a term. The small-step sematics is specified by the relation $t \longrightarrow t'$, and we call values those terms t for which $t \not\longrightarrow$. For reference we include its complete definition in Appendix A. Since the STLC is well-understood we focus on SC.

Figure 1 presents the intrinsically-typed syntax of SC. The whole development is parameterized by a security policy specified in the form of a lattice structure on the set of labels $(\mathcal{L}, \sqsubseteq)$. The types reflect those in the STLC and include the new type constructor S_l. Typing judgements are of the form $\pi\ ;\ \Gamma \vdash^{\mathsf{sc}} t : A$ where: π is a non-empty *finite* set of labels drawn from $\mathcal{L}$, i.e., $\pi \subseteq \mathcal{L}$; Γ is an SC typing context; and A is an SC type. The component π is analogous to protection contexts from related work by Tse and Zdancewic [40].

The set of labels π in the typing judgement represent the sensitivities of all the data on which the program may depend. In order to clarify the role of π, let us consider a program with a typing derivation indexed by the set of labels $\pi_1 \coloneqq \{\texttt{secret}\}$, $\pi_1\ ;\ \cdot \vdash^{\mathsf{sc}} p : A$. This program may depend on data of types $\mathsf{S}_{\texttt{public}}\,A$ and $\mathsf{S}_{\texttt{secret}}\,A$ by unsealing. If, instead, p is indexed by the set $\pi_2 \coloneqq \{\texttt{public}\}$, i.e., $\pi_2\ ;\ \cdot \vdash^{\mathsf{sc}} p : A$, then the only terms that the program can unseal are of type $\mathsf{S}_{\texttt{public}}\,A$. This mechanism ensures that the flows of information are secure. It is useful to think that the labels that belong to π act as a kind of type-level *key* whose "possession" permits access to information at most as sensitive as the label itself.

The typing rules of the STLC fragment of SC, i.e., Rules Lam, App and If, are rather standard: they simply propagate the set of labels π to their premises. Observe that one has to explicitly unseal sensitive Booleans, i.e., of type $\mathsf{S}_l\,\mathsf{Bool}$, in order to branch on them using the Rule If. Rules Seal and Unseal are the most interesting since they enforce that information flows to the appropriate places. Rule Seal serves a double purpose: from premise to conclusion, it introduces terms of type $\mathsf{S}_l\,A$; and, from conclusion to premise, it extends the set of labels with the label l, i.e., $\pi \cup \{l\}$. The typing derivation above the premise can then unseal any term of type $\mathsf{S}_{l'}\,A$ such that its label l' can flow to l. Rule Unseal allows unsealing a term with type $\mathsf{S}_l\,A$ if the set π in its conclusion contains at least a label l' such that $l \sqsubseteq l'$. Continuing with the intuition of labels in π as keys, a key $l' \in \pi$ can be used to unseal terms of type $\mathsf{S}_l\,A$ exactly when $l \sqsubseteq l'$.

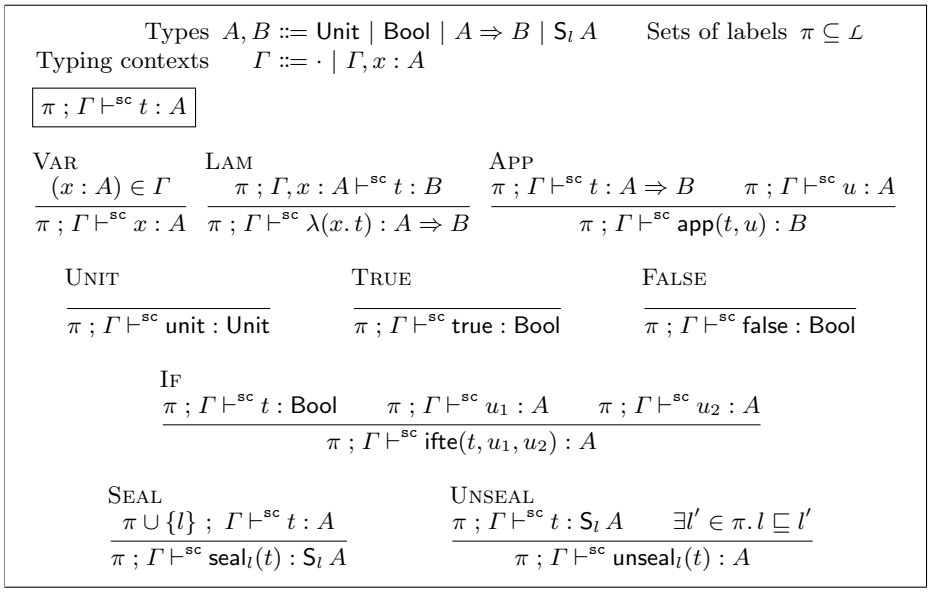

Fig. 1. Types and intrinsically-typed terms of SC

In order to use SC to enforce IFC on STLC programs, we define an erasure function from SC terms $\pi \,;\, \Gamma \vdash^{\mathsf{sc}} t : A$ to STLC programs $\varepsilon(\Gamma) \vdash \varepsilon(t) : \varepsilon(A)$:

$$\begin{array}{lll} \varepsilon(\mathsf{Unit}) = \mathsf{Unit} & \varepsilon(\cdot) = \cdot & \varepsilon(\lambda(x.\, t)) = \lambda(x.\, \varepsilon(t)) \\ \varepsilon(\mathsf{Bool}) = \mathsf{Bool} & \varepsilon(\Gamma, x : A) = \varepsilon(\Gamma), x : \varepsilon(A) & \varepsilon(\mathsf{app}(t, u)) = \mathsf{app}(\varepsilon(t), \varepsilon(u)) \\ \varepsilon(A \Rightarrow B) = \varepsilon(A) \Rightarrow \varepsilon(B) & & \varepsilon(\mathsf{seal}_l(t)) = \varepsilon(t) \\ \varepsilon(\mathsf{S}_l\, A) = \varepsilon(A) & & \varepsilon(\mathsf{unseal}_l(t)) = \varepsilon(t) \end{array}$$

To clarify this point further, the noninterference property enforced by an SC term $\{\texttt{public}\} \;;\, sec : \mathsf{S}_{\mathtt{secret}}\, \mathsf{Bool} \vdash^{\mathsf{sc}} t : \mathsf{Bool}$ on its underlying STLC program $sec : \mathsf{Bool} \vdash \varepsilon(f) : \mathsf{Bool}$ is that for all $\cdot \vdash s_1, s_2 : \mathsf{Bool}$ whenever both $\varepsilon(t)[s_1/sec]$ and $\varepsilon(t)[s_2/sec]$ terminate then they do so with the same Boolean. In this way, SC types and typing contexts play the role of security specifications, and SC terms of evidence that the underlying programs are secure, i.e., they satisfy the security specification. Indeed, for us SC terms do not have operational semantics, only their underlying STLC programs do. When convenient, we identify SC terms with the erased STLC programs. We observe that this interpretation of SC as an IFC enforcement mechanism is closed under the operational semantics of STLC:

Lemma 1. *Given a term $\pi \,;\, \cdot \vdash^{\mathsf{sc}} t : A$ such that $\varepsilon(t) \longrightarrow f$ then there exists a term $\pi \,;\, \cdot \vdash^{\mathsf{sc}} t' : A$ such that $\varepsilon(t') = f$.*

3 Effectful Information-Flow Control

In this section, we present the main contribution of this paper: the observation that a single primitive distr is enough to extend IFC in pure languages with effects. We study two extensions of the programming language and the IFC enforcement mechanism from Section 2 with printing and global store effects.

In these extensions, we treat effects *explicitly* in the style of Haskell's IO monad [16] and Moggi's monadic metalanguage [25]: the only programs that can perform effects are of type $\mathsf{Eff}_{\mathcal{C}}\, a$ for some effect annotation $\mathcal{C}$ and type a, and sequencing of effects is made explicit through the primitive bind. Specifically, we consider printing and global store effects as suitable representatives of the two kinds of effects that need to be secured:

Printing Effects. Printing on a channel can be observed externally to the program by the channel's observers. Observers can infer information about the program's input from what is being printed to the channel. To secure printing effects one must ensure that the decision to print and what is being printed only depends on data less sensitive than the channel's observers.

Global Store Effects. Reading from the store cannot be observed directly. However, reading effects need to be secured because what is read may influence the program's subsequent behavior. To secure reading effects one must ensure that what has been read is tracked as sensitive data.

In contrast to Haskell and the metalanguage, we employ a graded monad [18] whose effect annotation $\mathcal{C}$ tracks precisely to which channels a computation might print and which store locations a computation might access.

3.1 Printing Effects

In Figure 2 we present the extension of STLC which allows programs to perform printing effects. We dub this language $\text{STLC}^{\text{Print}}$. We assume that the set of printing channels $\mathcal{Ch}$ is fixed a priori, i.e., the channels are statically known. The set of types is extended with a new type for computations $\mathsf{Eff}_{\mathcal{C}}\, a$ that is indexed by a set of channels $\mathcal{C} \subseteq \mathcal{Ch}$. A program of type $\mathsf{Eff}_{\mathcal{C}}\, a$ when executed might *only* print to the channels that appear in $\mathcal{C}$ and return a result of type a.

The typing rules of the standard monadic operations, return and bind, are as expected: in Rule Return, the computation does not perform any effects thus the type is indexed by the empty set of channels; and, in Rule Bind, the type is indexed by the union of the channels on which the computations $\Gamma \vdash t : \mathsf{Eff}_{\mathcal{C}_1}\, a$ and $\Gamma \vdash u : a \Rightarrow \mathsf{Eff}_{\mathcal{C}_2}\, b$ might print, that is, $\mathcal{C}_1 \cup \mathcal{C}_2$. We include subeffecting—casting from a smaller to a larger set of channels—as the term subeff in the language, see Rule Subeff. Printing is performed via a family of primitive operations, print_{ch}, one for each available channel $ch \in \mathcal{Ch}$ (Rule Print). We assume, for simplicity, that only Boolean values can be printed. Further, observe that the resulting monadic type is indexed by the singleton set that only contains the channel on which the printing is performed, i.e., $\mathsf{Eff}_{\{ch\}}\, \mathsf{Unit}$.

The operational semantics of $\text{STLC}^{\text{Print}}$ is defined as the combination of the small-step operational semantics of STLC—see Appendix A—and the small-step operational semantics of computations defined in Figure 2. The semantics of effect-free terms, inherited from STLC, $t \longrightarrow u$, treats computations, such as $\mathsf{return}(t)$, as values even when their subterms are not values, e.g., $\mathsf{return}(\mathsf{app}(\lambda(x.\, x), \mathsf{true})) \not\longrightarrow$. The semantics of computations (alternatively, monadic semantics) is of the form $t \rightsquigarrow u, o$ and is interpreted as follows: program $\cdot \vdash t : \mathsf{Eff}_{\mathcal{C}}\, a$ evaluates in one step to program $\cdot \vdash u : \mathsf{Eff}_{\mathcal{C}}\, a$ and produces

output o. The output is a function from channels to lists of Boolean values $o \in \mathcal{Ch} \to \mathrm{List}\,\mathsf{Bool}$, and it represents the outputs of the program during execution.

