Semantics for Noninterference with Interaction Trees

Lucas Silver
University of Pennsylvania, USA

Paul He
University of Pennsylvania, USA

Ethan Cecchetti
University of Maryland, USA
University of Wisconsin – Madison, USA

Andrew K. Hirsch
State University of New York at Buffalo, USA

Steve Zdancewic
University of Pennsylvania, USA

Abstract

Noninterference is the strong information-security property that a program does not leak secrets through publicly-visible behavior. In the presence of effects such as nontermination, state, and exceptions, reasoning about noninterference quickly becomes subtle. We advocate using *interaction trees (ITrees)* to provide compositional mechanized proofs of noninterference for multi-language, effectful, nonterminating programs, while retaining executability of the semantics. We develop important foundations for security analysis with ITrees: two *indistinguishability* relations, leading to two standard notions of noninterference with adversaries of different strength, along with metatheory libraries for reasoning about each. We demonstrate the utility of our results using a simple imperative language with embedded assembly, along with a compiler into that assembly language.

2012 ACM Subject Classification Theory of computation → Denotational semantics; Security and privacy → Logic and verification; Security and privacy → Information flow control

Keywords and phrases verification, information-flow, denotational semantics, monads

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Information-flow guarantees state that programs respect the information-security specifications of their inputs and outputs. The most basic is *noninterference*, which states that secret data cannot influence publicly observable behavior. There are many languages designed to enforce information-flow properties, guaranteeing that programs treat their sensitive inputs correctly [e.g., 29, 40, 41]. The importance of information-security properties has increasingly led to verification efforts for such languages and systems [7, 21]. These efforts, however, are mostly limited to source-level guarantees for a single language. For security guarantees to be meaningful, the entire language toolchain must support them.

One of the key decisions when formalizing any effectful, possibly-nonterminating language is the choice of representation. Much prior work focuses on operational semantics defined as a relation on syntax, or on trace models defined as a predicate over lists or streams of observations [22, 26, 37]. However, such definitions often require auxiliary constructs, like program counters or evaluation contexts, making proofs brittle and hard to compose. These concerns are particularly pronounced for information-security properties, which often rely on subtle definitions with delicate correctness proofs. The complexity of multi-language settings further complicates the already-fraught choice of language representation.

Interaction Trees (ITrees) [58, 61] provide an alternative: a runnable denotational semantics for effectful, potentially-nonterminating programs, with a library implemented in Coq [30]. Intuitively, ITrees represent programs as interactions with the environment. At a technical level, ITrees are a coinductive data type based on free monads [51]. Programs are either done and provide a return value, emit an *event* to the environment and continue once the environment provides a response, or produce a "silent event," allowing ITrees to represent (silently) diverging programs in strongly normalizing metalanguages. By interpreting the events into a suitable monad [32], ITrees can express the semantics of diverse programming-language features, and thus many different languages. This versatility makes ITrees well-suited to cross-language reasoning [58] and reasoning about real-world toolchains [25, 61].

ITrees come equipped with a notion of program equivalence based on *weak bisimilarity*, which considers programs equivalent if they differ only by a finite number of silent steps. Properties like noninterference, however, require more nuanced reasoning because some program behaviors are visible to an attacker while others are not.

This work introduces two *indistinguishability* relations for ITrees to capture these intuitions: one *progress-sensitive* and one *progress-insensitive*. These definitions—motivated by corresponding notions found in the information-flow security literature [46, 56, 57]—adapt the notion of bisimilarity to account for what information is available to an adversary. They require delicate treatment of the interplay between nontermination and the interactions of a program with its environment. Progress-sensitive noninterference is a very strong guarantee, but is overly restrictive for many real-world programming tasks. For instance, it generally disallows loops that depend on secret data. Progress-insensitive noninterference is less demanding, but provides considerably less security [6].

While the definitions of ITrees and our indistinguishability relations are coinductive, we provide metatheoretic results allowing a proof engineer to reason with these relations without manual coinduction. These results further connect these indistinguishability relations to the standard ITrees notion of bisimilarity, providing compatability with existing results.

We validate this design with a simple toolchain for cross-language noninterference. The toolchain consists of a simple imperative language, Imp, and a simple assembly language, Asm. There are two type systems for Imp and a compiler from Imp to Asm. One type system enforces progress-sensitive noninterference and the other enforces progress-insensitive noninterference. In addition to standard information flow typing rules, the type systems allow for *semantic typing*: any semantically secure program can be considered well typed. This flexibility allows Imp to support embedded Asm blocks without giving a type system to Asm, and it demonstrates the powerful semantic composition of our security reasoning. We further verify that our Imp-to-Asm compiler preserves both kinds of noninterference. This preservation relies only on semantic security, not the type system, which is required to allow for security preservation with semantic typing.

To further demonstrate the utility of our approach, we include exceptions in Imp. Exceptions show how our indistinguishability semantics interact with effects that may alter control flow, which are a particular challenge for information-flow reasoning. This inclusion also requires an extension to the ITrees library that is orthogonal to the security extensions.

Section 2 reviews background on information-flow control and ITrees, the Imp language, and its semantics defined with ITrees. The contributions of this paper are as follows.

- Section 3 extends the ITrees library with exceptions and exception handlers.
- Section 4 adapts ITrees metatheory to reason about security guarantees, defining progress-sensitive and progress-insensitive notions of indistinguishability and noninterference.

