Noninterference Half-Off: The Semantics of Program Counter Labels and Effects

***Abstract*—Type systems designed for information-flow control use a *program-counter label* to track the sensitivity of the context, ruling out data leakage arising from effectful computation in a sensitive context. Proofs of noninterference—the core security theorem for information flow—have traditionally required careful manual reasoning about effects. We build a generic semantic framework for proving noninterference in a broad range of effectful languages. Our framework relies on a standard semantic technique for effectful programs to eliminate the need for much of the error-prone manual reasoning. In the process, we unify notions of security for different types of effects, including state, exceptions, and nontermination.**

***Keywords*—producer effects; information flow control; noninterference**

I. Introduction

Noninterference—the powerful safety property that a program's sensitive inputs will not influence its less-sensitive outputs [1]—is the foundational security theorem for information-flow systems. Unfortunately, it is difficult to prove in effectful languages. Great effort has gone into creating numerous specialized type systems that enforce noninterference for functional and imperative languages, with and without effects, where data sensitivity is defined by confidentiality, integrity, or even distributed consistency [2], [3], [4], [5], [6], [7], [8], [9], [10], [11].

While these works provide valuable tools for securing different language constructs and domains, their security proofs use carefully tailored and often complicated reasoning. The specificity of each approach makes it difficult to extract more general insights from an individual piece of work, but across multiple designs themes emerge. These static information-flow control (IFC) systems often use similar underlying ideas to prevent information leaks through either outputs or effects. In this paper, we distill these common ideas into a highly general framework for proving noninterference in a wide variety of effectful languages.

Static IFC assigns *information-flow labels* to data within a program. These labels describe the sensitivity of the data. For instance, data labeled secret is more sensitive than data labeled public. The type system then prevents more-sensitive information from influencing less-sensitive output. A flow of information can be *explicit*, if a program directly returns an input, or *implicit* if a program conditions on the input and returns a different value from each branch. In both cases, the type system can enforce noninterference by checking that its output is at least as sensitive as the input.

When we combine effects with implicit flows, however, this simple output checking becomes insufficient. Volpano et al. [2] demonstrate this concern with the following simple program where the secret value x is either 0 or 1 and write modifies the state and returns the singleton value $()$ of type unit:

$$\mathsf{if}\ x = 1\ \mathsf{then}\ \mathsf{write}(1)\ \mathsf{else}\ \mathsf{write}(0)$$

While this program does not directly write a secret and always returns the same thing, an attacker who can read the final state now learns the value of x.

Languages often rule out these effectful implicit flows by tracking the sensitivity of the current control-flow by including a *program-counter label* (written pc) in the typing judgment [5], [10], [12]. If the pc is private, then private data influenced which command is executing, so writing to public outputs may leak that data. If the pc is public, however, only public data has determined which program path was taken, so a decision to write leaks nothing. With this reasoning, we can prove that well-typed programs in pc-tracking languages are noninterfering.

We can alternatively check the above program by translating the stateful operations to a monadic form. That is, a form that returns a pair consisting of the original output and the state set by write.

$$\mathsf{if}\ x = 1\ \mathsf{then}\ ((), 0)\ \mathsf{else}\ ((), 1)$$

Now, simply checking the output is sufficient to detect any leaks, without need for a pc label.

This suggests a strong semantic connection between program counters and monadic treatments of effects. We develop this semantic connection by developing a general framework of effectful languages with information flow labels. We base our framework on a categorical construct called a *productor* [13]. Productors provide the most-general known framework for the semantics of *producer effects*—a generalization of monadic effects. (In fact, Tate [13] argues that productors are the most general possible framework for producer effects.)

Our core theorem, the *Noninterference Half-Off Theorem*, enables proofs of noninterference for effectful languages (with pc labels) that fit our framework while only proving it directly for the pure part of the languages (without pc labels). We refer to this style of proof more broadly as *Noninterference Half-Off*. As far aware, this is the first general framework for proving noninterference in a large swath of languages.

We further use our framework to unify different notions of noninterference from the information-flow literature. Some notions consider the termination behavior of programs (termination-sensitive), while others do not (termination-insensitive). Our framework shows that these two notions of

noninterference are distinguished by whether or not nontermination is considered an effect. This view is both intellectually satisfying and provides half-off proofs of termination-sensitive noninterference.

Finally, our framework formalizes a piece of folklore: the pc label is a lower bound on the effects that can occur in a program. Taken literally, this folklore does not even seem to type-check: effects are not labels. However, because our framework interprets the pc as a connection between effects and labels, we can make it type-check.

In Section II, we review the Dependency Core Calculus (DCC) [4], a simple, pure, noninterfering language. DCC serves as an introduction to necessary parts of IFC languages and as the basis of our example languages throughout the paper. We then add the following contributions:

- We show how a monadic translation leads to a simple proof of noninterference for a language with state and exceptions (Section III). Pottier and Simonet [5] made such proofs tractable, but their technique required difficult and messy reasoning about effects.
- By treating possible nontermination as an effect—as is common in the effects literature—we obtain a simple proof of termination-sensitive noninterference (Section IV).
- We present our general semantic framework for effectful languages with IFC labels (Section V), allowing us to prove properties about a wide class of IFC languages.
- We define and prove the *Noninterference Half-Off Theorem* (Theorem 5), allowing us to extend noninterference from pure languages to effectful languages in many settings (Section VI).

II. An Information-Flow-Control Type System for a Pure Language

We begin by reviewing Abadi et al.'s [4] *Dependency Core Calculus* (DCC), a pure language with a simple noninterference property. It will form the basis of our examples in this paper. It also serves as a good language to introduce information-flow control (IFC) and noninterference, as well as the notation for this paper.

Figure 1 contains the syntax of DCC. The heart of DCC is the simply-typed λ-calculus with products and sums. The only additional terms are the security features that make DCC interesting from our perspective: $\mathsf{label}_\ell\,(e)$ and $\mathsf{unlabel}\ e_1\ \mathsf{as}\ x\ \mathsf{in}\ e_2$. We will also make free use of let notation, with its standard definition. (For simplicity, we omit the fixpoint operator present in the original language [4], though we will add it back in Section IV.)

The security terms use a set of *information-flow labels*, $\mathcal{L}$, over which DCC is parameterized, that represent restrictions on data use. For instance, if we have label secret and public, then data labeled secret should not be used to compute data labeled public. We require that labels form a preorder. That is, there is a reflexive and transitive relation $\sqsubseteq$ (pronounced "flows to"). For presentation clarity, we also assume that $\mathcal{L}$ forms a join semilattice, meaning any two labels ℓ_1 and ℓ_2 have a join—a least upper bound—denoted $\ell_1 \sqcup \ell_2$, and there is a *top* element, denoted $\top$, such that $\ell \sqsubseteq \top$ for all $\ell \in \mathcal{L}$. Intuitively, if $\ell_1 \sqsubseteq \ell_2$, then ℓ_2 is at least as restrictive as ℓ_1, so $\top$ is the most-restrictive label. We note that most IFC work assumes a lattice, meaning labels also have greatest lower bounds and there is a least element $\bot$. We omit this additional structure as we do not find it helpful.

The term $\mathsf{label}_\ell\,(e)$ represents protecting the output of e at label ℓ. That is, e should only be used to compute information at levels at least as high as ℓ. Such computations are possible using the term $\mathsf{unlabel}\ e_1\ \mathsf{as}\ x\ \mathsf{in}\ e_2$, which requires the output type of e_2 to be at a high-enough level, and if it is, allows use of e_1 through the variable x as if it were not labeled.

The concept of a type τ being "of high enough level" to use information at label ℓ is expressed in a relation $\ell \lhd \tau$, which is read as "ℓ is protected by τ" or "τ protects ℓ." The formal rules defining this relation are in Figure 2. Intuitively, if $\ell \lhd \tau$, then τ information is at least as secret as ℓ.

The typing rules for $\mathsf{label}_\ell\,(e)$ and $\mathsf{unlabel}\ e_1\ \mathsf{as}\ \ell\ \mathsf{in}\ e_2$ are as follows:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{label}_\ell\,(e) : L_\ell\,(\tau)}$$

$$\frac{\Gamma \vdash e_1 : L_\ell\,(\tau_1) \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \qquad \ell \lhd \tau_2}{\Gamma \vdash \mathsf{unlabel}\ e_1\ \mathsf{as}\ x\ \mathsf{in}\ e_2 : \tau_2}$$

Notice the use of the protection relation in the latter. Since unlabel allows a program to compute with labeled data, this check requires the output of that computation to be at least as sensitive as the input. This protection premise is the main security check in DCC's type system.

The operational semantics of DCC are mostly the standard semantics of call-by-value simply-typed λ-calculus, so we only discuss the semantics of the terms $\mathsf{label}_\ell\,(e)$ and $\mathsf{unlabel}\ e_1\ \mathsf{as}\ \ell\ \mathsf{in}\ e_2$. We first note that computation can take place under both $\mathsf{label}_\ell\,(-)$ and $\mathsf{unlabel}\ -\ \mathsf{as}\ x\ \mathsf{in}\ e_2$. Computation *cannot*, however, take place in the second expression of an unlabel operation, since this is expected to run after binding the variable x. The only remaining operational semantics rule is as follows:

$$\mathsf{unlabel}\ (\mathsf{label}_\ell\,(v))\ \mathsf{as}\ x\ \mathsf{in}\ e \longrightarrow e[x \mapsto v].$$

DCC's main security theorem is its *noninterference* theorem. It formalizes the fact that programs do not compute, e.g., public information with secret data. The theorem requires a notion of equivalence at a label ℓ, representing what an attacker who can see values only up to label ℓ can distinguish. The definition is contextual to allow for comparison of first-class functions.

Definition 1 (ℓ-Equivalent Programs)**.** We say programs e_1 and e_2 are *ℓ-equivalent*, denoted $e_1 \approx_\ell e_2$, if $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$, and for all substitutions $\theta : \Gamma$ and programs e such that $\Gamma, x : \tau \vdash e : L_\ell\,(\mathsf{unit} + \mathsf{unit})$ and $\theta(e[x \mapsto e_i]) \longrightarrow^* v_i$ for both $i = 1, 2$, then $v_1 = v_2$.