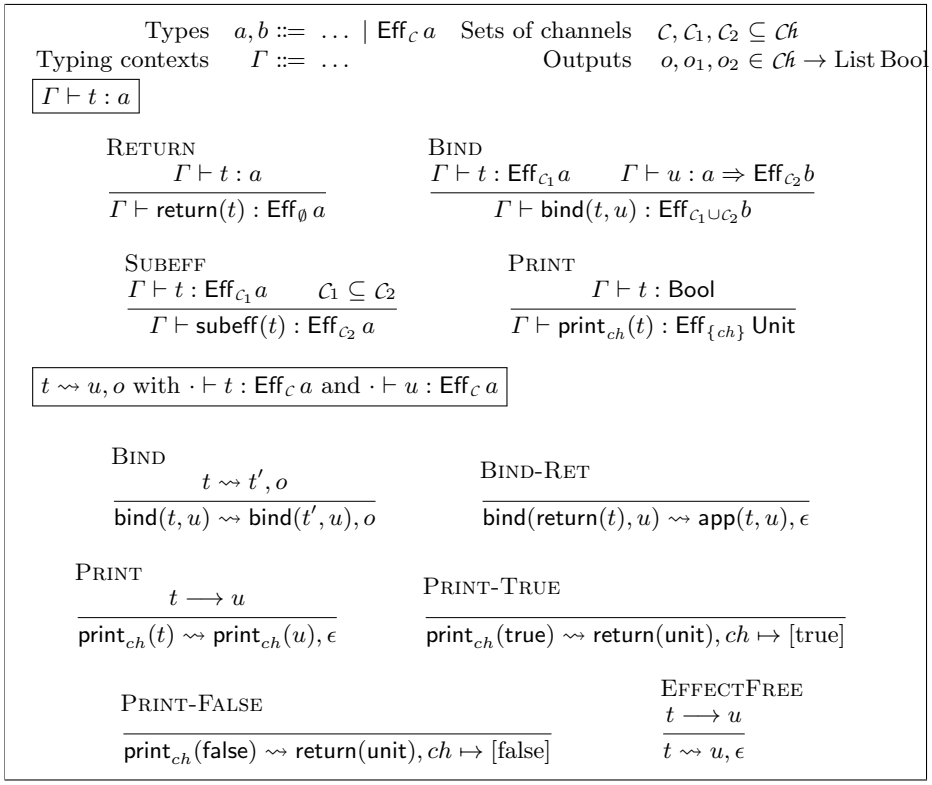

Fig. 2. Types, well-typed terms and small-step semantics of STLC$^{\textsc{Print}}$ (excerpts)

We now briefly explain the semantics. Rule Bind reduces the left subterm of bind and executes its effects, o. Once the left subterm is a value, i.e., $\mathsf{return}(t)$, Bind-Ret applies the rest of the computation u to the result t. Applying the continuation u does not produce effects—recall that we are in a pure language—thus the step contains the empty output on the right, i.e., ϵ. The empty output maps every channel to the empty output, i.e., $\epsilon := \lambda ch.[]$. Rule Print reduces the argument t of $\mathsf{print}_{ch}(t)$ until it is a value of type Bool, either true or false, and then Rules Print-True and Print-False print the corresponding Boolean on the output channel ch. The output $ch \mapsto [v]$ is the function that maps the channel ch to the singleton list $[v]$ and every other channel to the empty list. Observe that these rules make $\mathsf{print}_{ch}(t)$ strict in its argument. Rule EffectFree serves to lift effect-free reductions to the level of computations. Since by definition

effect-free reductions do not produce effects, the right hand side of the effect contains the empty output ϵ. To complete the picture, we denote by $t \rightsquigarrow^* u, o$ the reflexive-transitive closure of the monadic reduction relation. To combine effects, we lift the monoid structure on List Bool to outputs. The rules that handle the term subeff (omitted) are straightforward.

Two reduction relations. While it might seem unnecessary to define the semantics using the combination of a small-step relation of effect-free programs and a small-step relation of computations, it is a natural form of expressing the operational semantics of pure languages with effects [45]. The effect-free relation evaluates programs that *cannot* perform effects, whilst the relation for computations evaluates programs which can, and computes those effects.

To conclude the presentation of STLC$^{\text{Print}}$, we enunciate the following lemma which states that the index $\mathcal{C}$ in the type of computations $\mathsf{Eff}_{\mathcal{C}}\, a$ is a sound approximation of the set of channels where a program $\cdot \vdash t : \mathsf{Eff}_{\mathcal{C}}\, a$ may print, i.e., t does not produce output in any channel not in $\mathcal{C}$. In the security literature it is usually called the *confinement lemma*:

Lemma 2 (Confinement for STLC$^{\text{Print}}$). *For any* STLC$^{\text{Print}}$ *program of type* $\cdot \vdash f : \mathsf{Eff}_{\mathcal{C}}\, a$, STLC$^{\text{Print}}$ *value* $\cdot \vdash v : \mathsf{Eff}_{\mathcal{C}}\, a$, *and output* o, *if* $t \rightsquigarrow^* v, o$ *then* $\forall ch \in \mathcal{Ch}.\, ch \notin \mathcal{C} \Rightarrow o(ch) = []$.

Types $A, B ::= \ldots \mid \mathsf{Eff}_{\mathcal{C}}\, A$ Sets of channels $\mathcal{C}, \mathcal{C}_1, \mathcal{C}_2 \subseteq \mathcal{Ch}$

Typing contexts $\Gamma ::= \ldots$ Sensitivity of channels $\text{label} \in \mathcal{Ch} \to \mathcal{L}$

$\boxed{\pi \;;\; \Gamma \vdash^{\mathsf{sc}} t : A}$

RETURN
$$\frac{\pi \;;\; \Gamma \vdash^{\mathsf{sc}} t : A}{\pi \;;\; \Gamma \vdash^{\mathsf{sc}} \mathsf{return}(t) : \mathsf{Eff}_{\emptyset}\, A}$$

BIND
$$\frac{\pi \;;\; \Gamma \vdash^{\mathsf{sc}} t : \mathsf{Eff}_{\mathcal{C}_1}\, A \qquad \pi \;;\; \Gamma \vdash^{\mathsf{sc}} u : A \Rightarrow \mathsf{Eff}_{\mathcal{C}_2}\, B}{\pi \;;\; \Gamma \vdash^{\mathsf{sc}} \mathsf{bind}(t, u) : \mathsf{Eff}_{\mathcal{C}_1 \cup \mathcal{C}_2}\, B}$$

SUBEFF
$$\frac{\pi \;;\; \Gamma \vdash^{\mathsf{sc}} t : \mathsf{Eff}_{\mathcal{C}_1}\, A \qquad \mathcal{C}_1 \subseteq \mathcal{C}_2}{\pi \;;\; \Gamma \vdash^{\mathsf{sc}} \mathsf{subeff}(t) : \mathsf{Eff}_{\mathcal{C}_2}\, A}$$

PRINT
$$\frac{\pi \;;\; \Gamma \vdash^{\mathsf{sc}} t : \mathsf{Bool}}{\pi \;;\; \Gamma \vdash^{\mathsf{sc}} \mathsf{print}_{ch}(t) : \mathsf{Eff}_{\{ch\}}\, \mathsf{Unit}}$$

DISTR
$$\frac{\pi \;;\; \Gamma \vdash^{\mathsf{sc}} t : \mathsf{S}_l\, (\mathsf{Eff}_{\mathcal{C}}\, A) \qquad \forall ch \in \mathcal{C}.\; l \sqsubseteq \text{label}(ch)}{\pi \;;\; \Gamma \vdash^{\mathsf{sc}} \mathsf{distr}(t) : \mathsf{Eff}_{\mathcal{C}}\, (\mathsf{S}_l\, A)}$$

Fig. 3. Types and intrinsically-typed terms of SC$^{\text{Print}}$

After having defined the programming language, we are in position to turn our attention to enforcing IFC. We assume that the security policy specifies the sensitivity of each printing channel, i.e., the greatest lower bound of the sensitivities of all its observers, in the form of a function $\text{label} \in \mathcal{Ch} \to \mathcal{L}$. In

Figure 3 we present the extension of SC (Figure 1) that accommodates printing effects. We name it SC^{Print} hereafter. The types reflect those in SC and include the new type constructor $\mathsf{Eff}_{\mathcal{C}}$ of computations. The typing rules for the $\text{STLC}^{\text{Print}}$ fragment of SC^{Print}, i.e., Rules Return to Print, simply propagate the set of labels π to their premises. Observe, again, that one has to explicitly unseal sensitive Booleans, i.e., of type $\mathsf{S}_l\,\mathsf{Bool}$, to apply the Rule Print. Before detailing Rule Distr, we extend the erasure function ε from SC^{Print}-terms to $\text{STLC}^{\text{Print}}$ programs in the obvious way: i.e., the type former $\mathsf{Eff}_{\mathcal{C}}$ erases to "itself" and, analogously to seal_l and unseal_l, distr is a no-op.

$$\begin{array}{ll} \varepsilon(\dots) = \dots & \varepsilon(\dots) = \dots \\ \varepsilon(\mathsf{Eff}_{\mathcal{C}}\,A) = \mathsf{Eff}_{\mathcal{C}}\,\varepsilon(A) & \varepsilon(\mathsf{distr}(t)) = \varepsilon(t) \end{array}$$

Rule Distr introduces one of the novelties of our work; an enforcement mechanism that selectively permits to execute the effects of computations that depend on sensitive data. In SC^{Print}, a term of type $\mathsf{S}_l\,(\mathsf{Eff}_{\mathcal{C}}\,A)$ describes a computation that might *only* print on the channels in $\mathcal{C}$ and what is printed and the decision to print potentially depends on data of sensitivity l. Then, it is natural to ask, when it is secure to execute the effects of the inner computation of type $\mathsf{Eff}_{\mathcal{C}}\,A$? Clearly, whenever the sensitivities of the computation's effects, i.e., the channels in $\mathcal{C}$, are as high as the sensitivity of the data used to decide to perform the effects, i.e., sensitivity l. The premise of the rule exactly captures this condition: $\forall ch \in \mathcal{C}.\ l \sqsubseteq \text{label}(ch)$.