- Section 5 uses ITrees and the new relations to prove the security of two standard information-flow type systems for Imp.
- Section 6 extends Xia et al.'s [58] simple compiler from Imp to Asm with exceptions and print effects. We then show that Xia et al.'s notion of compiler correctness immediately implies security preservation using only the metatheory of indistinguishability.

Finally, Section 7 discusses related work and Section 8 concludes. All definitions and theorems described in this paper have been formalized in Coq.[1]

2 Background

We now review background on information-flow control, interaction trees, and Imp.

2.1 Information-Flow Control

We represent information-security policies using a set of *information-flow labels* $\mathcal{L}$ that must form a preorder. That is, there is a reflexive, transitive relation $\sqsubseteq$ (pronounced "flows to") on labels where $\ell \sqsubseteq \ell'$ means that any *adversary* who can see information with label ℓ' can also see information with label ℓ. We also identify adversaries with labels. An adversary at label ℓ can only see information with labels that flow to ℓ. Information-flow systems use a variety of orderings, including simply "public" and "secret," subsets of permissions [63], lattices over principals making up a system [5, 34, 50], and orderings based on logical implication [40].

The classic information-flow security policy is *noninterference*: if an adversary cannot distinguish a program's inputs, they should not be able to distinguish its outputs or its interactions with the environment. Because information-flow labels determine which data an adversary can observe, a semantic version of noninterference requires a semantic model of information-flow labels. Sabelfeld and Sands [47] suggest modeling labels as partial equivalence relations (PERs) on terms. PERs are relations that are symmetric and transitive, but not necessarily reflexive. PERs act like equivalence relations on a subset of their domain. For information-flow security, such PERs are called "indistinguishability relations."

This model further asserts that indistinguishable programs take indistinguishable inputs to indistinguishable outputs. That is, related programs, applied to related inputs, produce related computations. This closure property allows a semantic version of noninterference to be defined as self-relation of a program. A program is related to itself—and noninterfering—if and only if, for every adversary, given any two inputs an adversary cannot distinguish, it produces two computations that adversary cannot distinguish.

As we will see in Section 4, indistinguishability gives a natural way to reason about noninterference using ITrees. Requiring every indistinguishability relation to be a PER, however, corresponds to strong assumptions about the adversary. In particular, it requires that the adversary be able to distinguish a program that silently diverges from a program that takes arbitrarily long to produce an observable interaction with its environment. Noninterference against this strong adversary is known as *progress-sensitive* noninterference. While this strength provides more security, enforcing progress-sensitive noninterference results in a prohibitively expensive programming model [Section 5.1, 46, 56]. To allow for enforcement of *progress-insensitive* noninterference, the indistinguishability model is often relaxed to not require transitivity [16, 43, 55].

[1] **For reviewers:** Our Coq development is available as part of the review process, and we intend to submit (a better-documented version of) it for artifact evaluation should the paper be accepted.

2.2 Basic Definitions for Interaction Trees

Interaction Trees (ITrees) [58] are a coinductive data structure designed to give denotational semantics to effectful, possibly divergent programs. ITrees model such computations as branching trees where internal nodes represent *events*, or interactions with the environment, with a branch for each different possible response from the environment. The use of coinduction means that these trees can be infinite, modeling diverging programs. Because ITrees give a denotational semantics to programs, they are a language-agnostic view of programs. Thus, we can use ITrees as a common domain for multiple languages, allowing us to reason about how those languages interact.

The type of an ITree includes an event signature E and a result type R. The result type simply specifies the output type if the program halts normally. The event signature E defines the interface by which the environment interacts with the program. $E : \mathit{Type} \to \mathit{Type}$ is a type transformer that takes an answer type A and returns $E\ A$, the type of an event that produces a value of type A. For example, the event signature, `stateE`, modeling a state effect might have two constructors: `get` and `set`. A `get` event represents a state access that returns a number, so it has type $\texttt{stateE}(\mathbb{N})$. A `set` event represents an assignment that need not return any useful information, so it has type $\texttt{stateE}(\texttt{unit})$.

ITrees have the following constructors.

$$\frac{r : R}{\texttt{ret}\ r : \texttt{itree}\ E\ R} \qquad \frac{t : \texttt{itree}\ E\ R}{\tau \cdot t : \texttt{itree}\ E\ R} \qquad \frac{e : E\ A \qquad k : A \to \texttt{itree}\ E\ R}{\texttt{Vis}\ e\ k : \texttt{itree}\ E\ R}$$

In this paper, a double line in an inference rule means that it should be interpreted coinductively, while a single line is interpreted inductively, as usual. This definition, then, is a fully coinductive definition, since the only single-line definition is a base case.

The ITree `ret` r represents a program terminating with a value r. The ITree $\tau \cdot t$ represents a silent internal step of computation, followed by the ITree t. Because ITrees are a *coinductive* data structure, we can chain an infinite number of τ's together in the ITree $t_{\text{spin}} = \tau \cdot t_{\text{spin}}$. Here, t_{spin} models a divergent program that causes no side effects. Finally, the ITree `Vis` $e\ k$ represents a visible event e of type $E\ A$ for some answer type A, followed by a continuation k that takes an answer of type A and produces an `itree` $E\ R$. Intuitively, k defines how the computation proceeds after the environment handles event e. Since k's behavior may differ depending on the value returned by e, there is one possible computational "branch" for each value of type A. In this view, ITrees are potentially infinitely long trees.