Here a substitution $\theta : \Gamma$ is a function on expressions that, for each variable x in Γ, substitutes x with a well-typed closed

$$\frac{\ell \sqsubseteq \ell'}{\ell \lhd L_{\ell'}(\tau)} \qquad \frac{\ell \lhd \tau}{\ell \lhd L_{\ell'}(\tau)} \qquad \frac{\ell \lhd \tau_1 \qquad \ell \lhd \tau_2}{\ell \lhd \tau_1 \times \tau_2} \qquad \frac{\ell \lhd \tau_2}{\ell \lhd \tau_1 \to \tau_2}$$

Fig. 2. Protection Rules for DCC

term of type $\Gamma(x)$. Therefore, if $\Gamma \vdash e : \tau$ and $\theta : \Gamma$, then $\vdash \theta(e) : \tau$.

Intuitively, expressions are ℓ-equivalent if no well-typed decision procedure with output labeled ℓ can distinguish them. Note that this definition only requires equivalent outputs when both programs terminate. Because we omitted DCC's fixpoint operator, the language is strongly normalizing so both terms always converge. We will address potential nontermination in Section IV.

We use this definition to say that two values must be ℓ-equivalent unless their labels allow them to influence ℓ. Formally, the protection relation define the label of data, leading to the following theorem. The version we use here was proven by Bowman and Ahmed [14] and a verified-correct proof written in Agda was given by Algehed and Bernardy [15].

Theorem 1 (Noninterference for DCC [14], [15])**.** *For expressions e_1 and e_2 and $\ell \in \mathcal{L}$, if $\Gamma \vdash e_1 : \tau$, $\Gamma \vdash e_2 : \tau$, and $\ell \lhd \tau$, then for all labels $\ell_{\mathsf{Atk}} \in \mathcal{L}$, either $\ell \sqsubseteq \ell_{\mathsf{Atk}}$ or $e_1 \approx_{\ell_{\mathsf{Atk}}} e_2$.*

III. Example: Noninterference in a Language with State and Exceptions

We now extend our simple noninterfering pure language with two effects: state and exceptions. Both are simplified for space and ease of understanding. Specifically, we consider only one (typed) state cell and one type of exception. Moreover, the type of this state cell, σ, must not contain a function as a subterm in order to avoid concerns around higher-order state.[1] However, neither is difficult to broaden to more-realistic versions. We extend the language with the following syntax:

$$\begin{array}{rcl} e & ::= & \cdots \mid \mathsf{read} \mid \mathsf{write}(e) \\ & \mid & \mathsf{throw} \mid \mathsf{try}\ \{e_1\}\ \mathsf{catch}\ \{e_2\} \\ T & ::= & [\cdot] \mid T\ e \mid v\ T \mid (T, e) \mid (v, T) \mid \mathsf{proj}_i(T) \\ & \mid & \mathsf{inl}(T) \mid \mathsf{inr}(T) \\ & \mid & (\mathsf{match}\ T\ \mathsf{with} \mid \mathsf{inl}(x) \Rightarrow e_1 \mid \mathsf{inr}(y) \Rightarrow e_2\ \mathsf{end}) \\ & \mid & \mathsf{label}_\ell(T) \mid \mathsf{unlabel}\ T\ \mathsf{as}\ x\ \mathsf{in}\ e \mid \mathsf{write}(T) \end{array}$$

read returns the value of type σ currently stored in the only state cell, while write(e) replaces that value with e and returns unit. The term throw throws an exception, which propagates through contexts until it either hits top-level or a try block. We use a throw context T to implement this propagation. T is identical to evaluation contexts except it does not include try blocks. Finally, try $\{e_1\}$ catch $\{e_2\}$ runs e_1 until it returns either a value or an exception. If it returns a value v, then the try-catch returns v as well. If it throws an exception, however, the try-catch instead discards the exception and runs e_2.

These considerations give rise to the following typing rules:

$$\frac{}{\Gamma \vdash \mathsf{read} : \sigma} \qquad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \mathsf{write}(e) : \mathsf{unit}}$$

$$\frac{}{\Gamma \vdash \mathsf{throw} : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{try}\ \{e_1\}\ \mathsf{catch}\ \{e_2\} : \tau}$$

The resulting operational semantic rules are defined on pairs of an expression and a state cell s of type σ. The rules for our effectful operations are as follows:

$$\langle \mathsf{read}, s\rangle \longrightarrow \langle s, s\rangle \qquad \langle \mathsf{write}(v), s\rangle \longrightarrow \langle (), v\rangle$$

$$\langle T[\mathsf{throw}], s\rangle \longrightarrow \langle \mathsf{throw}, s\rangle$$

$$\langle \mathsf{try}\ \{v\}\ \mathsf{catch}\ \{e\}, s\rangle \longrightarrow \langle v, s\rangle$$

$$\langle \mathsf{try}\ \{\mathsf{throw}\}\ \mathsf{catch}\ \{e\}, s\rangle \longrightarrow \langle e, s\rangle$$

We modify the rest of the operational semantics by including s without modification in every other rule, as is standard for simply-typed λ-calculus with state [16, Chapter 13.3].

Technically, our previous noninterference theorem (Theorem 1) still holds, and by the same proof. However, the statement of this theorem is now very weak: it assumes that an attacker cannot see the state or distinguish throw from any other statement. This allows implicit flows. To see how exceptions allow implicit flows, consider the following example program that leaks its input to anyone who can distinguish throw from ():

$$h : L_\ell(\mathsf{unit} + \mathsf{unit}) \vdash \begin{array}{l} \mathsf{unlabel}\ h\ \mathsf{as}\ x \\ \mathsf{in\ match}\ x\ \mathsf{with} \\ \quad \mid \mathsf{inl}(\_) \Rightarrow \mathsf{throw} \\ \quad \mid \mathsf{inr}(\_) \Rightarrow \mathsf{label}_\ell() \\ \mathsf{end} \end{array} : L_\ell(\mathsf{unit})$$

A. Ruling Out Implicit Flows

We now aim to eliminate implicit flows and recover a strong notion of noninterference with realistic assumptions about what

[1] Allowing higher-order state is possible, but it requires recursive types and complicates reasoning about termination.

$$\begin{array}{lrcl} \text{Labels} & \ell & \in & \mathcal{L} \\ \text{Types} & \tau & ::= & \mathsf{unit} \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid L_\ell(\tau) \\ \text{Values} & v & ::= & () \mid \mathsf{inl}(v) \mid \mathsf{inr}(v) \mid (v_1, v_2) \mid \lambda x : \tau.\, e \mid \mathsf{label}_\ell(v) \\ \text{Expressions} & e & ::= & x \mid () \mid \lambda x : \tau.\, e \mid e_1\ e_2 \mid (e_1, e_2) \mid \mathsf{proj}_1(e) \mid \mathsf{proj}_2(e) \\ & & \mid & \mathsf{inl}(e) \mid \mathsf{inr}(e) \mid (\mathsf{match}\ e\ \mathsf{with} \mid \mathsf{inl}(x) \Rightarrow e_1 \mid \mathsf{inr}(y) \Rightarrow e_2\ \mathsf{end}) \\ & & \mid & \mathsf{label}_\ell(e) \mid \mathsf{unlabel}\ e_1\ \mathsf{as}\ x\ \mathsf{in}\ e_2 \end{array}$$

Fig. 1. Grammar for DCC Types and Terms

an attacker's power. We achieve this result by changing our typing rules. We associate a *program-counter label* $\mathrm{pc} \in \mathcal{L}$ with the typing judgment to track the sensitivity of the context. Thus, the typing judgment of this new type system takes the form $\Gamma \diamond \mathrm{pc} \vdash e : \tau$ where pc is an information-flow label.

In the examples of implicit flows we have seen so far, vulnerabilities arose when we performed certain actions depending on the value of a secret expression. To prevent such problems, we might update pc so that it is at least as high as any value we conditioned on in a match statement. We then ensure that actions that might leak information about those values cannot type-check in a sensitive environment. However, there is a problem with doing this in DCC: we never match on labeled data. Instead, we must first use unlabel, removing the label from the data before we can use that data in any way, including in a match expression. This is because DCC is a *coarse-grained* information-flow language. (In fact, it is the paradigmatic coarse-grained information-flow language.)

The fact that labeled data can only be used in an unlabel expression means that the unlabel rule is the only rule in which we can reasonably increase the pc. (This may seem like a significant restriction, since we are increasing the program-counter label even when we do not match on the data we are unlabeling. However, recent research has shown that coarse-grained information flow is equivalent to fine-grained information flow, which would increase the program-counter label in the match statement [17].) We increase the program counter label using the join operator on labels we discussed in Section II. Thus, the rule for unlabel becomes the following:

$$\frac{\Gamma \diamond \mathrm{pc} \vdash e_1 : L_\ell\,(\tau_1) \qquad \Gamma, x : \tau_1 \diamond \mathrm{pc} \sqcup \ell \vdash e_2 : \tau_2 \qquad \ell \lhd \tau_2}{\Gamma \diamond \mathrm{pc} \vdash \mathsf{unlabel}\ e_1\ \mathsf{as}\ x\ \mathsf{in}\ e_2 : \tau_2}$$

We must also change the typing rules for functions. Intuitively, an expression $\lambda x : \tau.\, e$ will execute the actions of e when it is used, not when it is defined. It is therefore safe to *construct* a λ-expression in any context, but it is only safe to *apply* one in a context where its effects do not leak information. Since we cannot, in general, know where a function will be used when it is constructed, we instead change the type of the function to restrict where it can be applied. The type $\tau_1 \xrightarrow{\mathrm{pc}} \tau_2$ is a function that takes an argument of type τ_1, returns a value of type τ_2, and can be safely run in contexts which have not discriminated on anything higher than pc. This gives rise to the following two typing rules:

$$\frac{\Gamma, x : \tau_1 \diamond \mathrm{pc}_1 \vdash e : \tau_2}{\Gamma \diamond \mathrm{pc}_2 \vdash \lambda x : \tau_1.\, e : \tau_1 \xrightarrow{\mathrm{pc}_1} \tau_2}$$

$$\frac{\Gamma \diamond \mathrm{pc}_1 \vdash e_1 : \tau_1 \xrightarrow{\mathrm{pc}_2} \tau_2 \qquad \Gamma \diamond \mathrm{pc}_1 \vdash e_2 : \tau_1 \qquad \mathrm{pc}_1 \sqsubseteq \mathrm{pc}_2}{\Gamma \diamond \mathrm{pc}_1 \vdash e_1\ e_2 : \tau_2}$$

This change also necessitates adjusting the function protection rule from Figure 2. Applying a function with type $\tau_1 \xrightarrow{\mathrm{pc}} \tau_2$ can still reveal data through its output at the level of τ_2, but it can also reveal information about control flow up to label pc. We therefore need to protect pc as well, leading to the following modified protection rule.

$$\frac{\ell \lhd \tau_2 \qquad \ell \sqsubseteq \mathrm{pc}}{\ell \lhd \tau_1 \xrightarrow{\mathrm{pc}} \tau_2}$$

To determine how the pc label should relate to effects, we need to be clear about what effects the attacker can and cannot see. We assume that anyone who can see things labeled ℓ_{State} can see the value stored in the state cell. An attacker who can read ℓ_{State} can see writes to state—since writes can change the stored value—but not reads—since reads leave the value unchanged. This is a reasonable assumption in many cases, but not all. For instance, it assumes the attacker cannot extract information through cache-based timing attacks [e.g., 18]. We also assume that any attacker who can see information labeled ℓ_{Exn} can distinguish between exceptions and other values. An attacker who *cannot* distinguish is therefore not privy to the success or error status of the program, meaning they also cannot see the result if it returns successfully. They may, however, still be able to observe the state cell if they can read ℓ_{State}.