To illustrate Distr in action, let us consider the following term in SC^{Print} that prints `Alice`'s sensitive input to the $\mathtt{Ch}_{\mathtt{secret}}$ channel:

$$\begin{array}{l} \mathit{prog}_6 : \mathsf{S}_{\mathtt{Alice}}\,\mathsf{Bool} \Rightarrow \mathsf{Eff}_{\{\mathtt{Ch}_{\mathtt{secret}}\}}\,\mathsf{Unit} \\ \mathit{prog}_6 = \lambda(sb.\,\mathsf{distr}(\mathsf{seal}_{\mathtt{Alice}}(\mathsf{print}_{\mathtt{Ch}_{\mathtt{secret}}}(\mathsf{unseal}_{\mathtt{Alice}}(sb)))) \gg\!= \mathsf{return}(\mathsf{unit})) \end{array}$$

Inside the term λ, the primitive distr permits to execute the effects of the term $\mathsf{seal}_{\mathtt{Alice}}(\mathsf{print}_{\mathtt{Ch}_{\mathtt{secret}}}(\mathsf{unseal}_{\mathtt{Alice}}(sb)))$ of type $\mathsf{S}_{\mathtt{Alice}}\,(\mathsf{Eff}_{\{\mathtt{Ch}_{\mathtt{secret}}\}}\,\mathsf{Bool})$. The term distr protects the return type of the computation at the same sensitivity as the premise's type $\mathsf{S}_l\,A$. This requirement is necessary to enforce IFC because the trivial computation that performs no effects and just returns has access to sensitive data, as exemplified by the following program:

$$\begin{array}{l} \mathit{prog}_7 : \mathsf{S}_{\mathtt{Alice}}\,\mathsf{Bool} \Rightarrow \mathsf{S}_{\mathtt{Bob}}\,\mathsf{Bool} \Rightarrow \mathsf{Eff}_{\{\mathtt{Ch}_{\mathtt{secret}}\}}\,(\mathsf{S}_{\mathtt{secret}}\,\mathsf{Bool}) \\ \mathit{prog}_7 = \lambda(sb_1\,sb_2.\,\mathsf{distr}(\mathsf{seal}_{\mathtt{secret}}(\mathsf{print}_{\mathtt{Ch}_{\mathtt{secret}}}\,(\mathsf{unseal}_{\mathtt{Alice}}(sb_1))) \\ \qquad\qquad\qquad\qquad \gg \mathsf{return}(\mathsf{unseal}_{\mathtt{Bob}}(sb_2)))) \end{array}$$

3.2 Global Store Effects

We turn our attention to global store effects which combines printing effects from the previous section with reading from locations in the store. Following the same steps, in Figure 4 we present the extension of STLC (Appendix A) in which programs have access to a global store and can read from and write to it. We name this language $\text{STLC}^{\text{Store}}$. Our development rests on two assumptions: 1. the set of locations in the store $\mathcal{Loc}$ is fixed during execution, and 2. only terms of ground type, i.e., Bool and Unit, can be stored. This helps simplify our presentation and permit us to side-step issues which are orthogonal to the point

of this work. Recent work by Rajani and Garg [32] and Gregersen et al. [12] hints to possible ways of lifting these two simplifications.

STLC$^{\textsc{Store}}$ extends STLC with a type of computations $\mathsf{Eff}_{\mathcal{S}}\, a$, indexed by a set of store locations $\mathcal{S}$ (alternatively, references), and a type of references $\mathsf{Ref}_s\, r$. By assumption 1, the whole development is parameterized by a fixed set of store locations $\mathcal{Loc}$. A program of type $\mathsf{Eff}_{\mathcal{S}}\, a$ when executed might *only* write to the references in $\mathcal{S}$ and finally return a result of type a. A term of type $\mathsf{Ref}_s\, r$ is a reference in the store s that contains terms of ground type r. References permit both "printing" effects via writing—like channels in STLC$^{\textsc{Print}}$—but also reading effects. Note that the type of computations *does not* mention the store locations from which it might read. This asymmetry stems from how the execution of programs performing these effects interact with their environment: writing alters it whilst reading does not—at least directly.

Typing judgements are of the form $\Sigma, \Gamma \vdash t : a$ where Σ is a store typing which determines the shape of the store, i.e., which types it contains and in what locations. The rest of components are like those in STLC. The typing rules of the monadic operations return, bind and subeff follow the same pattern as in STLC$^{\textsc{Print}}$ (Figure 2), thus we do not discuss them any further. The term ref_s (Rule Ref) is the runtime representation of references. Reading and writing is achieved via primitives read and write (Rules Read and Write respectively). Observe that in Read, the type of the computation $\mathsf{Eff}_{\emptyset}\, r$ in the conclusion of the rule is indexed by the empty set of locations, while in Write is indexed by the singleton set $\{s\}$.

The operational semantics of the language is defined as in STLC$^{\textsc{Print}}$; a combination of the small-step semantics for STLC that treats effectful primitives as values, and a small-step semantics for computations. Given a store typing Σ, a store θ is a function from locations to typed terms according to Σ. Since the language considers a fixed-size store, we use the notation θ instead of $\theta(\Sigma)$. The semantics of computations is of the form $\theta_1, t \rightsquigarrow \theta_2, u$, and is interpreted as: program $\Sigma, \cdot \vdash t : \mathsf{Eff}_{\mathcal{S}}\, a$ paired with store θ_1 evaluates in one step to program $\Sigma, \cdot \vdash u : \mathsf{Eff}_{\mathcal{S}}\, a$ and store θ_2.

We now explain the semantics. Rule Read reduces the argument of read and it does not modify the store. When the argument is a store location, i.e., ref_s, Read-Ref retrieves the term t from the store, i.e., $\theta(s) = t$. The rules for writing Rules Write and Write-Ref first reduce the left subterm of $\mathsf{write}(u, t)$ to a store location and then write the right subterm t on the store. Different from STLC$^{\textsc{Print}}$, we permit to write any term on the store, not only values.

To briefly illustrate STLC$^{\textsc{Store}}$, consider the following program:

$$
\begin{aligned}
&\mathit{prog}_8 : \mathsf{Bool} \Rightarrow \mathsf{Eff}_{\{s'\}}\, \mathsf{Unit} \\
&\mathit{prog}_8\, b = \mathsf{if}\ b\ \mathsf{then}\ (\mathsf{read}(s) \ggg\!= \lambda(x.\, \mathsf{write}(s', x))) \\
&\qquad\qquad\quad \mathsf{else}\ \mathsf{return}(\mathsf{unit})
\end{aligned}
$$

Based on the Boolean input, prog_8 copies the contents of the store location s to s'. Observe that the type only mentions the location s' in its index.

We conclude the explanation of STLC$^{\textsc{Store}}$ with a confinement lemma—similar to that of STLC$^{\textsc{Print}}$ (Lemma 2):

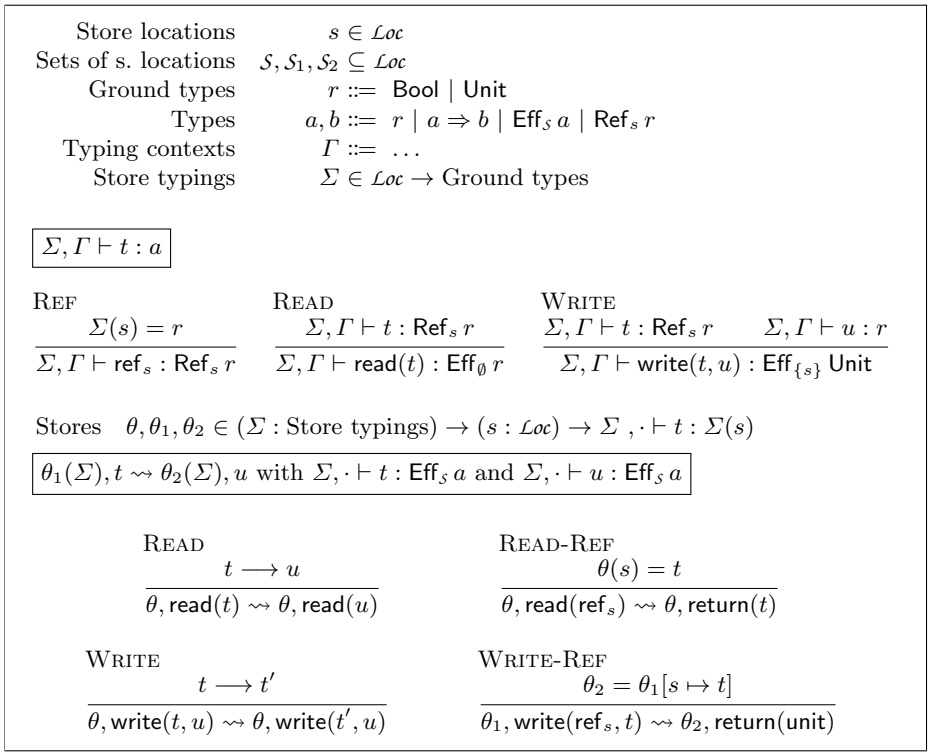

Fig. 4. Types, well-typed terms and small-step semantics of STLC$^{\text{Store}}$ (excerpts)

Lemma 3 (Confinement for STLC$^{\text{Store}}$). *For any* STLC$^{\text{Store}}$ *program* $\Sigma, \cdot \vdash f : \mathsf{Eff}_{\mathcal{S}}\, a$, STLC$^{\text{Store}}$ *value* $\Sigma, \cdot \vdash v : \mathsf{Eff}_{\mathcal{S}}\, a$, *and stores* $\theta_2, \theta_2 : \Sigma$, *if* $\theta_1, f \rightsquigarrow^* \theta_2, v$ *then* $\forall s \in \mathcal{Loc}.\, s \notin \mathcal{S} \Rightarrow \theta_1(s) = \theta_2(s)$.

Now its turn to explain the IFC enforcement mechanism SC$^{\text{Store}}$ (Figure 5). As in SC$^{\text{Print}}$ (Figure 3) we assume that the security policy specifies for each store location its sensitivity as a function label $\in \mathcal{Loc} \to \mathcal{L}$. The typing rules for the monadic primitives are analogous to SC$^{\text{Print}}$ thus we have omitted them. The rule for references is straightforward (Rule Ref). More interesting is the typing rule for reading from the store (Rule Read). In the conclusion of the rule, the return type of the computation is the SC type for sensitive data $\mathsf{S}_l\, R$. The sensitivity of the location is l, i.e., label$(s) = l$ in the premise of the rule, thus to protect the flow of information is necessary to wrap also the return type.

The type of read diverges from usual presentations of IFC libraries (e.g., LIO [39], MAC [33] with the exception of SLIO [32]) in that reading from the store is wrapped in the type S_l. These libraries incorporate the data into their monad of computations, which keeps track of the sensitivities of the observed values.