For any event signature E, `itree` E forms a monad [32]. The unit operation is provided by the `ret` constructor, and the bind operation, written $m \gg= k$, is defined as a corecursive function which replaces every `ret` r in m with $k\ r$. We will also use the common monad notation $x \leftarrow t_1 ; t_2$ in place of $t_1 \gg= \lambda x.t_2$. ITrees satisfy the monad laws up to strong bisimulation, which we use as an equivalence on ITrees since they are potentially infinite objects. Two ITrees are strongly bisimilar when they have exactly the same shape (including the values returned at corresponding leaves).

In combination with the monad operations, another useful operation is `trigger`, which lifts an event into an ITree that immediately returns the environment's response:

$$\texttt{trigger}\ e = \texttt{Vis}\ e\ \texttt{ret}$$

ITrees also support an *iteration* operation:

$$\texttt{iter} : \forall A, B.(A \to \texttt{itree}\ E\ (A \oplus B)) \to A \to \texttt{itree}\ E\ B$$

$$\begin{array}{llll}
\text{Expressions} & e & ::= & x \mid n \mid e+e \mid e-e \mid e*e \\
\text{Commands} & c & ::= & \mathsf{skip} \mid x \coloneqq e \mid c_1 \,;\, c_2 \mid \mathsf{while}\ (e)\ \mathsf{do}\ \{c\} \\
 & & \mid & \mathsf{if}\ (e)\ \mathsf{then}\ \{c_1\}\ \mathsf{else}\ \{c_2\} \mid \mathsf{print}(\ell, e) \mid \mathsf{inline}\ \{a\} \\
\text{Inlined Assembly} & a & ::= & \text{(see Section 6)}
\end{array}$$

Figure 1 Imp syntax, where x is a variable, n is a number, and ℓ is an information-flow label.

Intuitively, `iter` *body* a acts as a do-while loop, running *body* on input a and either continuing with a new value of type A, or stopping with a final value of type B.

2.3 Semantics for Imp with Security Labels

To explore how ITrees can help us verify noninterference properties, we will use a simple imperative language, Imp, as a running example and case study. Conveniently, previous work on both ITrees [58] and noninterference [46] use Imp as case studies, ensuring that the connection we make corresponds with existing tools and techniques in both domains. Our version of Imp, presented in Figure 1, includes features not present in the works cited above: the ability to print expressions to one of several output streams, and the ability to inline code from a simple assembly language. Section 3 will further extend Imp to allow throwing and catching exceptions. The output streams are indexed by information-flow labels, and we think of stream ℓ as being visible to any adversary at or above ℓ, but no others. Thus, printing secret information to a public stream leaks data.

The assembly language, Asm, is a simplification of standard assembly language. We allow an infinite number of registers, and we assume that the heap is addressed by variables, as in Imp. We also do not allow dynamic jumps, only jumps to fixed addresses. Beyond those simplifications, we include features similar to those in Imp: we allow printing to streams indexed by information-flow labels and, as we show later, the Asm semantics can model *uncaught* exceptions, both features necessary for correct compilation of Imp code. We discuss the syntax and semantics of Asm in more detail in Section 6.

As in languages like C, embedding Asm in Imp allows developers more control over the performance of their code. For instance, the simple compiler in Section 6 would compile the Imp program $y \coloneqq x+1\,;z \coloneqq x+2$ to an Asm program that loads data from x into a register twice, once for each assignment. Since Loads are relatively expensive, when the Imp code above appears in a critical loop a developer might replace it with the following Asm code:

$$\begin{array}{lllll}
\textsc{Start}: & \textsc{load} & \$0 & \leftarrow & x \\
 & \textsc{add} & \$0 & \leftarrow & \$0, 1 \\
 & \textsc{store} & y & \leftarrow & \$0 \\
 & \textsc{add} & \$0 & \leftarrow & \$0, 1 \\
 & \textsc{store} & z & \leftarrow & \$0 \\
 & \textsc{jmp} & \textsc{Exit} & &
\end{array}$$

This program starts from the Start label, and terminates the program by jumping to the Exit label. Unlike our compiler's output, this custom Asm only has one load instruction.

Giving semantics to Imp using ITrees requires defining events representing possible interactions between an Imp program and its environment. Imp has three types of events: `stateE` for the heap state, `regE` for the register state, and `printE` for output. There are two constructors for `stateE` events, one for reading and one for writing.