We can use this model to determine the pc-based typing rules for our effectful operations. Three of the rules are fairly simple. The read rule allows any pc, while the write and throw rules must check that the context is not too sensitive to run this computation.

$$\frac{}{\Gamma \diamond \mathrm{pc} \vdash \mathsf{read} : \sigma} \qquad \frac{\Gamma \diamond \mathrm{pc} \vdash e : \sigma \qquad \mathrm{pc} \sqsubseteq \ell_{\mathsf{State}}}{\Gamma \diamond \mathrm{pc} \vdash \mathsf{write}(e) : \mathsf{unit}}$$

$$\frac{\mathrm{pc} \sqsubseteq \ell_{\mathsf{Exn}}}{\Gamma \diamond \mathrm{pc} \vdash \mathsf{throw} : \tau}$$

The try-catch rule is slightly more complicated, since the catch block only executes if the try block throws an exception, and therefore may return different values depending on whether or not an exception occurs. To ensure this control flow does not leak data, the output must be at least as sensitive as the control flow: ℓ_{Exn}.

$$\frac{\Gamma \diamond \mathrm{pc} \vdash e_1 : \tau \qquad \Gamma \diamond \mathrm{pc} \vdash e_2 : \tau \qquad \ell_{\mathsf{Exn}} \lhd \tau}{\Gamma \diamond \mathrm{pc} \vdash \mathsf{try}\ \{e_1\}\ \mathsf{catch}\ \{e_2\} : \tau}$$

None of the other rules change pc, since none of the other rules can leak information about the context or change the context based on a labeled value. These rules may appear to ensure security against the attacker we sketched above, but unfortunately this intuition misses the fact that exceptions can impact control flow outside just try-catch blocks. Consider the following program:

$$\begin{array}{l} h : L_{\ell_{\mathsf{Exn}}}\,(\mathsf{unit} + \mathsf{unit}), s : \sigma \vdash \\ \quad \mathsf{let}\ \_ = \mathsf{unlabel}\ h\ \mathsf{as}\ x \\ \quad\quad \mathsf{in\ match}\ x\ \mathsf{with} \\ \quad\quad\quad \mid \mathsf{inl}(\_) \Rightarrow \mathsf{throw} \\ \quad\quad\quad \mid \mathsf{inr}(\_) \Rightarrow \mathsf{label}_{\ell_{\mathsf{Exn}}}\,() \\ \quad\quad \mathsf{end} \\ \quad \mathsf{in}\ \mathsf{write}(s) \\ : \mathsf{unit} \end{array} \tag{1}$$

If $h = \mathsf{label}_{\ell_{\mathsf{Exn}}}(\mathsf{inl}\ ())$, this will throw an exception and the write will never execute. Notably, this program leaks the value of h to anyone who can see ℓ_{State}. Pottier and Simonet [5] eliminate this leak by constraining what effects can execute after an expression that may throw an exception. We take a more restrictive but far simpler approach and require that $\ell_{\mathsf{Exn}} \sqsubseteq \ell_{\mathsf{State}}$.

Proving that we have successfully eliminated data leaks requires using a notion of ℓ-equivalence that accounts for exceptions and state. Notably, to properly capture which attackers may view which values, our notion differs depending on how ℓ relates to ℓ_{State} and ℓ_{Exn}.

Definition 2. We say two expressions e_1 and e_2 are *state and exception ℓ-equivalent*, denoted $e_1 \cong_\ell^{\mathrm{SE}} e_2$, if $\Gamma \diamond \mathrm{pc} \vdash e_1 : \tau$ and $\Gamma \diamond \mathrm{pc} \vdash e_2 : \tau$, and for all substitutions $\theta : \Gamma$, programs e such that $\Gamma, x : \tau \diamond \mathrm{pc}' \vdash e : L_\ell(\mathsf{unit} + \mathsf{unit})$, and all values s where $\vdash s : \sigma$, if $\langle \theta(e[x \mapsto e_1]), s\rangle \longrightarrow^* \langle v_1, s_1\rangle$ and $\langle \theta(e[x \mapsto e_2]), s\rangle \longrightarrow^* \langle v_2, s_2\rangle$, then $v_1 = v_2$ if $\ell_{\mathsf{Exn}} \sqsubseteq \ell$ and $s_1 = s_2$ if $\ell_{\mathsf{State}} \sqsubseteq \ell$.

Intuitively, Definition 2 says that e_1 and e_2 are equivalent if no program will let an attacker at label ℓ distinguish them through either the program output or the state. Because the attacker can only see the program output if $\ell_{\mathsf{Exn}} \sqsubseteq \ell$, we only check output equivalence in that case. Similarly, because the attacker can only see the state cell if $\ell_{\mathsf{State}} \sqsubseteq \ell$, we only check equivalence of the state cells when the flow holds. Note that structural equality on state values is sufficient because we assumed σ contains no function types as subterms. Note also that e_1 and e_2 type under pc, while e types with an unrelated pc', allowing it to have an entirely different set of effects.

While it is possible to directly prove our type system enforces noninterference using such an equivalence relation [8], [19], [20], we take a different approach. We formalize the folklore view of the pc as a bound on the effects in a computation. This point-of-view simplifies the noninterference proof and helps avoid the need for clever ad-hoc reasoning, such as the argument we made for the try-catch rule.

B. Tracking Effects

To show that the pc label is a bound on effects, we need to track the effects in a program. To do so, we use a standard type-and-effect system [21], [22], [23], [24]. This type-and-effect system assigns each step of a typing proof an effect ε from a set $\mathcal{E}$ of possible effects. Therefore, we need to decide on the contents of $\mathcal{E}$.

So far, we have described our language has having two effects: state and exceptions. We might therefore let $\mathcal{E} = \{\mathrm{S}, \mathrm{E}\}$ where S represents state and E exceptions.

This choice raises two concerns. First, consider the following program (where the state is of type $\mathsf{unit} + \mathsf{unit}$):

$$x : \mathsf{unit} + \mathsf{unit} \vdash \begin{array}{l}\mathsf{match}\ x\ \mathsf{with}\\ \quad \mid \mathsf{inl}(\_) \Rightarrow \mathsf{read}\\ \quad \mid \mathsf{inr}(\_) \Rightarrow \mathsf{throw}\\ \mathsf{end}\end{array} : \mathsf{unit} + \mathsf{unit}$$

This program could either read state or throw an exception. So which effect should we give it? We must note that both are possible, which we can do by having an effect that represents "can use state and/or throw an exception." In fact, we need an effect for each possible *collection* of our effects. Thus our possible effects come from the power set, so $\mathcal{E} = 2^{\{\mathrm{S},\mathrm{E}\}}$. Notably, this gives our effects a nice lattice structure, as is standard for a power set.

Second, considering state as a single effect does not allow us to represent that our attacker can only see writes. For instance, consider the following program:

$$\begin{array}{l} h : L_\ell(\mathsf{unit} + \mathsf{unit}) \vdash \\ \qquad \begin{array}{l}\mathsf{unlabel}\ h\ \mathsf{as}\ x\\ \mathsf{in}\ \mathsf{match}\ x\ \mathsf{with}\\ \qquad \mid \mathsf{inl}(\_) \Rightarrow \mathsf{label}_\ell(\mathsf{read})\\ \qquad \mid \mathsf{inr}(\_) \Rightarrow \mathsf{label}_\ell(\mathsf{inl}\ ())\\ \quad \mathsf{end}\end{array}\\ \quad : L_\ell(\mathsf{unit} + \mathsf{unit}) \end{array}$$

An attacker who cannot distinguish values labeled ℓ cannot distinguish which branch the program takes. However, if we were to keep state as a single effect, we would have to label that branch as having the state effect, and therefore disallow it. To resolve this problem, we separate state operations into read effects R and write effects W and change $\mathcal{E}$ to $2^{\{\mathrm{R},\mathrm{W},\mathrm{E}\}}$. Note that since our attacker cannot see reads at all, we could consider read to be a pure operation. Though this would simplify a few technical details, we find it is more intuitive to include R as an effect.

Now we can develop our type-and-effect system. We again change the form of the typing judgment, this time to $\Gamma \vdash e : \tau \diamond \varepsilon$, where $\varepsilon \subseteq \{\mathrm{R}, \mathrm{W}, \mathrm{E}\}$. For readability, we will often write this set without curly brackets. The example program above that could read state or throw an exception would therefore have the typing judgment $\Gamma \vdash e : L_\ell(\mathsf{unit} + \mathsf{unit}) \diamond \mathrm{R}, \mathrm{E}$.

Since we are trying to analyze the security of the program, we create a function $\ell\_$ from effects to labels, associating a label ℓ_ε with each effect ε. This label corresponds to our attacker model of who can observe the effect. For W and E we already have these labels—ℓ_{State} and ℓ_{Exn}, respectively. Because reads are not visible, we can set $\ell_{\mathrm{R}} = \top$. For other effects, the label ℓ_ε should capture who might observe *any* component of ε, so it should be a lower bound on the components of ε. That is, $\ell_{\mathrm{W},\mathrm{E}} \sqsubseteq \ell_{\mathsf{State}}$ and $\ell_{\mathrm{W},\mathrm{E}} \sqsubseteq \ell_{\mathsf{Exn}}$.