We conclude the section with a concrete example of SC$^{\text{Store}}$ that shows that $prog_8$ is secure with respect to the following specification: label$(s) =$ `Alice`,

$$
\begin{array}{rl}
\text{Ground types} & R ::= \mathsf{Bool} \mid \mathsf{Unit} \\
\text{Types} & A, B ::= R \mid A \Rightarrow B \mid \mathsf{Eff}_s\, A \mid \mathsf{Ref}_s\, R \\
\text{Store typings} & \Sigma \in \mathcal{L}oc \to \text{Ground types} \\
\text{Sensitivity of loc.} & \mathrm{label} \in \mathcal{L}oc \to \mathcal{L}
\end{array}
$$

$$\boxed{\pi \,;\, \Sigma, \Gamma \vdash^{\mathsf{sc}} t : A}$$

$$\textsc{Ref}\quad \frac{\Sigma(s) = R}{\pi \,;\, \Sigma, \Gamma \vdash^{\mathsf{sc}} \mathsf{ref}_s : \mathsf{Ref}_s\, R} \qquad \textsc{Read}\quad \frac{\pi \,;\, \Sigma, \Gamma \vdash^{\mathsf{sc}} t : \mathsf{Ref}_s\, R \qquad \mathrm{label}(s) = l}{\pi \,;\, \Sigma, \Gamma \vdash^{\mathsf{sc}} \mathsf{read}(t) : \mathsf{Eff}_\emptyset\, (\mathsf{S}_l\, R)}$$

$$\textsc{Write}\quad \frac{\pi \,;\, \Sigma, \Gamma \vdash^{\mathsf{sc}} t : \mathsf{Ref}_s\, R \qquad \pi \,;\, \Sigma, \Gamma \vdash^{\mathsf{sc}} u : R}{\pi \,;\, \Sigma, \Gamma \vdash^{\mathsf{sc}} \mathsf{write}(t, u) : \mathsf{Eff}_{\{s\}}\, \mathsf{Unit}}$$

$$\textsc{Distr}\quad \frac{\pi \,;\, \Sigma, \Gamma \vdash^{\mathsf{sc}} t : \mathsf{S}_l\, (\mathsf{Eff}_s\, A) \qquad \forall s \in \mathcal{S}.\ l \sqsubseteq \mathrm{label}(s)}{\pi \,;\, \Gamma \vdash^{\mathsf{sc}} \mathsf{distr}(t) : \mathsf{Eff}_s\, (\mathsf{S}_l\, A)}$$

Fig. 5. Types and intrinsically-typed terms of $\mathrm{SC}^{\textsc{Store}}$ (excerpts)

$\mathrm{label}(s') = \mathtt{secret}$ and the Boolean argument is $\mathtt{secret}$, i.e., $sb : \mathsf{S}_{\mathtt{secret}}\, \mathsf{Bool}$:

$$
\begin{array}{l}
prog'_8 : \mathsf{S}_{\mathtt{secret}}\, \mathsf{Bool} \Rightarrow \mathsf{Eff}_{\{s'\}}\, (\mathsf{S}_{\mathtt{secret}}\, \mathsf{Unit}) \\
prog'_8\ sb = \mathsf{distr}(\mathsf{seal}_{\mathtt{secret}}(\mathsf{if}\ \mathsf{unseal}_{\mathtt{Alice}}(sb) \\
\qquad\qquad\qquad\qquad \mathsf{then}\ (\mathsf{read}(s) \gg\!= \lambda(x.\, \mathsf{write}(s', \mathsf{unseal}_{\mathtt{secret}}(x)))) \\
\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{return}(\mathsf{unit})))
\end{array}
$$

The above program exemplifies how $\mathrm{SC}^{\textsc{Store}}$ enforces that flows from the store to the program and back to the store are secure. Note that $\varepsilon(prog'_8) = prog_8$.

4 Security Guarantees

In this section, we prove that the IFC mechanisms SC, $\mathrm{SC}^{\textsc{Print}}$ and $\mathrm{SC}^{\textsc{Store}}$ can be used to enforce noninterference for the programming languages STLC, $\mathrm{STLC}^{\textsc{Print}}$ and $\mathrm{STLC}^{\textsc{Store}}$ presented in Sections 2, 3.1 and 3.2, respectively. Our noninterference proofs employ the technique of logical relations (LR) [23]. For each language, we construct a family of LRs parameterized by the sensitivity of the attacker $l_{\mathtt{atk}}$ and the SC types. In the effect-free setting, the LR interprets each SC type A as a binary relation over STLC programs of the erased type $\varepsilon(A)$. The relation captures the idea of indistinguishable programs: if the type is *public* enough, e.g., $\mathsf{S}_{\mathtt{public}}\, \mathsf{Bool}$ and $\mathtt{public}$ flows to $l_{\mathtt{atk}}$, then two programs are related when they evaluate to the same value. Noninterference then follows as a corollary of the so called fundamental theorem of the LR.

Although the languages we consider do not have primitives for writing general recursive programs, the noninterference properties that we prove, namely termination-insensitive noninterference (TINI), do not assume that programs terminate. A non-terminating loop in a language with printing effects can leak secrets, however, the leakage bandwidth is exponential in the size of the secret [6],

and hence, most IFC tools (e.g., Lifty [31]) ignore such leaks and enforce TINI. From an IFC perspective, it is straightforward to consider general recursion (e.g., as done in Russo, Claessen, and Hughes [34] and Schoepe, Hedin, and Sabelfeld [36]).

TINI states that when two runs of a program with different `secret` inputs terminate, then its `public` outputs agree. In the effect-free setting, the inputs to a program are its arguments, and the output is its return value. In the effectful setting, what we need to consider as inputs and outputs changes: in STLC$^{\textsc{Store}}$, for instance, the store must be considered an additional input to the program.

Definition 1 (TINI for STLC). *An* STLC *program* $sec : \mathsf{Bool} \vdash f : \mathsf{Bool}$ *satisfies TINI if for any two* STLC *terms* $\cdot \vdash s_1, s_2 : \mathsf{Bool}$*, and any two* STLC *values* $\cdot \vdash v_1, v_2 : \mathsf{Bool}$*, if* $f[s_1/sec] \longrightarrow^* v_1$ *and* $f[s_2/sec] \longrightarrow^* v_2$ *then* $v_1 \equiv v_2$*.*

In the definition $v_1 \equiv v_2$ denotes that v_1 and v_2 are syntactically equal values. Note that programs that diverge for any input vacuously satisfy TINI: the assumption that the substituted programs terminate will never hold.

SC can be used to enforce TINI:

Theorem 1. *Given any* SC *term* $\{l_{\mathtt{atk}}\} ; sec : \mathsf{S}_{l_{\mathtt{sec}}}\ \mathsf{Bool} \vdash^{\mathsf{sc}} t : \mathsf{S}_{l_{\mathtt{atk}}}\ \mathsf{Bool}$ *where* $l_{\mathtt{sec}} \not\sqsubseteq l_{\mathtt{atk}}$*, the erased program* $sec : \mathsf{Bool} \vdash \varepsilon(t) : \mathsf{Bool}$ *satisfies TINI.*

In practice, we are concerned that information does not flow from $l_{\mathtt{sec}}$-protected data to the attacker with sensitivity $l_{\mathtt{atk}}$. Thus, to show that an STLC program $sec : \mathsf{Bool} \vdash f : \mathsf{Bool}$ does not leak information from $l_{\mathtt{sec}}$ to $l_{\mathtt{atk}}$, it is enough to find an SC term $\{l_{\mathtt{atk}}\} ; sec : \mathsf{S}_{l_{\mathtt{sec}}}\ \mathsf{Bool} \vdash^{\mathsf{sc}} t : \mathsf{S}_{l_{\mathtt{atk}}}\ \mathsf{Bool}$ such that $f = \varepsilon(t)$.

Logical relation. In order to prove that SC enforces TINI (Theorem 1), we construct an LR parameterized by the attacker's sensitivity $l_{\mathtt{atk}}$ and the SC types—see Figure 6. The proof, as we will show, then falls out as a consequence of the fundamental theorem of the LR.

At each SC type the LR defines what the attacker can observe about pairs of STLC programs of erased type—alternatively, the same program with different secrets. We split the definition depending on whether programs are evaluated, $\mathcal{R}_{\mathcal{V}}^{l_{\mathtt{atk}}}[\![-]\!]$ and $\mathcal{R}_{\mathcal{E}}^{l_{\mathtt{atk}}}[\![-]\!]$ respectively. We briefly explain these. At SC type Bool, for instance, see $\mathcal{R}_{\mathcal{E}}^{l_{\mathtt{atk}}}[\![\mathsf{Bool}]\!]$ and $\mathcal{R}_{\mathcal{V}}^{l_{\mathtt{atk}}}[\![\mathsf{Bool}]\!]$, the LR states that if the programs terminate then the attacker can observe if they return equal values. At higher types, i.e., $A \Rightarrow B$, two functions are related if whenever they reduce to a value, see $\mathcal{R}_{\mathcal{E}}^{l_{\mathtt{atk}}}[\![A \Rightarrow B]\!]$, they map related inputs $\mathcal{R}_{\mathcal{E}}^{l_{\mathtt{atk}}}[\![A]\!](u_1, u_2)$ to related outputs $\mathcal{R}_{\mathcal{E}}^{l_{\mathtt{atk}}}[\![A]\!](\mathsf{app}(t_1, u_1), \mathsf{app}(t_2, u_2))$, see $\mathcal{R}_{\mathcal{V}}^{l_{\mathtt{atk}}}[\![A \Rightarrow B]\!]$. Lastly, at SC type $\mathsf{S}_l\ A$, see $\mathcal{R}_{\mathcal{V}}^{l_{\mathtt{atk}}}[\![\mathsf{S}_l\ A]\!]$, the LR compares the sensitivity of the attacker with the label l, and in case it is less sensitive, i.e., $l \sqsubseteq l_{\mathtt{atk}}$, the programs have to be related at SC type A. If the label l is more sensitive than the attacker's label, i.e., $l \not\sqsubseteq l_{\mathtt{atk}}$ then the programs do not need to be related.

Definitions $\mathcal{R}_{\mathcal{V}}^{l_{\mathtt{atk}}}[\![-]\!]$ and $\mathcal{R}_{\mathcal{E}}^{l_{\mathtt{atk}}}[\![-]\!]$ work on closed terms, however, in order to prove the fundamental theorem we have to lift them to substitutions. A substitution assigns to each type a in a typing context Γ a closed term of

that type, i.e., $\cdot \vdash f : a$. We denote substitutions by γ and use $\gamma : \cdot \vdash \Gamma$ to mean that γ is in the set of substitutions over Γ. We define the LR for substitutions, $\mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{S}}[\![-]\!]$, by induction on SC typing contexts. At the empty context $\cdot$ the empty substitutions (ϵ, ϵ) are trivially related—denoted by $\top$. Two non-empty substitutions (γ_1, t_1) and (γ_2, t_2) are related whenever they are pointwise related, i.e., $\mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{E}}[\![A]\!](t_1, t_2)$ and $\mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{S}}[\![\Gamma]\!](\gamma_1, \gamma_2)$. The LR for open terms, written $\mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{T}}[\![(\Gamma, A)]\!]$, is indexed by a pair consisting of an SC typing context Γ and an SC type A. It states that two STLC terms are related if for any two related substitutions the substituted terms are related at type A.

$$\mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{V}}[\![-]\!] \in (A : \text{SC type}) \to (t_1 : \cdot \vdash \varepsilon(A)) \to (t_2 : \cdot \vdash \varepsilon(A)) \to \text{Set}$$

$$\mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{V}}[\![\mathsf{Unit}]\!](t_1, t_2) :\Leftrightarrow t_1 \equiv t_2$$

$$\mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{V}}[\![\mathsf{Bool}]\!](t_1, t_2) :\Leftrightarrow t_1 \equiv t_2$$

$$\mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{V}}[\![A \Rightarrow B]\!](t_1, t_2) :\Leftrightarrow \forall (u_1, u_2 : \cdot \vdash \varepsilon(A)).\ \mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{E}}[\![A]\!](u_1, u_2) \Rightarrow \mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{E}}[\![B]\!](\mathsf{app}(t_1, u_1), \mathsf{app}(t_2, u_2))$$

$$\mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{V}}[\![\mathsf{S}_l\ A]\!](t_1, t_2) :\Leftrightarrow l \sqsubseteq l_{\mathsf{atk}} \Rightarrow \mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{V}}[\![A]\!](t_1, t_2)$$

$$\mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{E}}[\![A]\!](t_1, t_2) :\Leftrightarrow \forall (u_1, u_2 : \cdot \vdash \varepsilon(A)).\ t_1 \longrightarrow^* u_1 \wedge t_2 \longrightarrow^* u_2 \Rightarrow \mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{V}}[\![A]\!](u_1, u_2)$$

$$\mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{S}}[\![\cdot]\!](\epsilon, \epsilon) :\Leftrightarrow \top$$

$$\mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{S}}[\![\Gamma, x : A]\!]((\gamma_1, t_1), (\gamma_2, t_2)) :\Leftrightarrow \mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{E}}[\![A]\!](t_1, t_2) \wedge \mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{S}}[\![\Gamma]\!](\gamma_1, \gamma_2)$$

$$\mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{T}}[\![(\Gamma, A)]\!](t_1, t_2) :\Leftrightarrow \forall (\gamma_1, \gamma_2 : \cdot \vdash \varepsilon(\Gamma)).\ \mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{S}}[\![\Gamma]\!](\gamma_1, \gamma_2) \Rightarrow \mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{E}}[\![A]\!](t_1[\gamma_1], t_2[\gamma_2])$$

Fig. 6. Logical relation for STLC-SC

The fundamental theorem of the LR states that the underlying program of an SC term is related to itself. Formally:

Theorem 2 (Fundamental Theorem of the LR for STLC). *For any attacker with sensitivity l_{atk}, and* SC *term $\{l_{\mathsf{atk}}\}\ ; \Gamma \vdash^{\mathsf{sc}} t : A$, it is the case that $\mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{T}}[\![(\Gamma, A)]\!](\varepsilon(t), \varepsilon(t))$.*

Proof. By induction on the typing derivation.