$$\mathtt{get} : \mathsf{var} \to \mathtt{stateE}(\mathbb{N}) \qquad\qquad \mathtt{set} : \mathsf{var} \to \mathbb{N} \to \mathtt{stateE}(\mathsf{unit})$$

$$\boxed{[\![e]\!]_e : \texttt{itree progE}\ \mathbb{N}} \qquad \boxed{[\![c]\!]_c : \texttt{itree progE unit}}$$

$$\begin{aligned}
[\![x]\!]_e &= \texttt{trigger get}(x) \\
[\![n]\!]_e &= \texttt{ret}\ n \\
[\![e_1 + e_2]\!]_e &= x \leftarrow [\![e_1]\!]_e\,; \\
&\quad\ y \leftarrow [\![e_2]\!]_e\,; \\
&\quad\ \texttt{ret}\ (x + y)
\end{aligned}$$

$$\begin{aligned}
[\![\mathsf{skip}]\!]_c &= \texttt{ret}\ () \\
[\![x := e]\!]_c &= n \leftarrow [\![e]\!]_e\,; \texttt{trigger set}(x, n) \\
[\![\mathsf{print}(\ell, e)]\!]_c &= n \leftarrow [\![e]\!]_e\,; \texttt{trigger print}(\ell, n) \\
[\![c_1\,;c_2]\!]_c &= [\![c_1]\!]_c\,; [\![c_2]\!]_c \\
\left[\!\!\left[\begin{array}{l}\mathsf{if}\ e \\ \mathsf{then}\ \{c_1\} \\ \mathsf{else}\ \{c_2\}\end{array}\right]\!\!\right]_c &= n \leftarrow [\![e]\!]_e\,; \begin{array}{l}\texttt{if}\ n \neq 0 \\ \texttt{then}\ \ [\![c_1]\!]_c \\ \texttt{else}\ \ [\![c_2]\!]_c\end{array} \\
[\![\mathsf{while}\ (e)\ \mathsf{do}\ \{c\}]\!]_c &= \texttt{iter}\left(\begin{array}{l}\lambda_.\, n \leftarrow [\![e]\!]_e\,; \\ \quad \texttt{if}\ n \neq 0 \\ \quad \texttt{then}\ \ ([\![c]\!]_c\,; \texttt{ret inl}()) \\ \quad \texttt{else ret inr}()\end{array}\right)() \\
[\![\mathsf{inline}\ \{a\}]\!]_c &= [\![a]\!]_{\texttt{asm}}
\end{aligned}$$

Figure 2 Imp denotational semantics

The `regE` events require another two constructors, again one for reading and one for writing.

$$\texttt{getreg} : \mathit{reg} \to \texttt{regE}(\mathbb{N}) \qquad\qquad \texttt{setreg} : \mathit{reg} \to \mathbb{N} \to \texttt{regE}(\mathsf{unit})$$

There is only one constructor for `printE` events: $\texttt{print} : \mathcal{L} \to \mathbb{N} \to \texttt{printE}(\mathsf{unit})$.

As Imp programs can produce all three types of events, we combine them with disjoint union. The resulting event type for Imp programs is $\texttt{progE} = \texttt{regE} \oplus \texttt{stateE} \oplus \texttt{printE}$. For notational simplicity, we elide the injection operator when using these compound events.

Figure 2 presents the denotation of Imp using these events. Note that there are two denotation functions: $[\![\cdot]\!]_e$ for expression and $[\![\cdot]\!]_c$ for commands. As expressions produce numbers and commands have no output, $[\![\cdot]\!]_e$ produces computations of type $\texttt{itree progE}\ \mathbb{N}$, while $[\![\cdot]\!]_c$ produces computations of type `itree progE unit`. The function $[\![\cdot]\!]_{\texttt{asm}}$ gives ITree-based semantics to Asm. Its full definition can be found in the work of Xia et al. [58]; we discuss the modifications necessary to accommodate our changes in Section 6.

The denotation for expressions is fairly straightforward, and, importantly for proofs, completely compositional—an expression's meaning is constructed from that of its subexpressions. The denotation of a variable is a `get` event, a literal n becomes $\texttt{ret}\ n$, and arithmetic expressions simply denote each argument and return the resulting value using bind.

Most commands are equally simple and compositional. skip is an immediate `ret`. Both assignment and print first denote the argument and then bind the result into the appropriate event. Sequencing is implemented with bind on a unit value that we elide. Conditionals first denote the condition, and then return the denotation of either the left or right command depending on the result.

Loops are more complex and make use of the `iter` combinator. The combinator expects a function that returns $\texttt{itree progE}\ (\mathsf{unit} \oplus \mathsf{unit})$, where a left value indicates "continue" and a right value indicates that the loop should terminate. The function given to `iter` first computes the value of the loop's guard expression. If the value is not zero, it sequences a single denotation of the loop body with `ret inl()`, indicating the loop should continue. Otherwise, if the value is zero, it signals to halt the iteration with `ret inr()`.

2.4 Handlers and Interpretations

The events in an ITree can be thought of as a kind of syntax. Even though we give them names that suggest certain behaviors, like `get` and `set`, nothing about their structure enforces this behavior. Consider the ITree $\texttt{trigger set}(x, 0)\,;\texttt{trigger get}(x)$: while the names suggest that the result of this `get` should be 0, it actually produces a tree with one branch for every natural number. Likewise, the ITree $[\![c]\!]_c$ representing an Imp program c does not fully express the behavior we would expect from c because it has uninterpreted state events.