Now we can build our type-and-effect system, modifying the rules of the pure typing system. As in the pc case, most of the typing rules do not change ε, since most terms do not change the effects a program may run. For the same reasons as before, functions and the four rules we added explicitly for effects do change. Since λ-expressions execute effects when applied but not defined, we take the same approach as before and record the effects in the type. We also modify the function protection rule as we did previously. This gives rise to the

following rules:

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \diamond \varepsilon}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \xrightarrow{\varepsilon} \tau_2 \diamond \varnothing} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\varepsilon_1} \tau_2 \diamond \varepsilon_2 \quad \Gamma \vdash e_2 : \tau_1 \diamond \varepsilon_3}{\Gamma \vdash e_1\ e_2 : \tau_2 \diamond \varepsilon_1 \cup \varepsilon_2 \cup \varepsilon_3}$$

$$\frac{\ell \lhd \tau_2 \qquad \ell \sqsubseteq \ell_\varepsilon}{\ell \lhd \tau_1 \xrightarrow{\varepsilon} \tau_2}$$

The rules for our extended expressions must incur appropriate effects. Again, three are fairly simple.

$$\frac{}{\Gamma \vdash \mathsf{read} : \sigma \diamond \mathrm{R}} \qquad \frac{\Gamma \vdash e : \sigma \diamond \varepsilon}{\Gamma \vdash \mathsf{write}(e) : \mathsf{unit} \diamond \varepsilon \cup \mathrm{W}}$$

$$\frac{}{\Gamma \vdash \mathsf{throw} : \tau \diamond \mathrm{E}}$$

Reading a value gives an R effect, while $\mathsf{write}(e)$ evaluates e and therefore any effects e runs, and then runs a write effect. Throwing an exception creates an E effect, but as before, catching an exception is more complicated. First, a try-catch expression does not generate any new effects, though it can combine effects from both the try and catch blocks. More importantly, since we are still aiming to enforce security through our type-and-effect system, the security concerns surrounding control flow still apply. We therefore again require $\ell_{\mathsf{Exn}} \lhd \tau$. The resulting rule is as follows.

$$\frac{\Gamma \vdash e_1 : \tau \diamond \varepsilon_1 \cup \mathrm{E} \qquad \Gamma \vdash e_2 : \tau \diamond \varepsilon_2 \qquad \ell_{\mathsf{Exn}} \lhd \tau}{\Gamma \vdash \mathsf{try}\ \{e_1\}\ \mathsf{catch}\ \{e_2\} : \tau \diamond \varepsilon_1 \cup \varepsilon_2}$$

As several of these rules require precisely equal effects, we include a rule allowing judgments to overstate a program's effect:

$$\frac{\Gamma \vdash e : \tau \diamond \varepsilon' \qquad \varepsilon' \subseteq \varepsilon}{\Gamma \vdash e : \tau \diamond \varepsilon}$$

We now have the mechanics we need to formally state a connection between the program-counter label and effects. Intuitively, a program with a sufficiently restrictive pc cannot have certain effects. We capture this intuition with the following lemma:

Lemma 1 (Connection between Program-Counter Label and Effects)**.** *The program-counter label forces effects to be well typed, so if* $\Gamma \diamond \mathrm{pc} \vdash e : \tau$*, then* $\Gamma \vdash e : \tau \diamond \varepsilon$ *for some* ε*. Moreover, the* pc *controls which effects are possible. In particular:*

- *If* $\mathrm{pc} \not\sqsubseteq \ell_{\mathsf{State}}$ *and* $\Gamma \diamond \mathrm{pc} \vdash e : \tau$*, then there exists an* ε *such that* $\Gamma \vdash e : \tau \diamond \varepsilon$ *and* $\mathrm{W} \notin \varepsilon$*.*
- *If* $\mathrm{pc} \not\sqsubseteq \ell_{\mathsf{Exn}}$ *and* $\Gamma \diamond \mathrm{pc} \vdash e : \tau$*, then there exists an* ε *such that* $\Gamma \vdash e : \tau \diamond \varepsilon$ *and* $\mathrm{E} \notin \varepsilon$*.*

Note that the two type systems use slightly different type constructors for functions—the pc system includes labels while the type-and-effect system includes effects. We implicitly convert between the two here, as the label associated with each effect makes the conversion simple.

Lemma 1 provides a direct connection between program-counter labels and effects. From the point-of-view we have been advocating—that program-counter labels give a lower bound on effects—this is the semantics of a program-counter label. We will discuss this semantics for program-counter labels in more depth in Section V.

C. Effectful Noninterference Half-Off

We now aim to use our effect tracking to prove noninterference for our extended language. We use a strategy from work on the semantics of type-and-effect systems and give meaning to effects via translation into pure programs. Importantly, if we do this in such a way that our notions of "low-equivalent" match up, we can get noninterference automatically.

Our translation uses monads to represent effects, as is common. In fact, we have one monad per (set of) effect(s), defined as follows. For technical reasons, we use the same monad for $\{\mathrm{W}\}$ and $\{\mathrm{R}, \mathrm{W}\}$, as well as for $\{\mathrm{W}, \mathrm{E}\}$ and $\{\mathrm{R}, \mathrm{W}, \mathrm{E}\}$.[2] We also assume for simplicity that σ, the type of our state cell, already has at least ℓ_{State} sensitivity—that is, $\ell_{\mathsf{State}} \lhd \sigma$—allowing us to omit explicit use of ℓ_{State}. If this were not the case, we would use $L_{\ell_{\mathsf{State}}}(\sigma)$ instead of σ.

ε	$P_\varepsilon(\tau)$
$\varnothing$	τ
$\{\mathrm{R}\}$	$\sigma \to \tau$
$\{\mathrm{E}\}$	$L_{\ell_{\mathsf{Exn}}}(\mathsf{unit} + \tau)$
$\{\mathrm{W}\}$ and $\{\mathrm{R}, \mathrm{W}\}$	$\sigma \to (\tau \times \sigma)$
$\{\mathrm{R}, \mathrm{E}\}$	$\sigma \to L_{\ell_{\mathsf{Exn}}}(\mathsf{unit} + \tau)$
$\{\mathrm{W}, \mathrm{E}\}$ and $\{\mathrm{R}, \mathrm{W}, \mathrm{E}\}$	$\sigma \to (L_{\ell_{\mathsf{Exn}}}(\mathsf{unit} + \tau) \times \sigma)$

We refer to the monad for a (set of) effect(s) ε as P_ε. Use this notation instead of the more-traditional $M_\_$ as we will generalize to a productor in Section V. These monads are standard, but they are not automatic. Instead, they reflect some choices about the semantics of programs. For instance, the fact that the monad for the set $\{\mathrm{R}, \mathrm{W}, \mathrm{E}\}$ is $\sigma \to (\mathsf{unit} + \tau) \times \sigma$ instead of $\mathsf{unit} + (\sigma \to \tau \times \sigma)$ reflects the fact that state persists even when an exception is thrown.

Note that we can define three programs:

- For any set ε and program $\Gamma, x : \tau_1 \vdash p : P_\varepsilon(\tau_2)$, we can define $\Gamma, x : P_\varepsilon(\tau_1) \vdash \mathsf{bind}_\varepsilon(p) : P_\varepsilon(\tau_2)$.
- For any type τ and set ε, we can define $\eta_\varepsilon : \tau \to P_\varepsilon(\tau)$.
- For any type τ and pair of sets ε_1 and ε_2 such that $\varepsilon_1 \subseteq \varepsilon_2$, we can define a program $\mathsf{coerce}_{\varepsilon_1 \mapsto \varepsilon_2} : P_{\varepsilon_1}(\tau) \to P_{\varepsilon_2}(\tau)$.

This (along with some easily-proven properties of these programs) makes $P_\_$ an *indexed monad* [25], [26], which is a mathematical object that gives semantics to systems of effects. Note that this requires $L_\_$ itself to be an indexed monad, which was proven by Abadi et al. [4]. As the name suggests, indexed monads are a generalization of monads. Indexed monads also give a standard way to translate effectful programs into pure programs, transforming a proof $\Gamma \vdash e : \tau \diamond \varepsilon$ into a program e' such that $\Gamma \vdash e' : P_\varepsilon(\tau)$.

[2] This is because the *writer* monad, for the write effect without read, is not a monad unless σ is a monoid. However, the *state* monad, for read and write effects, is a cartesian monad no matter what σ is, so we use that.

This translation generates an important security result. Because all well-typed pure programs guarantee noninterference, we can extend that result to our effectful language if our translation has two specific properties. First, it must be sound. Indeed, if we omit any of the careful reasoning about exceptions from Section III-B our translation would be unsound. The point at which the soundness proof breaks down is, however, often informative. For example, when translating Program 1, our translation would need to return a value of type $L_{\ell_{\mathsf{Exn}}}(\mathsf{unit}+\mathsf{unit}) \times \sigma$ after unlabeling a value of type $L_{\ell_{\mathsf{Exn}}}(\mathsf{unit}+\mathsf{unit})$. Satisfying the premise of the unlabel rule that the removed label protects the output type forces exactly the $\ell_{\mathsf{Exn}} \sqsubseteq \ell_{\mathsf{State}}$ assumption we made above.

The second requirement to obtain effectful noninterference is that our monadic translation faithfully translates our effectful notion of equivalence to our pure one. In this case it does because monadic programs simulate effectful programs. Without labels, this is a well-known theorem of Wadler and Thiemann [25], adding labels does not significantly change the proof. We can therefore use our type-and-effect system and monadic translation to make a strong claim of noninterference for the pc system.

Theorem 2 (Noninterference for State and Exceptions)**.** *For expressions e_1 and e_2 and $\ell \in \mathcal{L}$, if $\Gamma \diamond \mathrm{pc} \vdash e_1 : \tau$ and $\Gamma \diamond \mathrm{pc} \vdash e_2 : \tau$, and $\ell \lhd \tau$ and $\ell \sqsubseteq \mathrm{pc}$, then for all labels $\ell_{\mathsf{Atk}} \in \mathcal{L}$, either $\ell \sqsubseteq \ell_{\mathsf{Atk}}$ or $e_1 \cong^{\mathrm{SE}}_{\ell_{\mathsf{Atk}}} e_2$.*

Proof. This is a special case of the Noninterference Half-Off Theorem (Theorem 5) which uses the fact that effectful programs are equivalent if their monadic translations are equivalent. Theorem 5 also relies on the fact that pure programs are noninterfering (Theorem 1). □

The requirement that $\ell \sqsubseteq \mathrm{pc}$ may appear odd next to classic definitions of noninterference. We require such a flow because our contextual notion of equivalence treats e_1 and e_2 as program inputs, but they may have effects. This requirement constrains those effects so that if $\ell \not\sqsubseteq \ell_{\mathsf{Atk}}$, then ℓ_{Atk} will be unable to see them. In most classic noninterference statements, the inputs are values, meaning this restriction is unnecessary as any well-typed value type-checks with $\mathrm{pc} = \top$.

IV. Example: Termination-Sensitive Noninterference

Type-and-effect systems can tell us more about programs than whether they access state or throw exceptions. One classic application is checking termination by considering possible nontermination to be an effect. From our perspective on program counters, this application means they can control whether a possibly-nonterminating program may run in a context. This gives rise to *termination-sensitive noninterference*, where an attacker may observe whether a computation has failed to terminate or not.