TINI (Theorem 1) follows as a corollary of the fundamental theorem:

Proof. Assume two secrets $\cdot \vdash s_1, s_2 : \mathsf{Bool}$. Since $l_{\mathsf{sec}} \not\sqsubseteq l_{\mathsf{atk}}$, the secrets are related, i.e., $\mathcal{R}^{l_{\mathsf{atk}}}_{\mathcal{E}}[\![\mathsf{S}_{l_{\mathsf{sec}}}\ \mathsf{Bool}]\!](s_1, s_2)$, and thus the two substitutions $\gamma_1 = \{sec \mapsto$

$s_1\}$ and $\gamma_2 = \{sec \mapsto s_2\}$ are related. By the fundamental theorem, the term $\varepsilon(t)$ is related to itself, i.e., $\mathcal{R}^{l_{\mathtt{atk}}}_{\mathcal{E}}[\![\mathsf{S}_{l_{\mathtt{atk}}}\,\mathsf{Bool}]\!](\varepsilon(t)[\gamma_1], \varepsilon(t)[\gamma_2])$. Unfolding the definitions of we obtain that $\forall(v_1, v_2 : \cdot \vdash \mathsf{Bool}).\, \varepsilon(t)[\gamma_1] \longrightarrow^* v_1 \wedge \varepsilon(t)[\gamma_2] \longrightarrow^* v_2 \Rightarrow v_1 \equiv v_2$.

4.1 Noninterference for Printing Effects: STLC$^{\mathbf{Print}}$ and SC$^{\mathbf{Print}}$

It order to formalize noninterference for STLC$^{\textsc{Print}}$ we look at effectful programs that might print. For that, we first define indistinguishability of outputs with respect to a subset of channels:

Definition 2 (Output indistinguishability). *Let $\mathcal{C} \subseteq \mathcal{Ch}$. Two outputs o_1 and o_2 are $\mathcal{C}$-indistinguishable, denoted by $o_1 =_{\mathcal{C}} o_2$, if the two outputs agree in $\mathcal{C}$, i.e., $\forall ch \in \mathcal{C}.\, o_1(ch) = o_2(ch)$.*

We define TINI for STLC$^{\textsc{Print}}$ for programs from Bool to $\mathsf{Eff}_{\mathcal{C}}\,\mathsf{Unit}$, i.e., programs that depending on a Boolean produce output in an arbitrary set of channels $\mathcal{C}$:

Definition 3 (TINI for STLC$^{\mathbf{Print}}$). *An* STLC$^{\textsc{Print}}$ *program* $sec : \mathsf{Bool} \vdash f : \mathsf{Eff}_{\mathcal{C}}\,\mathsf{Unit}$ *satisfies TINI* with respect to $\mathcal{C}' \subseteq \mathcal{C}$, *if for any two* STLC$^{\textsc{Print}}$ *terms* $\cdot \vdash s_1, s_2 : \mathsf{Bool}$, *any two* STLC$^{\textsc{Print}}$ *values* $\cdot \vdash v_1, v_2 : \mathsf{Eff}_{\mathcal{C}}\,\mathsf{Unit}$, *and any two outputs o_1 and o_2, if $f[s_1/sec] \rightsquigarrow^* v_1, o_1$ and $f[s_2/sec] \rightsquigarrow^* v_2, o_2$ then $o_1 =_{\mathcal{C}'} o_2$.*

TINI is parameterized by a subset of the channels where the the outputs have to agree. We will instantiate $\mathcal{C}'$ with the set of channels observable by the attacker.

SC$^{\textsc{Print}}$ can be used to enforce TINI on STLC$^{\textsc{Print}}$ programs:

Theorem 3. *Given any* SC$^{\textsc{Print}}$ *term* $\{l_{\mathtt{atk}}\}$; $sec : \mathsf{S}_{l_{\mathtt{sec}}}\,\mathsf{Bool} \vdash^{\mathsf{sc}} t : \mathsf{Eff}_{\mathcal{C}}\,\mathsf{Unit}$ *where $l_{\mathtt{sec}} \not\sqsubseteq l_{\mathtt{atk}}$ the erased program $sec : \mathsf{Bool} \vdash \varepsilon(t) : \mathsf{Eff}_{\mathcal{C}}\,\mathsf{Unit}$ satisfies TINI* with respect to $\mathcal{C}_{\mathtt{atk}}$ *where* $\mathcal{C}_{\mathtt{atk}} := \{ch \mid ch \in \mathcal{C}, \mathrm{label}(ch) \sqsubseteq l_{\mathtt{atk}}\}$.

$$\mathcal{R}^{l_{\mathtt{atk}}}_{\mathcal{V}}[\![\ldots]\!](t_1, t_2) :\Leftrightarrow \ldots$$
$$\mathcal{R}^{l_{\mathtt{atk}}}_{\mathcal{V}}[\![\mathsf{Eff}_{\mathcal{S}}\,A]\!](t_1, t_2) :\Leftrightarrow \forall(u_1, u_2 : \cdot \vdash \varepsilon(A)), o_1, o_2.\, t_1 \rightsquigarrow^* \mathsf{return}(u_1), o_1 \wedge t_2 \rightsquigarrow^* \mathsf{return}(u_2), o_2 \Rightarrow \mathcal{R}^{l_{\mathtt{atk}}}_{\mathcal{E}}[\![A]\!](u_1, u_2) \wedge o_1 =_{\mathcal{C}_{\mathtt{atk}}} o_2$$

Fig. 7. Logical relation for STLC$^{\textsc{Print}}$ (excerpts)

Logical relation. The LR for STLC$^{\textsc{Print}}$ (Figure 7) is very similar to that of STLC (Figure 6) so we skip over the commonalities and directly discuss its definition at the type of computations. The LR relates two computations $\mathcal{R}^{l_{\mathtt{atk}}}_{\mathcal{V}}[\![\mathsf{Eff}_{\mathcal{C}}\,A]\!](t_1, t_2)$ if whenever they terminate, i.e., $t_1 \rightsquigarrow^* \mathsf{return}(u_1), o_1$ and $t_2 \rightsquigarrow^* \mathsf{return}(u_2), o_2$, the resulting terms are related, i.e., $\mathcal{R}^{l_{\mathtt{atk}}}_{\mathcal{E}}[\![\tau]\!](u_1, u_2)$, and the outputs are indistinguishable by the attacker, i.e., $o_1 =_{\mathcal{C}_{\mathtt{atk}}} o_2$. The fundamental theorem of the LR states that erased terms are related to themselves:

Theorem 4 (Fundamental Theorem of the LR for STLC$^{\mathbf{Print}}$). *For any attacker with sensitivity $l_{\mathtt{atk}}$, and* SC$^{\textsc{Print}}$ *term* $\{l_{\mathtt{atk}}\} ; \Gamma \vdash^{\mathtt{sc}} t : A$, *it is the case that* $\mathcal{R}^{l_{\mathtt{atk}}}_{\mathcal{T}}[\![(\Gamma, A)]\!](\varepsilon(t), \varepsilon(t))$.

Proof. By induction on the typing derivation with use of the Lemma 2.

The proof that SC$^{\textsc{Print}}$ enforces TINI (Theorem 3) follows as a corollary:

Proof. Let $\cdot \vdash s_1, s_2 : \mathsf{Bool}$ be two secrets. Since $l_{\mathtt{sec}} \not\sqsubseteq l_{\mathtt{atk}}$, the secrets are related, i.e., $\mathcal{R}^{l_{\mathtt{atk}}}_{\mathcal{E}}[\![\mathsf{S}_{l_{\mathtt{sec}}}\,\mathsf{Bool}]\!](s_1, s_2)$, and thus the substitutions $\gamma_1 = \{sec \mapsto s_1\}$ and $\gamma_2 = \{sec \mapsto s_2\}$ are related. By the fundamental theorem, the term $\varepsilon(t)$ is related to itself $\mathcal{R}^{l_{\mathtt{atk}}}_{\mathcal{E}}[\![\mathsf{Eff}_{\mathcal{C}}\,\mathsf{Unit}]\!](\varepsilon(t)[\gamma_1], \varepsilon(t)[\gamma_2])$. By assumption, $\varepsilon(t)[s_1/sec] \rightsquigarrow^* v_1, o_1$ and $\varepsilon(t)[s_2/sec] \rightsquigarrow^* v_2', o_2$. By reasoning about the operational semantics, there exists two intermediate programs, t_1', t_2' such that: $\varepsilon(t)[s_1/sec] \longrightarrow^* t_1'$ and $t_1' \rightsquigarrow^* v_1, o_1$; and $\varepsilon(t)[s_2/sec] \longrightarrow^* t_2'$ and $t_2' \rightsquigarrow^* v_2, o_2$. We apply $\mathcal{R}^{l_{\mathtt{atk}}}_{\mathcal{E}}[\![\mathsf{Eff}_{\mathcal{C}}\,\mathsf{Unit}]\!](\varepsilon(t)[s_1/sec], \varepsilon(t)[s_2/sec])$ to the two effect-free reductions which gives us that $\mathcal{R}^{l_{\mathtt{atk}}}_{\mathcal{V}}[\![\mathsf{Eff}_{\mathcal{C}}\,\mathsf{Unit}]\!](t_1', t_2')$. We apply this to the monadic reductions and obtain that $o_1 =_{\mathcal{C}_{\mathtt{atk}}} o_2$.