The behavior of events is determined by a function called an *event handler* from events to effectful computations. As is standard, we represent effectful computations as elements of a monad M, giving an event handler the type $\forall A.\ E\ A \to\ M\ A$. For example, consider h_{prog} which uses the standard monadic interpretation of state to interpret `progE` events:

$$
\begin{aligned}
h_{prog}(\texttt{get}(x)) &= \lambda(r,h).\,\texttt{ret}\ (r,h,h(x)) \\
h_{prog}(\texttt{set}(x,n)) &= \lambda(r,h).\,\texttt{ret}\ (r,h[x\mapsto n],()) \\
h_{prog}(\texttt{getreg}(x)) &= \lambda(r,h).\,\texttt{ret}\ (r,h,r(x)) \\
h_{prog}(\texttt{setreg}(x,n)) &= \lambda(r,h).\,\texttt{ret}\ (r[x\mapsto n],h,()) \\
h_{prog}(\texttt{print}(\ell,n)) &= \lambda(r,h).\,\texttt{trigger print}(\ell,n)\,;\texttt{ret}\ (r,h,())
\end{aligned}
$$

Any event handler can be lifted to a function from ITrees to effectful computations using the `interp` function, which traverses an ITree, replacing each event with the effectful computation assigned by the handler. The full semantics of an Imp program is the *interpreted* ITree, $\texttt{interp}\ h_{prog}\ [\![c]\!]_c$.

2.5 Inlined Asm and Undefined Behavior

Adding support for inlined Asm code introduces a new complication to the semantics of Imp: undefined behavior. To analyze the correctness and security of a language toolchain, we need to define the behavior of source-level programs. The semantics defined in Section 2.3 and Section 2.4 do that for Imp as long as any inlined Asm has well-defined behavior. However, consider the following Imp program, which contains inlined Asm.

$$
\begin{array}{rclrll}
p & = & c\,;\mathsf{inline}\ \{ & \textsc{Start}: & \textsc{brz} & \$0\ \ \mathrm{A1}\ \ \mathrm{A2} \\
 & & & \mathrm{A1}: & \textsc{load} & X\ \leftarrow\ 0 \\
 & & & & \textsc{jmp} & \textsc{Exit} \\
 & & & \mathrm{A2}: & \textsc{load} & X\ \leftarrow\ 1 \\
 & & & & \textsc{jmp} & \textsc{Exit}\qquad \}
\end{array}
$$

The inlined Asm program looks at the value in register 0 and, if it is zero, jumps to address A1; otherwise it jumps to address A2. Thus, the value of X after executing program p depends on the value of register \$0 after c is executed. However, it is not clear what the register's value will be when this program is compiled and run, since reasonable compilers could use the register \$0 in different ways—or not at all—to compile the Imp command c, resulting in different register states. We thus consider inlining any Asm program that relies on the initial values of registers to be undefined behavior. We formalize this property in Section 5.3. We further take the same approach as CompCert,[2] and only verify the correctness and security of programs that are well-defined.

---

[2] Personal Communication with Xavier Leroy.

$$\frac{\mathcal{R}(r_1, r_2)}{E \vdash \texttt{ret}\ r_1 \approx_{\mathcal{R}} \texttt{ret}\ r_2} \qquad \frac{e : E\ A \qquad \forall (a : A), E \vdash k_1(a) \approx_{\mathcal{R}} k_2(a)}{E \vdash \texttt{Vis}\ e\ k_1 \approx_{\mathcal{R}} \texttt{Vis}\ e\ k_2}$$

$$\frac{E \vdash t_1 \approx_{\mathcal{R}} t_2}{E \vdash \tau \cdot t_1 \approx_{\mathcal{R}} \tau \cdot t_2} \qquad \frac{E \vdash t_1 \approx_{\mathcal{R}} t_2}{E \vdash \tau \cdot t_1 \approx_{\mathcal{R}} t_2} \qquad \frac{E \vdash t_1 \approx_{\mathcal{R}} t_2}{E \vdash t_1 \approx_{\mathcal{R}} \tau \cdot t_2}$$

Figure 3 Inference rules for weak bisimulation

2.6 Weak Bisimulation

Much of the power of ITrees comes from their equational theory. While it is natural to reason about coinductive structures like ITrees using *bisimulation*, the "obvious" bisimulation relation is too strong for our needs. For example, the more complex operations we have introduced, like `iter` and `interp`, insert some (finite number of) silent internal τ steps, which would be convenient to ignore. For this reason, we often prefer to work with a coarser equivalence called *weak bisimulation*, or *equivalence-up-to-tau* (`eutt`), which ignores finite numbers of τs when comparing two ITrees.

Weak bisimulation is defined by the inference rules in Figure 3, where the relation is parameterized by a relation $\mathcal{R}$ used to compare return values. Furthermore, the event signature of the two ITrees is made explicit by the E parameter. The first three inference rules correspond to the three constructors of an ITree and are exactly the definition of strong bisimulation. The last two rules allow us to ignore any finite number of τs. The fact that these rules are inductive rather than coinductive is crucial. If these rules were coinductive, we could use them to show that a diverging ITree with only τ constructors is equivalent to any other ITree. Using this technique of mixed induction and coinduction, coinductive rules may be used infinitely often, while inductive rules can only be used a finite number of times before either terminating with a base case or applying a coinductive rule.

Xia et al. [58] formalize the ITrees data structure and its metatheory in a Coq library,[3] providing a rich equational theory up to this definition of weak bisimulation. This theory allows users to prove termination-sensitive properties about ITrees without explicitly performing coinductive proofs, greatly reducing the proof burden.

3 Exceptions with Interaction Trees

As mentioned in Section 1, we include exceptions in Imp since they are an important example of an effect which can change the control flow. In this section, we show how to model exceptions with ITrees by adding throw and catch constructs to Imp as follows:

$$\text{Commands} \quad c \quad ::= \quad \cdots \mid \mathsf{throw}(\ell) \mid \mathsf{try}\ \{c_1\}\ \mathsf{catch}\ \{c_2\}$$

Note that the throw command includes an information flow label, specifying who may see the exception.