The fragment of DCC we use in Section II is strongly-normalizing; thus, all programs terminate. Even the extensions in Section III did not allow for nonterminating behavior, since we require that the type of the state cell is lower-order. We can, however, easily add a standard fixpoint operator:

$$e \quad ::= \quad \ldots \mid \mathsf{fix}\ f : \tau.\, e$$

$$\frac{\Gamma, f : \tau \vdash e : \tau}{\Gamma \vdash \mathsf{fix}\ f : \tau.\, e : \tau} \qquad \textsc{Fix}\ \mathsf{fix}\ f : \tau.\, e \to e[f \mapsto \mathsf{fix}\ f : \tau.\, e]$$

Because programs may not terminate in this extended language, the fact that our definition of ℓ-equivalence (Definition 1) allows for different termination behavior matters. In particular, our previous noninterference theorem (Theorem 1) now says that if both programs terminate, they must produce the same value. If *either* program diverges, however, it makes no guarantees.

This guarantee models an attacker who cannot tell if a program has failed to terminate, or if it will produce an output on the next step. That is, the attacker is *insensitive* to termination behavior. This notion of noninterference is therefore called *termination-insensitive* noninterference.

While DCC with the call-by-value semantics we are using enforces termination-insensitive noninterference [3], [4], we would like to remove the strong assumption that attackers infer nothing from divergent programs. We thus define a stronger form of equivalence, *termination-sensitive ℓ-equivalence*, and use it to analyze security.

Definition 3 (Termination-Sensitive ℓ-Equivalence)**.** We say e_1 is *termination-sensitive ℓ-equivalent* to e_2, denoted $e_1 \cong^{\mathrm{TS}}_{\ell} e_2$, if $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$, and for any substitution $\theta : \Gamma$ and any program e such that $\Gamma, x : \tau \vdash e : L_\ell(\mathsf{unit}+\mathsf{unit})$, it is the case that $\theta(e[x \mapsto e_1]) \longrightarrow^* v$ if and only if $\theta(e[x \mapsto e_2]) \longrightarrow^* v$.

We follow the same approach as in Section III to ensure noninterference with respect to this stronger definition: we restrict effects with a pc label, build a corresponding type-and-effect system, and prove security by translating to DCC. Because nontermination from a fixed point is the only effect, this change is considerably simpler than last time.

Traditionally, termination-sensitive noninterference assumes all attackers can see whether or not a program terminates. We take a more general approach and assume that some attackers are termination-sensitive, while others may not be. Specifically, we imagine there is a label $\ell_{\mathsf{PNT}} \in \mathcal{L}$ such that any attacker who can read ℓ_{PNT} will eventually infer information from nontermination, but others will not. This gives rise to the following rule:

$$\frac{\Gamma, f : \tau \diamond \mathrm{pc} \vdash e : \tau \qquad \mathrm{pc} \sqsubseteq \ell_{\mathsf{PNT}}}{\Gamma \diamond \mathrm{pc} \vdash \mathsf{fix}\ f : \tau.\, e : \tau}$$

The rule ensures that only data available at label ℓ_{PNT}—visible to any termination-sensitive attacker—can influence the program's termination behavior. This single label-based rule allows us to model termination-insensitivity by setting $\ell_{\mathsf{PNT}} = \top$, model traditional termination-sensitivity using a bottom label $\bot$ by setting $\ell_{\mathsf{PNT}} = \bot$, or express policies about other levels of termination visibility.

We can now move on to the type-and-effect system. This time it has only two possible effects: $\varnothing$ or PNT with labels $\top$ and ℓ_{PNT}, respectively. The typing rule for the fixed-point operator is then:

$$\frac{\Gamma, f : \tau \vdash e : \tau \diamond \varepsilon}{\Gamma \vdash \mathsf{fix}\ f : \tau.\, e : \tau \diamond \mathrm{PNT}}$$

No other typing rules constrain or change effects, though functions change as in Section III.

Finally, we translate this type-and-effect system into pure DCC using a monadic translation. Unfortunately, this time there is no such monad definable in the language of Section II. Luckily, the original definition of DCC [4] allowed for nontermination with a fixed-point operator, but only if the result was a *pointed* type. We thus add fixed points and pointed types—the parts of DCC we omitted from Section II—and use pointed types to define our monad.

$$\begin{array}{rcl} \tau & ::= & \cdots \mid \tau_\bot \\ e & ::= & \cdots \mid \mathsf{lift}(e) \mid \mathsf{seq}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{fix}\ f : \tau.\, e \end{array}$$

The type $\tau_\bot$ represents a version of τ that supports fixed points, while the expression $\mathsf{lift}(e)$ lifts the expression e from type τ to $\tau_\bot$. The expression $\mathsf{seq}\ x = e_1\ \mathsf{in}\ e_2$ waits for e_1 to terminate and, if it does, binds the result to x in e_2. Finally, the term $\mathsf{fix}\ f : \tau.\, e$ defines a fixpoint.

We then define a judgment determining when a type is a pointed type as follows:

$$\frac{}{\vdash \tau_\bot\ \mathrm{ptd}} \qquad \frac{\vdash \tau_1\ \mathrm{ptd} \qquad \vdash \tau_2\ \mathrm{ptd}}{\vdash \tau_1 \times \tau_2\ \mathrm{ptd}}$$

$$\frac{\vdash \tau\ \mathrm{ptd}}{\vdash L_\ell(\tau)\ \mathrm{ptd}} \qquad \frac{\vdash \tau_2\ \mathrm{ptd}}{\vdash \tau_1 \to \tau_2\ \mathrm{ptd}}$$

This allows us to state the following typing and semantic rules for the newly-added terms:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{lift}(e) : \tau_\bot} \qquad \frac{\Gamma \vdash e_1 : \tau_{1\bot} \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \qquad \vdash \tau_2\ \mathrm{ptd}}{\Gamma \vdash \mathsf{seq}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

$$\frac{\Gamma, f : \tau \vdash e : \tau \qquad \vdash \tau\ \mathrm{ptd}}{\Gamma \vdash \mathsf{fix}\ f : \tau.\, e : \tau}$$

$$\mathsf{seq}\ x = \mathsf{lift}(v)\ \mathsf{in}\ e \longrightarrow e[x \mapsto v]$$

Because the possibility of nontermination is limited to pointed types, we consider the full DCC to be pure for our purposes. That is, we consider programs which are nonterminating, but which have a pointed type, pure. Because we encompass nontermination in pointed types (which form a monad), we are justified in considering them pure, just as we are justified in considering programs which carry state around explicitly pure. We can therefore use it as a translation target from the type-and-effect system we developed earlier to give that system semantics. However, the translation of the unlabel rule fails whenever a program returns a $\tau_\bot$, because $\tau_\bot$ protects no labels. This was Abadi et al.'s [4] original design, to ensure that labeled data can never determine the termination behavior of a program. This is too restrictive for us, since we want to allow data up to label ℓ_{PNT} to influence a program's termination behavior. Of course, the label must also be allowed to influence any output the program produces if it does terminate. This leads to the following rule:

$$\frac{\ell \sqsubseteq \ell_{\mathsf{PNT}} \qquad \ell \lhd \tau}{\ell \lhd \tau_\bot}$$

Note that, even when setting $\ell_{\mathsf{PNT}} = \bot$, this rule allows *public* data to influence termination behavior. We therefore differ slightly from Abadi et al.'s [4] definition in systems where public and unlabeled data are different.

With this rule, the traditional monadic translation works, which tells us that the type-and-effect system indeed enforces noninterference. Now we need to connect the pc system to the type-and-effect system in an analogous way to Lemma 1. If the pc cannot influence ℓ_{PNT}, then the program *must* terminate. We can formalize this as follows:[3]

Lemma 2. *If* $\Gamma \diamond \mathrm{pc} \vdash e : \tau$*, then* $\Gamma \vdash e : \tau \diamond \mathrm{PNT}$*. Moreover, if* $\mathrm{pc} \not\sqsubseteq \ell_{\mathsf{PNT}}$*, then* $\Gamma \vdash e : \tau \diamond \varnothing$*.*

Composing Lemma 2 with the monadic translation of the type-and-effect systems gives the following guarantee:

Theorem 3 (Termination-Sensitive Noninterference)**.** *For expressions* e_1 *and* e_2 *and* $\ell \in \mathcal{L}$*, if* $\Gamma \diamond \mathrm{pc} \vdash e_1 : \tau$ *and* $\Gamma \diamond \mathrm{pc} \vdash e_2 : \tau$*, and* $\ell \lhd \tau$ *and* $\ell \sqsubseteq \mathrm{pc}$*, then for all labels* $\ell_{\mathsf{Atk}} \in \mathcal{L}$ *where* $\ell_{\mathsf{PNT}} \sqsubseteq \ell_{\mathsf{Atk}}$*, either* $\ell \sqsubseteq \ell_{\mathsf{Atk}}$ *or* $e_1 \cong^{\mathrm{TS}}_{\ell_{\mathsf{Atk}}} e_2$*.*

This is a special case of Theorem 5. As with both previous noninterference theorems (Theorems 1 and 2), this theorem says data at label ℓ cannot leak to an attacker who cannot distinguish values at level ℓ. This time, however, the attacker can glean information from nontermination. Moreover, if we set $\ell_{\mathsf{PNT}} = \bot$, then $\bot \sqsubseteq \ell_{\mathsf{Atk}}$ for all ℓ_{Atk}, so if $\ell \neq \bot$ we get classic termination-sensitive noninterference.

Note that if either e_1 or e_2 may diverge, the theorem always allows a termination-sensitive attacker to distinguish them. In this case, Lemma 2 ensures $\mathrm{pc} \sqsubseteq \ell_{\mathsf{PNT}}$, which then guarantees $\ell \sqsubseteq \ell_{\mathsf{Atk}}$ by transitivity.

V. A Framework for Effectful Labeled Languages

We have now twice given semantics to pc systems using type-and-effect systems and monadic translations. The ability to analyze not just traditional effects like state, but combinations of effects and more unusual effects, like nontermination, demonstrates the power of this technique. We now generalize these results by examining what language properties are necessary to obtain the security results from Sections III and IV.