4.2 Noninterference for Global Store Effects: STLC$^{\mathbf{Store}}$ and SC$^{\mathbf{Store}}$

We formalize noninterference for STLC$^{\textsc{Store}}$ by looking at effectful programs which receive a store as input and produce a store as output. The contents of the store are possibly unevaluated STLC$^{\textsc{Store}}$ programs of ground type (see Figure 4). In order to compare stores, we define an indistinguishability relation for programs. This is a sort of LR parameterized by STLC$^{\textsc{Store}}$ ground types:

$$\mathcal{I}[\![-]\!] : (r : \text{STLC}^{\textsc{Store}} \text{ ground type}) \to (t_1 : \cdot \vdash r) \to (t_2 : \cdot \vdash r) \to \text{Set}$$
$$\mathcal{I}[\![\mathsf{Unit}]\!](t_1, t_2) :\Leftrightarrow \forall (v_1, v_2 : \cdot \vdash \mathsf{Unit}).\, t_1 \longrightarrow^* v_1 \wedge t_2 \longrightarrow^* v_2 \Rightarrow v_1 \equiv v_2$$
$$\mathcal{I}[\![\mathsf{Bool}]\!](t_1, t_2) :\Leftrightarrow \forall (v_1, v_2 : \cdot \vdash \mathsf{Bool}).\, t_1 \longrightarrow^* v_1 \wedge t_2 \longrightarrow^* v_2 \Rightarrow v_1 \equiv v_2$$

In some sense it resembles the LRs for Unit and Bool types in Figure 6.

Stores are parameterized by store typings that determine the type of the contents at each location. Since stores neither grow nor shrink, assumption 1 (Section 3.2), we define an indistinguishability relation for stores of the same store typing. Indistinguishability is parameterized by a subset of the locations.

Definition 4 (Store indistinguishability). *Let $\mathcal{S} \subseteq \mathcal{Loc}$. Two stores θ_1 and θ_2 of store typing a are $\mathcal{S}$-indistinguishable, denoted by $\theta_1 =_{\mathcal{S}} \theta_2$, if the two stores are indistinguishable at each location in $\mathcal{S}$, i.e., $\forall s \in \mathcal{S}.\ \mathcal{I}[\![\Sigma(s)]\!](\theta_1(s), \theta_2(s))$.*

TINI for STLC$^{\textsc{Store}}$ programs is:

Definition 5 (TINI for STLC$^{\mathbf{Store}}$). *An* STLC$^{\textsc{Store}}$ *program* $\Sigma, \cdot \vdash f : \mathsf{Eff}_{\mathcal{S}}\,\mathsf{Unit}$ *satisfies TINI* with respect to $\mathcal{S}' \subseteq \mathcal{Loc}$, *if for any two stores* $\theta_1, \theta_2 : \Sigma$, *any two* STLC$^{\textsc{Store}}$ *values* $\cdot \vdash v_1, v_2 : \mathsf{Eff}_{\mathcal{S}}\,\mathsf{Unit}$, *and any two stores* $\theta_1', \theta_2' : \Sigma$, *if* $\theta_1 =_{\mathcal{S}} \theta_2$ *and* $\theta_1, f \rightsquigarrow^* \theta_1', v_1$ *and* $\theta_2, f \rightsquigarrow^* \theta_2', v_2$ *then* $\theta_1' =_{\mathcal{S}} \theta_2'$.

Again, SC$^{\textsc{Store}}$ enforces TINI on STLC$^{\textsc{Store}}$ programs:

$$\mathcal{R}^{l_{\mathtt{atk}}}_{\mathcal{V}}[\![\ldots]\!](t_1,t_2) :\Leftrightarrow \ldots$$
$$\mathcal{R}^{l_{\mathtt{atk}}}_{\mathcal{V}}[\![\mathsf{Eff}_C\, A]\!](t_1,t_2) :\Leftrightarrow \forall(u_1,u_2 : \cdot \vdash \varepsilon(A))(\theta_1,\theta_2,\theta_1',\theta_2' : \Sigma).\, \theta_1 =_{\mathcal{S}_{\mathtt{atk}}} \theta_2 \wedge$$
$$\theta_1, t_1 \rightsquigarrow^* \theta_1', \mathsf{return}(u_1) \wedge \theta_2, t_2 \rightsquigarrow^* \theta_2', \mathsf{return}(u_2) \Rightarrow \mathcal{R}^{l_{\mathtt{atk}}}_{\mathcal{E}}[\![A]\!](u_1,u_2) \wedge \theta_1' =_{\mathcal{S}_{\mathtt{atk}}} \theta_2'$$

Fig. 8. Logical relation for STLC$^{\textsc{Store}}$ (excerpts)

Theorem 5. *Given any* SC$^{\textsc{Store}}$ *term* $\{l_{\mathtt{atk}}\}\ ;\ \Sigma, \cdot \vdash^{\mathsf{sc}} f : \mathsf{Eff}_{\mathcal{S}}\ \mathsf{Unit}$ *the erased program* $\Sigma, \cdot \vdash \varepsilon(f) : \mathsf{Eff}_{\mathcal{S}}\ \mathsf{Unit}$ *satisfies TINI* with respect to $\mathcal{S}_{\mathtt{atk}}$ *where* $\mathcal{S}_{\mathtt{atk}} := \{s \mid s \in \mathcal{L}oc, \mathrm{label}(s) \sqsubseteq l_{\mathtt{atk}}\}$.

The LR that we construct to prove TINI (Figure 8) is largely similar to that of STLC$^{\textsc{Print}}$ (Figures 6 and 7), with the difference that effectful programs take as argument and produce as result indistinguishable pairs of stores. TINI follows as a consequence of the fundamental theorem of the LR.

5 Implementation

In this section, we present an implementation of SC and SC$^{\textsc{Print}}$ (Sections 2 and 3.1) as a Haskell library, which we call SCLib. We omit SC$^{\textsc{Store}}$ (Section 3.2) for lack of space. However, its implementation is similar to that of SC$^{\textsc{Print}}$. Furthermore, we demonstrate that existing Haskell libraries for static IFC can be reimplemented in terms of the interface that SCLib exposes.

The main characteristic of SC is that typing judgements $\pi\ ;\ \Gamma \vdash^{\mathsf{sc}} t : A$ are indexed by a set of labels π. Onwards, we refer to the left part the judgement, i.e., $\pi\ ;\ \Gamma$, as the context of the term t. The set of labels plays an important role in enforcing IFC. However, individual labels are not first-class citizens: there is no type of labels and, hence, labels can neither be introduced nor eliminated. Further, some typing rules in SC modify the set of labels in their context: e.g., the rule for unseal_l (cf. Figure 1) augments the set in its premise with l. When shallowly embedding in Haskell any calculus that manipulates the context in this fashion, there is a natural problem to overcome: Haskell does not allow library implementors to have access to a program's context. For instance, to embed a linear type system in Haskell, Bernardy et al. [7] need to change the compiler.

To overcome these difficulties, our implementation resorts to a combination of Haskell's module system to hide the implementation details' from users, and a well-known trick of higher-rank polymorphism [20] to prevent first-class access to labels. We hope to convince the reader that our simple implementation matches the studied enforcement mechanisms, and that it can shed light on previous work on static IFC in Haskell.

5.1 Implementation of SC

When we introduced SC, we mentioned the intuition that labels in π are some sort of type-level keys whose possession permits access to sealed data. Our implementation takes this intuition literally: there is a type for keys whose elements are attached with type-level labels, and the primitive to unseal, i.e., unseal, is

parameterized by a key. The elements of this type are like capabilities [17] which need to be explicitly exercised.

```
1  module SCLib
2    (Key (), Label (..), FlowsTo (..), S (S), seal, unseal, ...)
3  where
4  -- Enumeration of security labels for the two-point lattice
5  data Label = H | L
6  -- "Flows to" relation as a type-class
7  class FlowsTo (l :: Label) (l' :: Label)
8  -- Instances
9  instance FlowsTo l l
10 instance FlowsTo L H
11 -- Type-level keys
12 data Key l s = Key
13 -- Security type
14 data S l a = S (forall s. Key l s -> a)
15 -- Sealing
16 seal :: (forall s. Key l s -> a) -> S l a
17 seal = Seal
18 -- Unsealing
19 unseal :: FlowsTo l' l => Key l s -> S l' a -> a
20 unseal k@Key (S f) = f Key
```

Fig. 9. Implementation of SC (e.g., for the two-point security lattice)

Figure 9 shows the *complete* implementation of SC in Haskell. Without loss of generality, we assume the two point-security lattice. As previous work (e.g., MAC [33], HLIO [9], and DCC [5]) we represent labels as types of kind `Label` (line 5, and the use of the GHC extension `DataKinds`) and encode the "flows to" relation via a type-class (lines 7–10). For simplicity we show the encoding of the two-point security lattice, however, this can be generalized (cf. [9]). In line 12 we introduce a new datatype `Key` which is parameterized by a type `l` of kind `Label` and a phantom type `s`. Then, line 14 introduces the type `S`, which is a wrapper over the function space between the types `Key l s` and `a`, i.e., `forall s. Key l s -> a`. The constructor of `S l a`, i.e., `S`, uses higher-rank polymorphism, as evidenced by the `forall` keyword, to quantify over the type variable `s` in `Key l s`. This ensures that the type of any newly bound key will be unique: the type `s` will not coincide with any other type `s'`. Launchbury and Jones [20] use the same trick, for example, to provide pure mutable references in Haskell where the type variable represents memory regions. Since the type variable `s` does not coincide with any other, elements of the type `Key l s` are not first-class values, and, for example, cannot be stored: the program `S (\k -> k)` does not type check.

We now implement the primitives seal and unseal—Rules Seal and Unseal from Figure 1. The combinator `seal` (lines 16 and 17) is a wrapper over the

constructor of the type `S l a`. The combinator `unseal` (lines 19–20) allows unsealing terms of type `S l a` in case we *have* a value of type `Key l' s` and the label in its type, i.e., `l'`, flows to `l`—see the constraint `FlowsTo l' l`. In order to enforce IFC, it is important that the constructors of `Key` are kept abstract from the user—observe `Key ()` in the export list of the module (line 2). Otherwise, anyone could extract the underlying term of type `a` from an `l`-sensitive value `secret :: S l a` by applying `unseal Key`. Similar to Russo, Claessen, and Hughes [34], the combinator `unseal` is strict in its argument (`k@Key`) in order to forbid forged keys like `undefined :: Key l s`. We remark that the noninterference property—recall TINI from Definitions 1 and 3—rules out programs that force `undefined` and halt with error.

The implementation discussed so far consists of the *trusted computing base* (TCB) of SCLib. From now on, users of the library can derive functionalities from the library's interface. For example, programmers can implement the `Functor`, `Applicative` and `Monad` instances for the type `S l a` as follows:

```
instance Functor (S l) where
  fmap f x = seal (\k -> f (unseal k x))

instance Applicative (S l) where
  pure x  = seal (\k -> x)
  f <*> a = seal (\k -> (unseal k f) (unseal k a))

instance Monad (S l) where
  return = pure
  m >>= f = seal (\k -> unseal k (f (unseal k m)))
```

Note that the programmer does not need access to the TBC in order to implement these instances—as opposed to MAC (cf. [41]). This phenomenon, we believe, is a sign of the simplicity and generality of our implementation.