[3] This Coq development, as well as our extension of it, defines coinductive relations using the paco library [19, 60] for coinductive reasoning.

3.1 Exceptions as Halting Events

We model exceptions in ITrees as *halting events*. Recall from Section 2.2 that events create one branch for every possible response from the system. If an event has an uninhabited response type, then that continuation can never be run since the answer type has no values. We call such events *halting* because they force the computation to stop. We formalize this with the following lemma:

▶ **Lemma 1.** *Suppose A is an uninhabited type and e is an event of type $E\ A$, then given any continuations k_1 and k_2 and any return relation $\mathcal{R}$, $E \vdash \texttt{Vis}\ e\ k_1 \approx_{\mathcal{R}} \texttt{Vis}\ e\ k_2$.*

The continuation of a halting event cannot be run and has no effect on the computational content of the ITree. This allows a programmer to assign such an ITree any desired return type without changing its computational content. This property makes halting events useful for modeling (uncaught) exceptions: an exception can have any type and causes computation to stop. To represent exceptions using this strategy, we use an event type `excE` with only a single constructor $\texttt{exc} : \mathit{Err} \to \texttt{excE}(\emptyset)$ which takes the exception's data payload and produces an event with an empty answer type. This allows us to define $[\![\mathsf{throw}(\ell)]\!]_c = \texttt{trigger exc}(\ell)$.

3.2 Catching Exceptions

Real-world languages do not just throw exceptions, they also *handle* them. To implement exception handling in ITrees, we use a common monadic interpretation of exceptions: we allow programs to return either a standard return value or an exception. Specifically, we move from an ITree of type $\texttt{itree}\ \ (\texttt{excE}\ \mathit{Err} \oplus E)\ R$ to one of type $\texttt{itree}\ \ (\texttt{excE}\ \mathit{Err} \oplus E)\ (\mathit{Err} \oplus R)$ using `interp` to lift the following h_{exc} event handler to the entire ITree, as described in Section 2.4.

$$h_{exc} : \forall A,\ (\texttt{excE}\ \mathit{Err} \oplus E)\ A \to \texttt{itree}\ \ (\texttt{excE}\ \mathit{Err} \oplus E)\ (\mathit{Err} \oplus A)$$

$$h_{exc}(\texttt{inl}(\texttt{exc}(e))) := \texttt{ret inl}(e)$$

$$h_{exc}(\texttt{inr}(e)) := x \leftarrow \texttt{trigger inr}(e); \texttt{ret inr}(x)$$

Even though the resulting ITree cannot have exception events, we still assign it a type that allows them so it can cleanly compose with ITrees that do contain exception events. This choice allows monadic bind to apply exception handlers—which may themselves contain exception events—to any left values (exceptions) while leaving right values (normal returns) unmodified. The result is the following exception-handling combinator, where $\texttt{case}\ k_1\ k_2$ chooses the continuation k_1 or k_2 if the return value is `inl` or `inr`, respectively.

$$\texttt{trycatch}(t, k_c) := \texttt{interp}\ h_{exc}\ t \gg\!= \texttt{case}\ k_c\ \texttt{ret}$$

This `trycatch` combinator has a straightforward metatheory. In particular, we show how it interacts with the constructors of ITrees, allowing proof engineers to reason about `trycatch` without using manual coinduction.

▶ **Theorem 2.** *The `trycatch` operator satisfies the following equivalences:*

$$\begin{aligned}
E \vdash \texttt{trycatch}(\texttt{ret}\ r, k_c) &\approx_{=} \texttt{ret}\ r \\
E \vdash \texttt{trycatch}(\tau \cdot t, k_c) &\approx_{=} \texttt{trycatch}(t, k_c) \\
E \vdash \texttt{trycatch}(\texttt{Vis inr}(a)\ k, k_c) &\approx_{=} \texttt{Vis inr}(a)\ \lambda x.\texttt{trycatch}(k(x), k_c) \\
E \vdash \texttt{trycatch}(\texttt{Vis inl}(\texttt{exc}(\varepsilon))\ k, k_c) &\approx_{=} k_c(\varepsilon)
\end{aligned}$$

Finally, the `trycatch` operator provides a simple denotation of Imp's try-catch blocks:

$$[\![\mathsf{try}\ \{c_1\}\ \mathsf{catch}\ \{c_2\}]\!]_c = \texttt{trycatch}([\![c_1]\!]_c, \lambda\_.\ [\![c_2]\!]_c)$$

4 Indistinguishability of Interaction Trees

To leverage the common semantic domain of ITrees to guarantee the security of a toolchain, we define our indistinguishability relation purely semantically. Intuitively, for programs to be indistinguishable, they must return indistinguishable results and have indistinguishable interactions with their environments.

Since return values can be arbitrary types, we follow `eutt` by parameterizing indistinguishability over a *return relation* $\mathcal{R}$. For indistinguishability, $\mathcal{R}$ describes when two values *appear* to be the same to the adversary. For example, consider a program that outputs a pair (a, b) where a is visible to Alice and b is visible to Bob, but not vice versa. The values $(1, 1)$ and $(1, 2)$ are not equal, but they are indistinguishable from Alice's perspective, as she can only see the first element. We can represent Alice's view of the output with a relation $\mathcal{R}_{\text{Alice}}$ defined by $\mathcal{R}_{\text{Alice}}((a, b), (a', b')) \iff a = a'$.