We therefore move to a semantic framework that does not lock us into a single language. Instead, we provide a set of typing rules and equations specifying the language features

[3] As with Lemma 1, we implicitly assume a simple translation between the two sets of type constructors.

$$\textsc{Seq}_{\leq}\ \frac{t_0 \vdash_{\mathrm{E}} p_1 \dashv t_1 \diamond \varepsilon_1 \quad \cdots \quad t_{n-1} \vdash_{\mathrm{E}} p_n \dashv t_n \diamond \varepsilon_n \qquad [\varepsilon_1, \ldots, \varepsilon_n] \leq \varepsilon}{t_0 \vdash_{\mathrm{E}} p_1 \,;\cdots;\, p_n \dashv t_n \diamond \varepsilon}$$

$$\textsc{Seq}\ \frac{\tau_0 \vdash_{\mathrm{P}} \rho_1 \dashv \tau_1 \quad \cdots \quad \tau_{n-1} \vdash_{\mathrm{P}} \rho_n \dashv \tau_n}{\tau_0 \vdash_{\mathrm{P}} \rho_1 \,;\cdots;\, \rho_n \dashv \tau_n}$$

$$\textsc{Capture}\ \frac{t \vdash_{\mathrm{E}} p \dashv t' \diamond \varepsilon}{(\!|t|\!) \vdash_{\mathrm{P}} \lfloor p \rfloor_{\varepsilon} \dashv P_{\varepsilon}(\!|t'|\!)}$$

$$\textsc{Map}\ \frac{\tau \vdash_{\mathrm{P}} \rho \dashv \tau'}{P_{\varepsilon}(\tau) \vdash_{\mathrm{P}} \mathsf{map}_{\varepsilon}(\rho) \dashv P_{\varepsilon}(\tau')} \qquad \textsc{Join}\ \frac{[\varepsilon_1, \ldots, \varepsilon_n] \leq \varepsilon}{P_{\varepsilon_1}(\cdots P_{\varepsilon_n}(\tau)\cdots) \vdash_{\mathrm{P}} \mathsf{join}_{[\varepsilon_1,\ldots,\varepsilon_n],\varepsilon} \dashv P_{\varepsilon}(\tau)}$$

$$\textsc{CapturedSeq}\ \frac{t_0 \vdash_{\mathrm{E}} p_1 \dashv t_1 \diamond \varepsilon_1 \quad \cdots \quad t_{n-1} \vdash_{\mathrm{E}} p_n \dashv t_n \diamond \varepsilon_n \qquad [\varepsilon_1, \ldots, \varepsilon_n] \leq \varepsilon}{\lfloor p_1 \,;\cdots;\, p_n \rfloor_{\varepsilon} \overset{=\!=\!=}{\underset{(\!|t_0|\!), P_{\varepsilon}(\!|t_n|\!)}{}} \lfloor p_1 \rfloor_{\varepsilon_1} \,;\, \mathsf{map}_{\varepsilon_1}(\lfloor p_2 \rfloor_{\varepsilon_2} \,;\, \mathsf{map}_{\varepsilon_2}(\cdots(\lfloor p_n \rfloor_{\varepsilon_n}))) \,;\, \mathsf{join}_{[\varepsilon_1,\cdots,\varepsilon_n],\varepsilon}}$$

$$\textsc{ProtectC}\ \frac{\ell \sqsubseteq \ell_{\varepsilon} \qquad \ell \lhd t}{\ell \lhd P_{\varepsilon}(\!|t|\!)} \qquad \textsc{EquivCap}\ \frac{\lfloor p \rfloor_{\varepsilon} \approx_{\ell} \lfloor q \rfloor_{\varepsilon}}{p \cong_{\ell}^{\varepsilon} q}$$

Fig. 3. Main Rules for Semantic Framework of Labeled Pure and Effectful Programming Languages

that are required for our analysis. This approach allows us to make strong guarantees about any language that has those properties, regardless of what other features may be present.

As with our examples, we have two languages, one effectful and one pure. The effectful language is defined with a type-and-effect system and judgments take the form $t \vdash_{\mathrm{E}} p \dashv t' \diamond \varepsilon$. Typing judgments for the pure language are simply $\tau \vdash_{\mathrm{P}} \rho \dashv \tau'$. For clarity, we annontate the turnstyles and color judgments in the two languages differently. We also use Roman letters to refer to types and programs in the effectful language and Greek letters in the pure language. Note that both judgments are single-input, single-output. This restriction follows Tate [13], who invented the structure we are discussing in this section.

To define the requisite structures on the effectful language we need a collection of effects $\mathcal{E}$. In Sections III and IV, $\mathcal{E}$ formed a lattice, allowing us to give meaning to sequential composition of programs with different effects. A lattice, however, is far more structure than is necessary. Here we mandate the minimal structure required to give meaning to such compositions: an *effector* [13]. An effector is a set $\mathcal{E}$ with a relation $[-, \ldots, -] \leq -$ defining how the effects can compose. Intuitively, $[\varepsilon_1, \ldots, \varepsilon_n] \leq \varepsilon$ means that sequentially composing n programs with effects ε_1 through ε_n, respectively, can result in a program with effect ε. Note that n may be zero or more, so $[] \leq \varepsilon$ means that a program judged to have effect ε may be pure, and $[\varepsilon] \leq \varepsilon'$ means that a program judged to have effect ε' may also have effect ε. The relation must follow appropriate versions of identity and associativity laws, reflecting these intuitions [13, Section 5].

Using effectors, we can define the requisite typing rules on each language (Figure 3). To begin, $\textsc{Seq}_{\leq}$ and $\textsc{Seq}$ require both languages to have a notion of n-way sequential composition. (When $n = 0$, we have an identity program which changes nothing: $t \vdash_{\mathrm{E}} \dashv t$.) In the effectful language, composition is allowed when the effects of the programs compose. In the pure language, composition is always allowed.

We also require a rule, $\textsc{Capture}$, that allows us to convert a program from the effectful language to the pure one.[4] We denote this translation $\lfloor - \rfloor_{\varepsilon}$. Note also that the pure and effectful languages may use different types. For instance, in our example languages, functions in the type-and-effect system were annotated by an effect, whereas pure functions were not. We therefore need a translation $(\!|-|\!)$ to translate effectful types to pure ones. This handles the translation of the annotated function type $\tau_1 \xrightarrow{\varepsilon} \tau_2$ to the pure, unannotated function type $\tau_1 \to P_{\varepsilon}(\tau)$ in our example languages.

Moreover, we need to ensure that $\lfloor p \rfloor_{\varepsilon}$ properly represents the effect ε of the program p. This is done by changing the output type to represent not only the outputs of p but also ε. This change must be a mathematical function of effect and type, which we write $P_{-}(-)$.

This second translation $P_{-}(-)$ must have two additional pieces of structure to allow us to faithfully translate sequential composition of effectful programs. First, we require a pure program transformer $\mathsf{map}_{\varepsilon}$ for every effect ε with the typing rule $\textsc{Map}$: it takes a program from τ to τ' and transforms it into a program from $P_{\varepsilon}(\tau)$ to $P_{\varepsilon}(\tau')$. Second, for any list of effects $\varepsilon_1, \ldots, \varepsilon_n$ and ε such that $[\varepsilon_1, \ldots, \varepsilon_n] \leq \varepsilon$, we require a pure program $\mathsf{join}_{[\varepsilon_1,\ldots,\varepsilon_n],\varepsilon}$ with typing rule $\textsc{Join}$: it takes an input of type $P_{\varepsilon_1}(\cdots P_{\varepsilon_n}(\tau)\cdots)$ and produces an output of type $P_{\varepsilon}(\tau)$.

$\textsc{CapturedSeq}$ says that we can define the capture of the sequential composition of effectful programs by capturing each program individually and using pure composition, map, and join appropriately. Here $\rho_1 \overset{=\!=}{\underset{\tau_1,\tau_2}{}} \rho_2$ means "ρ_1 and ρ_2 are equal as programs from τ_1 to τ_2."

$\textsc{Map}$ and $\textsc{Join}$ must also follow two simple rules, similar to the rules for effectors,[5] to ensure that the monad-like structure

[4]Tate [13] calls capture *thunking*, to bring attention to the similarity with the familiar concept in functional languages. We use the word "capture" here because it better fits how we use the concept.

[5]Essentially, (1) naturality of $\mathsf{join}_{[-],-}$, (2) $\mathsf{join}_{[\varepsilon],\varepsilon}$ is the identity, and (3) when $[\vec{\varepsilon_i}] \leq \varepsilon_i$ and $[\varepsilon_1, \ldots, \varepsilon_n] \leq \varepsilon$, the two evident ways of using $\mathsf{join}_{[-],-}$ to get from $P_{\vec{\varepsilon_1}}(-) \circ \cdots \circ P_{\vec{\varepsilon_n}}(-)$ to $P_{\varepsilon}(-)$ are equal.

$P_{\_}(-)$ forms a *productor* [13]. Note that productors are a generalization of monads. If we have only one effect ε such that $[\varepsilon, \varepsilon] \leq \varepsilon$ and $[] \leq \varepsilon$, then the above defines a monad. Productors also generalize *indexed* monads [25], [26] and *graded* monads [27], [28]. An indexed monad is a productor for a lattice, the structure we had in Sections III and IV, while a graded monad is a productor for an ordered monoid; both are examples of effectors.

Our framework up until this point is a slight variant of Tate's [13] framework for languages with a producer effect. While Tate's framework assumed a single language with both pure and effectful programs, we have assumed two languages, one pure and one effectful. Moreover, Tate did not assume MAP and JOIN directly, but rather assumed an effectful program which would execute the effects stored in a captured type. He then used this program to prove MAP and JOIN were admissible. We work in this more-general framework, which allows the language of Section IV to fit.

We also make a few assumptions about labels. We assume both the effectful and pure languages define protection relations, and we relate them with the PROTECTC rule. This rule says that if both an effectful type t and an effect ε have sensitivity at least ℓ, then $P_\varepsilon(\!|t|\!)$ must as well. Our examples admitted this rule by composing two simpler rules: protection transfers directly from the effectful to the pure type system, and if a *pure* type τ and an effect ε have sensitivity at least ℓ, then $\ell \lhd P_\varepsilon(\tau)$. Our results, however, only require the single combined rule.

Note that, while we assume that both languages define a protection relation, this does not have to be the main security mechanism of the language. Instead, it can be defined in terms of that security mechanism. For instance, a fine-grained system can define a protection relation using the fact that every type has a label associated with that type. Rajani and Garg [17] give an example of this sort of definition in their translation of a fine-grained system into a coarse-grained one.