5.2 Implementation of SC$^{\text{Print}}$

Figure 10 shows the implementation of SC$^{\text{Print}}$ which builds on the implementation of SC (Figure 9). The datatype `Eff` wraps `IO` computations and is indexed by a type-level set [28] `ls` of channels where the computation can write to. For simplicity, we will omit the map label (Figure 3) from channels to labels and identify channels with labels so that the index `ls` has kind `[Label]`. This makes the implementation simpler.

In lines 15–18 we implement the return and bind of the graded monad. `returnEff` does not produce effects, thus, its type is indexed by the empty set `[]` (cf. Figure 3). `bindEff` type is indexed by the union of the sets of labels of the computations (cf. Figure 3). The implementation of subeffecting is the identity function (lines 20–21). Printing effects can be performed by the combinator `printEff` (lines 26–30). The argument of type `SLabel l` is a term level representation of a the type-level label of printing channel.

Lines 36–38 show the implementation of the novel primitive `distr`. The type constraint `FlowsToSet l ls` (see the definitions in lines 32–35) ensures that

```
module SCLib
  (..., Eff (), FlowsToSet (), pureEff, appEff, returnEff
  , bindEff, distr, subeff, printEff)
where

newtype Eff (ls :: [Label]) a = Eff { runEff :: IO a }

-- Functor
instance Functor (Eff ls) where
  fmap f (Eff io) = Eff (fmap f io)

-- Applicative
pureEff :: a -> Eff [] a
pureEff = returnEff

appEff :: Eff ls1 (a -> b) -> Eff ls2 a -> Eff (Union ls1 ls2) b
appEff (Eff ioff) (Eff ioa) = Eff (ioff <*> ioa)

-- Graded 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)

-- Subeffecting
subeff :: Subset ls1 ls2 => Eff ls1 a -> Eff ls2 a
subeff (Eff m) = Eff m

-- Print
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:"

-- 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 => S l (Eff ls a) -> Eff ls (S l a)
distr m = Eff (do res <- (runEff (unseal Key m))
                  return (seal (\k -> res)))
```

Fig. 10. Implementation of SC$^{\text{Print}}$ (excerpts)

`l` $\sqsubseteq$ `l'` for every label `l'` in `ls`. The implementation uses the value `Key`, which pertains to the TCB, to unseal the `Eff` action, i.e., `unseal Key m`; and then it runs its effects, i.e., `res <- runEff (unseal Key m)`; finally it seals the result at the same label, i.e., `return (seal (\k -> res))`.

5.3 Implementing existing libraries for IFC

We conclude this section by showing that we can reimplement some of the existing libraries in Haskell for IFC. We show implementations of SecLib [34], SDCC [4] (an alternative presentation of DCC) and a variation of MAC [33][5] using SCLib interface. In some sense the implementations help to explain the mentioned libraries. Further, this shows that the programmer can choose to write programs against SCLib's "low-level" interface; or a more "high-level" interface, e.g., MAC; or a combination of both. We declare future work to compare the performance among implementations.

For each library we briefly explain its interface and show its implementation in terms of SCLib.

SecLib and SDCC. SecLib is one of the pioneers of static IFC libraries in the context of Haskell. Its main feature is a family of security monads `Sec` indexed by labels from the security lattice, each equipped with `>>=` (bind) and `return`. SecLib's special ingredient is a combinator `up` that allows coercions from lower to higher labels in the security monad.

SDCC is an alternative presentation of DCC which favors a simple set of combinators instead of DCC's non-standard bind and *protected at* relation. Similar to DCC, SDCC sports a family of monads indexed by security labels. These support `fmap`, `return` and `>>=` (bind). Further, SDCC implements two combinators `up` and `com`, that allow to relabel in the style of SecLib and commute terms of monadic type with different labels. SDCC's interface is strictly a superset of that of SecLib, thus we directly show the implementation of the former. The left column displays the interface whilst the right its implementation in SCLib.

```
type T l a

instance Functor (Sec l) where
instance Monad (Sec l) where

up :: FlowsTo l l' -> T l a -> T l' a

com :: T l (T l' a) -> T l' (T l a)
```

```
type T l a = S l a

...
...

up :: FlowsTo l l' => T l a -> T l' a
up lv = seal (\k -> unseal k lv)

com :: T l (T l' a) -> T l' (T l a)
com lv = seal (\k' -> seal (\k ->
           unseal k' (unseal k lv)))
```

MAC which we discussed in the introduction, is one of the state-of-the-art libraries for effectful IFC in Haskell. At its core, MAC defines two types: `Labeled l a` for pure sensitive values, and `MAC l a` for secure computations. `MAC l a` is a monad for each label `l` where the label: 1. protects the data in

[5] In our variation the type `Labeled` is a monad. This is "unsafe" in MAC (cf. [42]).

context; and 2. restricts the permitted effects. (cf. [43]) MAC's functionality stems from the interaction between `Labeled` and `MAC` through the primitives `label` and `unlabel`. In order to label a value one needs to do so within the `MAC l` monad: the `Labeled l a` type does not export a combinator `a -> Labeled l a`. Our implementation, however, permits to do so. The left column shows MAC's interface, with the combinator `join` in its full generality, and the right column hints its implementation in terms of SCLib.

```
type Labeled l a

type MAC l a


label :: FlowsTo l l'
      => a -> MAC l (Labeled l' a)

unlabel :: FlowsTo l l'
        => Labeled l a -> MAC l' a

join :: FlowsTo l l'
        => MAC l' a
        -> MAC l (Labeled l' a)
```

```
type Labeled l a = S l a

type MAC l a = forall ls. FlowsToSet l ls
                  => Eff ls (S l a)

label :: FlowsTo l l' => ...
label a = returnEff (seal (\k -> a))

unlabel :: FlowsTo l l' => ...
unlabel lv = returnEff lv

join :: FlowsTo l l' => ...
join m = bindEff m (\x -> seal (\k -> x))
```

6 Related work

IFC for effect-free and effectful languages. Algehed and Russo [5] add effects to their embedding of DCC in Haskell but argue that their approach only works for those effects that can be implemented within Haskell. Hirsch and Cecchetti [15] develop a formal framework based on productors and type-and-effect systems to characterize secure programs in impure languages with IFC. They give semantics to traditional security-type systems based on controlling implicit flows using PC labels. In contrast, our approach considers from starters a pure language, where the type of effectful computation is separated from that of effect-free programs.

Modalities for IFC. The languages and IFC enforcement mechanisms that we present are based on the Sealing Calculus (SC) of Shikuma and Igarashi [37]. Differently from them, we think of SC terms as evidence that STLC programs satisfy noninterference. The work by Miyamoto and Igarashi [24] gives an informal connection between a classical type system for IFC and a certain modal logic. Their type system is very different from our enforcement mechanism in that a typing judgement has two separate variable contexts. Recently, the work by Abel and Bernardy [3] presents a unified treatment of modalities in typed lambda calculi. The authors present a effect-free lambda calculus parameterized by family of modalities with certain mathematical structure, and show that many PL analyses, including IFC, are instantiations of their framework. In contrast to our work, it is not very clear how one would implement theirs system in Haskell, since it would require a fine-grained control over the variables in the context.

Kavvos [19] studies modalities for IFC in the classified sets model, which they use to prove noninterference properties for a range of calculi that includes SC.

Coeffectful type systems for IFC. A recent line of work suggests using coeffect type systems to enforce IFC. Petricek, Orchard, and Mycroft [30] develop a calculus to capture different granularity demands on contexts, i.e., flat whole-context coeffects (like implicit parameters [21]) or structural per-variable ones (like usage or data access patterns). The work by Gaboardi et al. [10] expands on that and uses graded monads and comonads to combine effects and coeffects. The authors describe *distributivity laws* that are similar to our primitive distr addresses. The article suggests IFC as an application where the coeffect system captures the IFC constraints and the effect system gives semantics to effects. The distributive laws explains how both are combined. However, their work does not state neither proves a security property for their calculus. Different from it, our work does not use comonads as the underlying structure for IFC, and further considers printing and global store effects. Granule is a recent programming language [27] based on *graded modal types* that impose usage constraints on the variables.

Logical relations for noninterference. Both Heintze and Riecke [14] and Zdancewic [46] use logical relation arguments to prove noninterference for a simply-typed security lambda calculus. Tse and Zdancewic [40] use logical relations to prove soundness of a translation from DCC [1] to System F and obtain noninterference from parametricity. Unfortunately, their translation is unsound (cf. [37]). Bowman and Ahmed [8] fix this by using "open" logical relations show their translation from DCC to System F_ω is sound. Different from the cited work so far, Rajani and Garg [32] use logical relations to prove noninterference for a language with references. Gregersen et al. [12] extend the use of logical relation to prove noninterference for languages with inpredicative polymorphism. Different from Rajani and Garg [32] and Gregersen et al. [12], we consider first-order references for simplicity. Otherwise, we should have had to utilize a step-indexed Kripke-style logical-relations model, which would have introduced technical complications that are orthogonal to the main contribution of our work.

7 Conclusions

In this paper, we have demonstrated that to enforce IFC in pure languages with a single primitive distr suffices to securely control what information flows from sensitive data to effects. To support our claim, we have presented IFC enforcement mechanisms for several kinds of effects and proved that they satisfy noninterference. Our development rests on the insight that effect-free IFC for pure languages can already express that a computation will not leak sensitive data through its effects when executed. Then, a single primitive, distr, to execute these is enough to extend IFC to effects and retain the security guarantees. We hope that this work brings a new perspective to IFC research for pure languages with effects.

A STLC

$$\begin{array}{rrl} \text{Types} & a, b & ::= \ \mathsf{Unit} \mid \mathsf{Bool} \mid a \Rightarrow b \\ \text{Typing contexts} & \Gamma & ::= \ \cdot \mid \Gamma, x : a \end{array}$$

$\boxed{\Gamma \vdash t : a}$

$$\textsc{Var}\ \frac{(x : a) \in \Gamma}{\Gamma \vdash x : a} \qquad \textsc{Lam}\ \frac{\Gamma, x : a \vdash t : b}{\Gamma \vdash \lambda(x.\, t) : a \Rightarrow b} \qquad \textsc{App}\ \frac{\Gamma \vdash t : a \Rightarrow b \qquad \Gamma \vdash u : a}{\Gamma \vdash \mathsf{app}(t, u) : b}$$

$$\textsc{Unit}\ \frac{}{\Gamma \vdash \mathsf{unit} : \mathsf{Unit}} \qquad \textsc{True}\ \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}} \qquad \textsc{False}\ \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}$$

$$\textsc{If}\ \frac{\Gamma \vdash t : \mathsf{Bool} \qquad \Gamma \vdash u_1 : a \qquad \Gamma \vdash u_2 : a}{\Gamma \vdash \mathsf{ifte}(t, u_1, u_2) : a}$$

$\boxed{t \longrightarrow u \text{ with } \cdot \vdash t : a \text{ and } \cdot \vdash u : a}$

$$\textsc{App}\ \frac{t \longrightarrow t'}{\mathsf{app}(t, u) \longrightarrow \mathsf{app}(t', u)} \qquad \textsc{Beta}\ \frac{}{\mathsf{app}(\lambda(x.\, t), u) \longrightarrow t[u/x]}$$

$$\textsc{If}\ \frac{t \longrightarrow t'}{\mathsf{ifte}(t, u_1, u_2) \longrightarrow \mathsf{ifte}(t', u_1, u_2)} \qquad \textsc{If-True}\ \frac{}{\mathsf{ifte}(\mathsf{true}, u_1, u_2) \longrightarrow u_1}$$

$$\textsc{If-False}\ \frac{}{\mathsf{ifte}(\mathsf{false}, u_1, u_2) \longrightarrow u_2}$$

Fig. 11. Types, intrinsically-typed terms and small-step semantics of (call-by-name) STLC.