We could simply use `eutt` with a return relation $\mathcal{R}$ modeling indistinguishability. The resulting relation would model an adversary who can only see some part of the program's output, but it would require the two programs to interact with the environment in precisely the same way. Most settings, however, allow adversaries to see some interactions, but not others. For example, memory may be partitioned into a protected heap the adversary can never see, and an unprotected heap that it can see at all times. Reasoning about security when some events are visible and others are not requires changing `eutt` to account for what the adversary can observe.

4.1 Secure Equivalence Up-To Taus

Our indistinguishability relation is called *secure equivalence up-to tau* or `seutt`. In addition to a return relation, `seutt` is also parameterized by a label ℓ, representing what the adversary can see, and a *sensitivity function* ρ that maps events to labels, representing who may observe which events. Intuitively, two ITrees are related by `seutt` if the environment interactions appear the same to an adversary who can see events only at or below label ℓ, and the return values are related by $\mathcal{R}$. We write the relation as $E; \rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}} t_2$.

Notably, we base the relation on `eutt`, which makes it progress sensitive. Recall from Section 2.1 that progress-sensitive noninterference allows any adversary to determine if a program silently diverges, and is often prohibitively expensive to enforce. We will also define `pi-seutt`, a progress-insensitive version of `seutt`, in Section 4.3. The judgments take the same form, so we annotate the turnstile with a subscript *ps* or *pi* to distinguish them visually.

For presentation, we separate the rules for `seutt` into three groups: rules covering returns, τs, and public events (Figure 4), rules covering secret events that do not halt the program (Figure 5), and rules covering secret halting events (Figure 6).

Public Events and Returns. When an adversary is able to see an event, indistinguishability acts just like weak bisimulation. The rules, found in Figure 4, are almost identical to the rules of `eutt`, but with the added requirement that any visible event be visible to the adversary. That is, we require $\rho(e) \sqsubseteq \ell$ in PubVis.

It might seem mysterious that we *require* the event to be visible in PubVis. But allowing this rule to apply no matter the visibility would allow the adversary too much power, since they would know that the same result is returned on both sides of the equivalence. As we will see, the rule for invisible events is stricter. We will also see how this strictness, when proving a program p indistinguishable from itself, corresponds to proving that the behavior of p does not differ in runs in *low-equivalent* environments. If we were to allow high events in

$$[\textsc{Ret}]\ \frac{\mathcal{R}(r_1, r_2)}{E;\rho \vdash_{ps} \texttt{ret}\ r_1 \approx^{\ell}_{\mathcal{R}} \texttt{ret}\ r_2} \qquad [\textsc{TauTau}]\ \frac{E;\rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}} t_2}{E;\rho \vdash_{ps} \tau \cdot t_1 \approx^{\ell}_{\mathcal{R}} \tau \cdot t_2}$$

$$[\textsc{PubVis}]\ \frac{\begin{array}{c}\forall a, E;\rho \vdash_{ps} k_1(a) \approx^{\ell}_{\mathcal{R}} k_2(a) \\ e : E\ A \qquad \rho(e) \sqsubseteq \ell\end{array}}{E;\rho \vdash_{ps} \texttt{Vis}\ e\ k_1 \approx^{\ell}_{\mathcal{R}} \texttt{Vis}\ e\ k_2} \qquad [\textsc{TauL}]\ \frac{E;\rho \vdash_{ps} \tau \cdot t_1 \approx^{\ell}_{\mathcal{R}} t_2}{E;\rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}} t_2}$$

$$[\textsc{TauR}]\ \frac{E;\rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}} \tau \cdot t_2}{E;\rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}} t_2}$$

Figure 4 Inference rules for indistinguishability, where all events are visible

$$[\textsc{PrivVisTau}]\ \frac{\begin{array}{c}\forall a, E;\rho \vdash_{ps} k(a) \approx^{\ell}_{\mathcal{R}} t \qquad e : E\ A \\ \neg empty(A) \qquad \rho(e) \not\sqsubseteq \ell\end{array}}{E;\rho \vdash_{ps} \texttt{Vis}\ e\ k \approx^{\ell}_{\mathcal{R}} \tau \cdot t} \qquad [\textsc{PrivVisIndL}]\ \frac{\begin{array}{c}\forall a, E;\rho \vdash_{ps} k(a) \approx^{\ell}_{\mathcal{R}} t \qquad e : E\ A \\ \neg empty(A) \qquad \rho(e) \not\sqsubseteq \ell\end{array}}{E;\rho \vdash_{ps} \texttt{Vis}\ e\ k \approx^{\ell}_{\mathcal{R}} t}$$

$$[\textsc{PrivVisVis}]\ \frac{\begin{array}{c}\forall (a\!:\!A)(b\!:\!B), E;\rho \vdash_{ps} k_1(a) \approx^{\ell}_{\mathcal{R}} k_2(b) \qquad e_1 : E\ A \qquad e_2 : E\ B \\ \rho(e_1) \not\sqsubseteq \ell \qquad \rho(e_2) \not\sqsubseteq \ell \qquad \neg empty(A) \qquad \neg empty(B)\end{array}}{E;\rho \vdash_{ps} \texttt{Vis}\ e_1\ k_1 \approx^{\ell}_{\mathcal{R}} \texttt{Vis}\ e_2\ k_2}$$

Figure 5 Inference rules for indistinguishability, where events are not visible but answer types are inhabited

PubVis, this would allow our proof to only consider the behavior of p in one environment, breaking our correspondence with information-flow security.