Finally, as in our examples, noninterference is generally defined using notions of equivalence that reflected what an attacker could see. To transfer a noninterference result from the pure language to the effectful one, both languages need such notions of equivalence, and those notions must correspond to each other. In Section III, for example, effectful equivalence considered both the program output and the state cell. By making the state cell part of the term when translating to the pure language, we ensured that the pure equivalence gave rise to the effectful equivalence. In general, the EQUIVCAP rule requires that the pure notion of equivalence, $\approx_{\_}$, be at least as precise on captured programs as the effectful equivalence, $\cong^{\varepsilon}_{\_}$, is on uncaptured ones.

A categorically-minded reader may have noticed that our pure and effectful languages each define categories where the category of effectful programs is essentially the Kleisli category for a monad-like structure on the category of pure programs. Indeed, productors are the most general structure known for studying *producer effects*, which are effects that have a Kleisli-like structure [13].

Moreover, as we have noted, the typing rules for label and unlabel in our example languages make $L_{\_}(-)$ an indexed monad, which is also a type of productor. Interestingly, when the label, unlabel, and PROTECTC rules are all admissible, we can define something like a distributive law [29] between the label productor and the effect productor. This essentially allows us to lift the label productor to the effectful language—that is, to the Kleisli category for the effect productor.

This framework will allow us to prove a strong and general security theorem for effectful programming languages. Moreover, it connects us to the category theory, giving us powerful tools for reasoning about effects.

VI. THE NONINTERFERENCE HALF-OFF THEOREM

Our main aim is to show how, given a pc system, we can give semantics to the pc label and prove noninterference for that system. We do this by translation to a type-and-effect system, which we give semantics using the framework from Section V. This is a powerful and general result, but it requires us to first demonstrate the security of the type-and-effect system itself.

A. Type-and-Effect Noninterference

In Sections III and IV we leveraged the ability to compile effects into a pure language to simplify reasoning about noninterference. This allowed us to prove noninterference for the effectful language while only proving it directly for the pure part of the language. Since the framework of Section V specifies when this is possible, we would like to prove noninterference for any languages which admit the rules in Figure 3.

As in our examples, noninterference formalizes the intuition that adversary at label ℓ_{Atk} can only distinguish data and effects at or below label ℓ_{Atk}. We again define the label of data using a protection relation, though this time we leave the details of that relation abstract. We also assign a label ℓ_ε to the effect ε to represent ε's sensitivity. Finally, as each language has a different notion of equivalence, use an abstract notion of equivalence $\equiv_{\_}$ parameterized on labels.

Definition 4 (Abstract Noninterference)**.** Let r be a program such that $t_1 \vdash_{\mathrm{E}} r \dashv t_2 \diamond \varepsilon_1$. We say that r is *noninterfering with respect to* $\equiv_{\_}$ if, for all labels $\ell \in \mathcal{L}$ and programs p and q such that

1) $t_3 \vdash_{\mathrm{E}} p \dashv t_1 \diamond \varepsilon_2$ and $t_3 \vdash_{\mathrm{E}} q \dashv t_1 \diamond \varepsilon_2$ with $[\varepsilon_2, \varepsilon_1] \leq \varepsilon$
2) $\ell \lhd t_1$ and $\ell \sqsubseteq \ell_{\varepsilon_2}$

then for all labels $\ell_{\mathsf{Atk}} \in \mathcal{L}$, either $\ell \sqsubseteq \ell_{\mathsf{Atk}}$ or $p\,;r \equiv_{\ell_{\mathsf{Atk}}} q\,;r$.

In the above definition, condition 1 requires the sequential compositions $p\,;r$ and $q\,;r$ to be well-typed, while p and q each produce effects (at most) ε_2. This sequential composition represents providing two different inputs to the program r, abstracting the contextual equivalence we used in Sections III and IV. Condition 2 requires that the sensitivity of both the input data of type t_1 and the effects of the input programs, ε_2, be at least ℓ. Intuitively, the conclusion says that an attacker at ℓ_{Atk} can either read those inputs—$\ell \sqsubseteq \ell_{\mathsf{Atk}}$—or no program allows the attacker to distinguish p from q.

We also allow the definition to apply to pure programs, replacing type-and-effect judgements with pure judgements

$$\textsc{ProtectTrans}\ \frac{\ell \lhd t}{\ell \lhd [\![t]\!]} \qquad \textsc{PCEff}\ \frac{t \diamond \mathrm{pc} \vdash_{\mathrm{PC}} p \dashv t'}{\exists \varepsilon. [\![t]\!] \vdash_{\mathrm{E}} p \dashv [\![t']\!] \diamond \varepsilon}$$

Fig. 4. Additional Rules for pc systems

and disregarding other references to effects. Our framework's rules are then sufficient to transfer a noninterference result from pure programs to effectful ones.

Theorem 4 (Type-and-Effect Noninterference)**.** *For any system satisfying all rules in Figure 3 where every well-typed pure program is noninterfering with respect to $\approx_{\_}$, then every program well-typed in the type-and-effect system is noninterfering with respect to $\cong^{\varepsilon}_{\_}$.*

Proof. Unfolding Definition 4, we have programs p, q, and r and a label ℓ such that

1) $t_1 \vdash_{\mathrm{E}} r \dashv t_2 \diamond \varepsilon_1$,
2) $t_3 \vdash_{\mathrm{E}} p \dashv t_1 \diamond \varepsilon_2$ and $t_3 \vdash_{\mathrm{E}} q \dashv t_1 \diamond \varepsilon_2$ with $[\varepsilon_2, \varepsilon_1] \leq \varepsilon$,
3) $\ell \lhd t_1$ and $\ell \sqsubseteq \ell_{\varepsilon_2}$,

and we aim to show that for all labels $\ell_{\mathsf{Atk}} \in \mathcal{L}$, either $\ell \sqsubseteq \ell_{\mathsf{Atk}}$ or $p\,;r \cong^{\varepsilon}_{\ell_{\mathsf{Atk}}} q\,;r$.

Let $\rho = \mathsf{map}_{\varepsilon_2}(\lfloor r \rfloor_{\varepsilon_1})\,;\mathsf{join}_{[\varepsilon_2,\varepsilon_1],\varepsilon}$. The rules in Figure 3 guarantee $P_{\varepsilon_2}(\!|t_1|\!) \vdash_{\mathrm{P}} \rho \dashv P_{\varepsilon}(\!|t_2|\!)$ and CapturedSeq requires $\lfloor p\,;r \rfloor_{\varepsilon} \overset{}{\underset{(\!|t_3|\!),P_{\varepsilon}(\!|t_2|\!)}{=\!=\!=}} \lfloor p \rfloor_{\varepsilon_2}\,;\rho$, and similarly for q. All well-typed pure programs are noninterfering with respect to $\approx_{\_}$, ρ is pure, and $\ell \lhd P_{\varepsilon_2}(\!|t_1|\!)$ by ProtectC. Thus, for any label ℓ_{Atk}, either $\ell \sqsubseteq \ell_{\mathsf{Atk}}$ or $\lfloor p \rfloor_{\varepsilon_2}\,;\rho \approx_{\ell_{\mathsf{Atk}}} \lfloor q \rfloor_{\varepsilon_2}\,;\rho$. In the first case, we are finished. In the second case, the program equality above and EquivCap give us

$$\begin{aligned} \lfloor p \rfloor_{\varepsilon_2}\,;\rho \approx_{\ell_{\mathsf{Atk}}} \lfloor q \rfloor_{\varepsilon_2}\,;\rho &\iff \lfloor p\,;r \rfloor_{\varepsilon} \approx_{\ell_{\mathsf{Atk}}} \lfloor q\,;r \rfloor_{\varepsilon} \\ &\implies p\,;r \cong^{\varepsilon}_{\ell_{\mathsf{Atk}}} q\,;r. \end{aligned}$$

□

B. Semantics of Program Counter Labels

We can now use the semantic framework we have developed for effectful labeled programs and noninterference for type-and-effect systems to talk about the semantics and security of the pc label. We extend the framework to include a pc system with judgments of the form $t \diamond \mathrm{pc} \vdash_{\mathrm{PC}} p \dashv t'$. Figure 4 shows the rules we require for this extended framework.

We give semantics to the pc by formalizing the folklore understanding that it bounds the effects within a program. Specifically, we define the semantics of the pc system by requiring a translation of typing proofs in the pc system to typing proofs in the type-and-effect system. PCEff formalizes this requirement.

We would like this translation to preserve types, since this would formalize the intuition that we are giving semantics to the pc label. Unfortunately, in our examples, the pc systems and type-and-effect systems had different function types. The pc system included a label on its functions ($\tau_1 \xrightarrow{\mathrm{pc}} \tau_2$), while the type-and-effect system included an effect ($\tau_1 \xrightarrow{\varepsilon} \tau_2$). We therefore allow the pc system to have different types, but the same programs, and require there to be a translation $[\![-]\!]$ from the pc types to the type-and-effect types. (We still write both types and programs in the pc system with Roman letters.) This translation must preserve the sensitivity of the data, represented as the protection level, which we formalize as rule ProtectTrans.

These rules complete the requirements for our core theorem.

Theorem 5 (The Noninterference Half-Off Theorem)**.** *For any system satisfying all rules in Figures 3 and 4 where every well-typed pure program is noninterfering with respect to $\approx_{\_}$, then every program well-typed in the* pc *system is noninterfering with respect to $\cong^{\varepsilon}_{\_}$.*

Proof. This follows directly from PCEff and Theorem 4. We note that this always instantiates the ε in $\cong^{\varepsilon}_{\_}$ with the same ε used to type check r. □

Importantly, this lays out exactly the work necessary to prove security of a pc system. First, the designer must prove the admissibility of the rules in Figures 3 and 4. Then, they must prove conditions (*i*) and (*ii*). Luckily, (*i*) usually comes for free from adequacy of the translation; this is the case in all of our examples. Moreover, for a large number of effects, adequacy of translation has already been proven, and so this imposes no extra cost. While (*ii*) does require some extra proof work, it is usually straightforward.

VII. Related Work

This work pulls mostly from two distinct areas: static IFC and the theory of effects. We discuss related work from each of these areas in turn.

A. Static Information-Flow Control

There are a variety of static information-flow type systems guaranteeing noninterference. Volpano et al. [2] introduce a secure type system for a simple imperative language with state, which Volpano and Smith [30] extend to include exceptions and enforce termination-sensitive noninterference. This extension assumes termination behavior and exceptional failures are entirely public, restricting the language but simplifying the proof. The SLam calculus [3] guarantees termination-insensitive noninterference for a simple λ-calculus-style language, and Abadi et al. [4] prove that it also provides termination-sensitive noninterference when using call-by-name semantics.