References

1. Abadi, M., Banerjee, A., Heintze, N., and Riecke, J.G.: A Core Calculus of Dependency. In: Appel, A.W., and Aiken, A. (eds.) POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999, pp. 147–160. ACM (1999). DOI: 10.1145/292540.292555
2. [SW Rel.] Abel, A., Allais, G., Cockx, J., Danielsson, N.A., Hausmann, P., Nordvall Forsberg, F., Norell, U., López Juan, V., Sicard-Ramírez, A., and Vezzosi, A., *Agda 2* version 2.6.1.3, 2005–2021. Chalmers University of Technology and Gothenburg

University. LIC: BSD3. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php, VCS: https://github.com/agda/agda.
3. Abel, A., and Bernardy, J.: A unified view of modalities in type systems. Proc. ACM Program. Lang. 4(ICFP), 90:1–90:28 (2020). DOI: 10.1145/3408972
4. Algehed, M.: A Perspective on the Dependency Core Calculus. In: Alvim, M.S., and Delaune, S. (eds.) Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2018, Toronto, ON, Canada, October 15-19, 2018, pp. 24–28. ACM (2018). DOI: 10.1145/3264820.3264823
5. Algehed, M., and Russo, A.: Encoding DCC in Haskell. In: Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2017, Dallas, TX, USA, October 30, 2017, pp. 77–89. ACM (2017). DOI: 10.1145/3139337.3139338
6. Askarov, A., Hunt, S., Sabelfeld, A., and Sands, D.: Termination-Insensitive Noninterference Leaks More Than Just a Bit. In: Jajodia, S., and López, J. (eds.) Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings. LNCS, vol. 5283, pp. 333–348. Springer, Heidelberg (2008). DOI: 10.1007/978-3-540-88313-5_22
7. Bernardy, J., Boespflug, M., Newton, R.R., Jones, S.P., and Spiwack, A.: Linear Haskell: practical linearity in a higher-order polymorphic language. CoRR abs/1710.09756 (2017)
8. Bowman, W.J., and Ahmed, A.: Noninterference for free. In: Fisher, K., and Reppy, J.H. (eds.) Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015, pp. 101–113. ACM (2015). DOI: 10.1145/2784731.2784733
9. Buiras, P., Vytiniotis, D., and Russo, A.: HLIO: mixing static and dynamic typing for information-flow control in Haskell. In: Fisher, K., and Reppy, J.H. (eds.) Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015, pp. 289–301. ACM (2015). DOI: 10.1145/2784731.2784758
10. Gaboardi, M., Katsumata, S., Orchard, D.A., Breuvart, F., and Uustalu, T.: Combining effects and coeffects via grading. In: Garrigue, J., Keller, G., and Sumii, E. (eds.) Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016, pp. 476–489. ACM (2016). DOI: 10.1145/2951913.2951939
11. Goguen, J.A., and Meseguer, J.: Security Policies and Security Models. In: 1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982, pp. 11–20. IEEE Computer Society (1982). DOI: 10.1109/SP.1982.10014
12. Gregersen, S.O., Bay, J., Timany, A., and Birkedal, L.: Mechanized logical relations for termination-insensitive noninterference. Proc. ACM Program. Lang. 5(POPL), 1–29 (2021). DOI: 10.1145/3434291
13. Hedin, D., and Sabelfeld, A.: A Perspective on Information-Flow Control. In: Software Safety and Security - Tools for Analysis and Verification. Ed. by T. Nipkow, O. Grumberg, and B. Hauptmann, pp. 319–347. IOS Press (2012). DOI: 10.3233/978-1-61499-028-4-319
14. Heintze, N., and Riecke, J.G.: The SLam Calculus: Programming with Secrecy and Integrity. In: MacQueen, D.B., and Cardelli, L. (eds.) POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998, pp. 365–377. ACM (1998). DOI: 10.1145/268946.268976

15. Hirsch, A.K., and Cecchetti, E.: Giving semantics to program-counter labels via secure effects. Proc. ACM Program. Lang. 5(POPL), 1–29 (2021). DOI: 10.1145/3434316
16. Jones, S.L.P., and Wadler, P.: Imperative Functional Programming. In: Deusen, M.S.V., and Lang, B. (eds.) Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993, pp. 71–84. ACM Press (1993). DOI: 10.1145/158511.158524
17. Kain, R.Y., and Landwehr, C.E.: On Access Checking in Capability-Based Systems. IEEE Trans. Software Eng. 13(2), 202–207 (1987). DOI: 10.1109/TSE.1987.232892
18. Katsumata, S.: Parametric effect monads and semantics of effect systems. In: Jagannathan, S., and Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014, pp. 633–646. ACM (2014). DOI: 10.1145/2535838.2535846
19. Kavvos, G.A.: Modalities, cohesion, and information flow. Proc. ACM Program. Lang. 3(POPL), 20:1–20:29 (2019). DOI: 10.1145/3290333
20. Launchbury, J., and Jones, S.L.P.: Lazy Functional State Threads. In: Sarkar, V., Ryder, B.G., and Soffa, M.L. (eds.) Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994, pp. 24–35. ACM (1994). DOI: 10.1145/178243.178246
21. Lewis, J.R., Launchbury, J., Meijer, E., and Shields, M.: Implicit Parameters: Dynamic Scoping with Static Types. In: Wegman, M.N., and Reps, T.W. (eds.) POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000, pp. 108–118. ACM (2000). DOI: 10.1145/325694.325708
22. Li, P., and Zdancewic, S.: Encoding Information Flow in Haskell. In: 19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy, p. 16. IEEE Computer Society (2006). DOI: 10.1109/CSFW.2006.13
23. Mitchell, J.C.: Foundations for programming languages. MIT Press (1996)
24. Miyamoto, K., and Igarashi, A.: A modal foundation for secure information flow. In: In Proceedings of IEEE Foundations of Computer Security (FCS), pp. 187–203 (2004)
25. Moggi, E.: Notions of Computation and Monads. Inf. Comput. 93(1), 55–92 (1991). DOI: 10.1016/0890-5401(91)90052-4
26. [SW] Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., and Nystrom, N., *Jif: Java information flow* version 3.0, 2006. URL: http://www.cs.cornell.edu/jif.
27. Orchard, D., Liepelt, V., and III, H.E.: Quantitative program reasoning with graded modal types. Proc. ACM Program. Lang. 3(ICFP), 110:1–110:30 (2019). DOI: 10.1145/3341714
28. Orchard, D.A., and Petricek, T.: Embedding effect systems in Haskell. In: Swierstra, W. (ed.) Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014, pp. 13–24. ACM (2014). DOI: 10.1145/2633357.2633368
29. Parker, J., Vazou, N., and Hicks, M.: LWeb: information flow security for multi-tier web applications. Proc. ACM Program. Lang. 3(POPL), 75:1–75:30 (2019). DOI: 10.1145/3290388
30. Petricek, T., Orchard, D.A., and Mycroft, A.: Coeffects: a calculus of context-dependent computation. In: Jeuring, J., and Chakravarty, M.M.T. (eds.) Proceed-

ings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014, pp. 123–135. ACM (2014). DOI: 10.1145/2628136.2628160
31. Polikarpova, N., Stefan, D., Yang, J., Itzhaky, S., Hance, T., and Solar-Lezama, A.: Liquid information flow control. Proc. ACM Program. Lang. 4(ICFP), 105:1–105:30 (2020). DOI: 10.1145/3408987
32. Rajani, V., and Garg, D.: On the expressiveness and semantics of information flow types. J. Comput. Secur. 28(1), 129–156 (2020). DOI: 10.3233/JCS-191382
33. Russo, A.: Functional pearl: two can keep a secret, if one of them uses Haskell. In: Fisher, K., and Reppy, J.H. (eds.) Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015, pp. 280–288. ACM (2015). DOI: 10.1145/2784731.2784756
34. Russo, A., Claessen, K., and Hughes, J.: A library for light-weight information-flow security in haskell. In: Gill, A. (ed.) Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008, pp. 13–24. ACM (2008). DOI: 10.1145/1411286.1411289
35. Sabelfeld, A., and Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. 21(1), 5–19 (2003). DOI: 10.1109/JSAC.2002.806121
36. Schoepe, D., Hedin, D., and Sabelfeld, A.: SeLINQ: tracking information across application-database boundaries. In: Jeuring, J., and Chakravarty, M.M.T. (eds.) Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014, pp. 25–38. ACM (2014). DOI: 10.1145/2628136.2628151
37. Shikuma, N., and Igarashi, A.: Proving Noninterference by a Fully Complete Translation to the Simply Typed Lambda-Calculus. Log. Methods Comput. Sci. 4(3) (2008). DOI: 10.2168/LMCS-4(3:10)2008
38. [SW] Simonet, V., *Flow Caml* 2003. URL: http://cristal.inria.fr/~simonet/soft/flowcaml/.
39. Stefan, D., Russo, A., Mitchell, J.C., and Mazières, D.: Flexible dynamic information flow control in Haskell. In: Claessen, K. (ed.) Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011, pp. 95–106. ACM (2011). DOI: 10.1145/2034675.2034688
40. Tse, S., and Zdancewic, S.: Translating dependency into parametricity. In: Okasaki, C., and Fisher, K. (eds.) Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004, pp. 115–125. ACM (2004). DOI: 10.1145/1016850.1016868
41. Vassena, M., Buiras, P., Waye, L., and Russo, A.: Flexible Manipulation of Labeled Values for Information-Flow Control Libraries. In: Askoxylakis, I.G., Ioannidis, S., Katsikas, S.K., and Meadows, C.A. (eds.) Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I. LNCS, vol. 9878, pp. 538–557. Springer, Heidelberg (2016). DOI: 10.1007/978-3-319-45744-4_27
42. Vassena, M., and Russo, A.: On Formalizing Information-Flow Control Libraries. In: Murray, T.C., and Stefan, D. (eds.) Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016, pp. 15–28. ACM (2016). DOI: 10.1145/2993600.2993608
43. Vassena, M., Russo, A., Buiras, P., and Waye, L.: MAC A verified static information-flow control library. Journal of Logical and Algebraic Methods in Programming 95, 148–180 (2018). DOI: https://doi.org/10.1016/j.jlamp.2017.12.003

44. Vassena, M., Russo, A., Garg, D., Rajani, V., and Stefan, D.: From fine- to coarse-grained dynamic information flow control and back. Proc. ACM Program. Lang. 3(POPL), 76:1–76:31 (2019). DOI: 10.1145/3290389
45. Wadler, P., and Thiemann, P.: The marriage of effects and monads. ACM Trans. Comput. Log. 4(1), 1–32 (2003). DOI: 10.1145/601775.601776
46. Zdancewic, S.A.: Programming languages for information security. Cornell University (2002)