Though ignoring termination concerns, FlowCaml [5] adds strong information-flow guarantees to ML. It enforces noninterference with arbitrary dynamically-allocated state cells, each with its own label, and different types of exceptions. Moreover, the labels for state cells and exceptions need not be related. Instead it uses type-and-effect system to track exceptions and modifies the pc label—which constrains both writes and exceptions—to account for uncaught exceptions. More recent systems like Fabric [31] and Jif [32], upon which Fabric is built, include similar rules, though neither provides a proof of noninterference.

The above works and others use a variety of techniques to prove noninterference. The first proofs of static noninterference [2], [3], [30] relied on structural induction with careful

manual reasoning. Pottier and Simonet [5] used bracketed pairs of terms to simulate two program executions with different high inputs and compare the outputs. This technique makes combining state and exceptions tractable, but provides no means to reason about termination. Other proofs rely on semantics using partial equivalence relations [4], [33], [34] or logical relations [17], [35]. The complexity of all of these approaches lies in reasoning about effects, demonstrating the value of Noninterference Half-Off.

We base all of the example languages in this paper on DCC [4]. DCC was originally designed to explore dependency, with information flow as an interesting special case. Interestingly, DCC was not given an operational semantics or a noninterference theorem in the original paper. Instead, Abadi et al. [4] described a domain-theoretic semantics, and used it to prove a semantic security theorem closely related to noninterference. Tse and Zdancewic [33] later developed an operational semantics for DCC, and proved a noninterference theorem analogous to the one we used in Section II by translating DCC into System F and using parametric reasoning. While Shikuma and Igarashi [35] found a flaw in Tse and Zdancewic's proof, Bowman and Ahmed [14] were later able to repair this proof. Algehed and Bernardy [15] were then able to extend and simplify the proof technique, leading to a verified version of the proof written in Agda.

DCC is the paradigmatic *coarse-grained* IFC language, a style that is characterized by labeling and unlabeling data. The other languages mentioned above are all *fine-grained* IFC languages, where each type includes a label. Though the two approaches may appear substantially different, Rajani and Garg [17] recently proved them equivalent.

Both DCC_{pc} [33] and the Sealing Calculus [35] include *protection context labels* that look very similar to our pc labels. Both languages, however, are pure, and the labels serve only to securely include a more permissive typing rule for $\mathsf{unlabel}$. While our examples could also employ this technique, it would increase the complexity of the type systems, particularly the type-and-effect systems which would need to include both protection context labels and effects.

Other work has implemented coarse-grained IFC as monadic libraries, mostly in Haskell [8], [11], [20], [36], [37], [38]. Both Algehed and Russo [36] and MAC [11], moreover, handle effectful computation via monadic reasoning. Algehed and Russo [36] in particular advocate building noninterfering pure languages, and using monads to define effects on top of them. They do not, however, explore the connections to pc systems. MAC [11] combines the monads for effects and the monad for labels, and therefore still requires a pc label.

B. The Theory of Effects

Type-and-effect systems originated as a program analysis technique [21], [22], [23]. This technique allowed compilers to leverage the type system of their source language to track other properties of programs, enabling optimizations like dead-code elimination that may behave differently depending on effects.

Wadler and Thiemann [25] gave type-and-effect systems semantics via monads by recognizing the correspondence between type-and-effect systems and Moggi's [39], [40] notions of computation. This result gave rise to a long line of work describing generalizations of monads which could be used to give semantics to as many type-and-effect systems as possible. The most relevant for this work are Wadler and Thiemann's (and Orchard et al.'s [26]) indexed monads, which work on a lattice of effects [25] and Tate's [13] productors, which work on an arbitrary effector.

VIII. Conclusion

We combined ideas from the theory of effects and static IFC to develop Noninterference Half-Off, a powerful technique for proving noninterference, and prove its broad applicability. We demonstrated how it simplifies proofs and unifies previously-disparate concepts.

Our general semantic framework and Noninterference Half-Off—both the technique and the theorem—formalize the folklore perspective that a pc label bounds the effects of a computation. We demonstrated the power of this perspective and our framework by showing how the view of nontermination as an effect allows us to unify termination sensitivity with detection of implicit flows from more-traditional effects like state and exceptions.

The same approach would allow proofs of timing-sensitive noninterference or noninterference for languages with continuation-based control structures. Using naive monadic encodings of these effects may result in overly-restrictive type systems. Our results, however, show that devising more permissive monads requires no more work than constructing more-permissive pc systems, but both admits a simple proof of noninterference and allows a designer to directly apply the many results in the literature about monadic effects.

By adopting the abstract point-of-view of the effects community, we were able to develop a framework and prove noninterference for a large swath of languages. We believe that Noninterference Half-Off demonstrates the power of such a general viewpoint. We moreover hope that it can lead to similarly expansive results for other more complex IFC properties, such as those constraining downgrading [41], [42].

References

[1] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Symposium on Security and Privacy (SSP) (Oakland)*, 1982.

[2] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis," *Journal of Computer Security (JCS)*, vol. 4, no. 3, 1996.

[3] N. Heintze and J. G. Riecke, "The SLam calculus: Programming with secrecy and integrity," in *Principles of Programming Languages (POPL)*, 1998.

[4] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke, "A core calculus of dependency," in *Principles of Programming Languages (POPL)*, 1999.

[5] F. Pottier and V. Simonet, "Information flow inference for ml," in *Principles of Programming Languages (POPL)*, 2002.

[6] S. Zdancewic and A. C. Myers, "Secure information flow via linear continuations," *Higher-Order and Symbolic Computation*, vol. 15, no. 2-3, 2002.

[7] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 21, no. 1, 2003.

[8] T.-c. Tsai, A. Russo, and J. Hughes, "A library for secure multi-threaded information flow in haskell," in *Computer Security Foundations (CSF)*, 2007.

[9] W. Rafnsson and A. Sabelfeld, "Compositional information-flow security for interactive systems," in *Computer Security Foundations (CSF)*, 2014.

[10] M. P. Milano and A. C. Myers, "MixT: A language for mixing consistency in geodistributed transactions," in *Programming Languages Design and Implementation (PLDI)*, 2018.

[11] M. Vassena, A. Russo, P. Buiras, and L. Waye, "MAC: A verified static information-flow control library," *Journal of Logical and Algebraic Methods in Programming (JLAMP)*, vol. 95, 2018.

[12] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Principles of Programming Languages (POPL)*, 1999, pp. 228–241.

[13] R. Tate, "The sequential semantics of producer effect systems," in *Principles of Programming Languages (POPL)*, 2013.

[14] W. J. Bowman and A. Ahmed, "Noninterference for free," in *International Conference on Functional Programming (ICFP)*, 2015.

[15] M. Algehed and J.-P. Bernardy, "Simple noninterference from parametricity," in *International Conference on Functional Programming (ICFP)*, 2019.

[16] B. C. Pierce, *Types and Programming Languages*. MIT press, 2002.

[17] V. Rajani and D. Garg, "Types for information flow control: Labeling granularity and semantic models," in *Computer Security Foundations (CSF)*, 2018.

[18] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *International Cryptology Conference (CRYPTO)*. Springer, 1996, pp. 104–113.

[19] L. Waye, P. Buiras, D. King, S. Chong, and A. Russo, "It's my privilege: Controlling downgrading in DC-labels," in *Security and Trust Management (STM)*, 2015.

[20] A. Russo, K. Claessen, and J. Hughes, "A library for lightweight information-flow security in haskell," in *Haskell Symposium (HASKELL)*, 2008.

[21] J. M. Lucassen and D. K. Gifford, "Polymorphic effect systems," in *Principles of Programming Languages (POPL)*, 1988.

[22] F. Nielson, "Annotated type and effect systems," *ACM Computing Surveys (CSUR)*, vol. 28, no. 2, 1996.

[23] F. Nielson and H. R. Nielson, "Type and effect systems," in *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*. Springer, 1999.

[24] D. Marino and T. Milstein, "A generic type-and-effect system," in *Types in Language Design and Implementation (TLDI)*, 2009.

[25] P. Wadler and P. Thiemann, "The marriage of effects and monads," in *International Conference on Functional Programming (ICFP)*, 1998.

[26] D. Orchard, T. Petricek, and A. Mycroft, "The semantic marriage of effects and monads," 2014. [Online]. Available: https://arxiv.org/abs/1401.5391

[27] S.-y. Katsumata, "Parametric effect monads and semantics of effect systems," in *Principles of Programming Languages (POPL)*, 2014.

[28] S. Fujii, S.-y. Katsumata, and P.-A. Mellisès, "Towards a formal theory of graded monads," in *Foundations of Software Science and Computational Structures (FOSSACS)*, 2016.

[29] J. Beck, "Distributive laws," in *Seminar on Triples and Categorical Homology Theory*, 2006.

[30] D. Volpano and G. Smith, "Eliminating covert flows with minimum typings," in *Computer Security Foundations Workshop (CSFW)*. IEEE, 1997.

[31] J. Liu, O. Arden, M. D. George, and A. C. Myers, "Fabric: Building open distributed systems securely by construction," *Journal of Computer Security (JCS)*, vol. 25, 2017.

[32] T. Magrino, J. Liu, O. Arden, C. Isradisaikul, and A. C. Myers, "Jif 3.5: Java information flow," June 2016, software release. [Online]. Available: https://www.cs.cornell.edu/jif

[33] S. Tse and S. Zdancewic, "Translating dependency into parametricity," in *International Conference on Functional Programming (ICFP)*, 2004.

[34] A. Sabelfeld and D. Sands, "A PER model of secure information flow in sequential programs," *Higher-Order and Symbolic Computation*, vol. 14, no. 1, 2001.

[35] N. Shikuma and A. Igarashi, "Proving noninterference by a fully complete translation to the simply typed λ-calculus," *Logical Methods in Computer Science (LMCS)*, vol. 4, no. 3, September 2008.

[36] M. Algehed and A. Russo, "Encoding dcc in haskell," in *Programming Languages and Analysis for Security (PLAS)*, 2017.

[37] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible dynamic information flow control in haskell," in *Haskell Symposium (HASKELL)*, 2011.

[38] O. Arden, "Flow-limited authorization," Ph.D. dissertation, Cornell University, 2017. [Online]. Available: https://users.soe.ucsc.edu/~owen/publications/pdfs/FLA_OwenArden.pdf

[39] E. Moggi, "Computational lambda-calculus and monads," in *Logic in Computer Science (LICS)*, 1989.

[40] ——, "Notions of computation and monads," *Information and Computation*, vol. 93, no. 1, 1991.

[41] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," in *Computer Security Foundations Workshop (CSFW)*, 2005.
[42] E. Cecchetti, A. C. Myers, and O. Arden, "Nonmalleable information flow control," in *Computer and Communication Security (CCS)*, 2017.