Private Events With Responses. When the adversary is *unable* to view an event, `seutt` cannot act like `eutt`. In this case, the rules are designed to formalize two intuitions. If the computation continues after a secret event, we should treat the event like a τ, since the adversary cannot observe either. If the event halts the computation, the event should be equivalent to a silently nonterminating computation.

The rules in Figure 5, along with symmetric analogues of PrivVisTau and PrivVisIndL, handle the case where the event allows computation to continue—that is, the event's answer type is inhabited. The first rule, PrivVisTau, relates a private event `Vis` e k with a $\tau \cdot t$. In addition to requiring the event to be secret ($\rho(e) \not\sqsubseteq \ell$) and have a non-empty answer type ($\neg empty(A)$), it also requires the continuation k produce an ITree indistinguishable from t for *every* possible response. This requirement ensures that the adversary's future observations cannot depend on the response to the private event. Note that the requirement that A be non-empty does more than just specify when the rule applies. Without it, a private halting event would trivially satisfy this condition, allowing it to relate to any ITree with a τ in front. Since the adversary can determine when a program has halted, they should be able to distinguish, for example, a program that throws a private exception from a program which, after a τ, prints to a public channel. This rule ensures that this intuition holds.

PrivVisIndL is analogous to TauL, but for secret events instead of τ nodes. This rule has the same premises as PrivVisTau for the same reasons. Moreover, it only removes a node from the head of one ITree, not both. As with the definition of `seutt`, TauL, and TauR, we

$$[\textsc{EmpVisTau}]\ \frac{E;\rho \vdash_{ps} \texttt{Vis}\ e\ k \approx^{\ell}_{\mathcal{R}} t \qquad e : E\ A \qquad \mathit{empty}(A) \qquad \rho(e) \not\sqsubseteq \ell}{E;\rho \vdash_{ps} \texttt{Vis}\ e\ k \approx^{\ell}_{\mathcal{R}} \tau \cdot t}
\qquad
[\textsc{EmpVisVisL}]\ \frac{\forall b, E;\rho \vdash_{ps} \texttt{Vis}\ e_1\ k_1 \approx^{\ell}_{\mathcal{R}} k_2(b) \qquad e_1 : E\ A \qquad e_2 : E\ B \qquad \mathit{empty}(A) \qquad \rho(e_1) \not\sqsubseteq \ell \qquad \rho(e_2) \not\sqsubseteq \ell}{E;\rho \vdash_{ps} \texttt{Vis}\ e_1\ k_1 \approx^{\ell}_{\mathcal{R}} \texttt{Vis}\ e_2\ k_2}$$

Figure 6 Inference rules for indistinguishability, where events are halting and not visible

therefore make PrivVisIndL inductive, not coinductive, to avoid relating a infinite stream of secret events to all other ITrees.

Finally, PrivVisVis removes a private event from the head of both sides of the relation. As with the previous rules, we require both events to be private and have non-empty answer types. This time, we require the continuations of the two events to be indistinguishable for every possible response *of both events separately*. This requirement formalizes the idea that the adversary should not be able to distinguish the program's behavior on any pair of secret responses.

To see the power of this rule, consider whether an adversary who can see l but not h would find the following ITrees indistinguishable from themselves:

$$t_{\text{sec}} \triangleq \begin{array}{l} x \leftarrow \texttt{trigger get}(l); \\ y \leftarrow \texttt{trigger get}(h); \\ \texttt{trigger set}(h, x+y) \end{array} \qquad\qquad t_{\text{insec}} \triangleq \begin{array}{l} x \leftarrow \texttt{trigger get}(l); \\ y \leftarrow \texttt{trigger get}(h); \\ \texttt{trigger set}(l, x+y) \end{array}$$

One would hope that t_{sec} would be indistinguishable from itself, while t_{insec} would not be, and indeed that is the case. To (attempt to) prove that either tree is equivalent to itself, we walk through each ITree. Since l is visible, so is $\texttt{get}(l)$, so PubVis applies and requires only that each possible value of x produce an ITree that is indistinguishable from itself. Because h is secret, the adversary should not be able to observe or infer its value, so we must use PrivVisVis to remove $\texttt{get}(h)$. PrivVisVis requires that, for all possible *pairs* of values y_1, y_2, the continuations be indistinguishable. Thus in t_{sec}, $\texttt{trigger set}(h, x+y_1)$ must be indistinguishable from $\texttt{trigger set}(h, x+y_2)$. Since h is secret, so are the `set` events, so PrivVisVis can remove them even when they differ. After removing `set`, the remaining continuation always produces `ret` (), so Ret finishes the proof.

However, in t_{insec}, PrivVisVis does not apply to the `set` events since l is visible. PubVis only relates ITrees starting with the same event, but $\texttt{set}(l, x+y_1) \neq \texttt{set}(l, x+y_2)$ when $y_1 \neq y_2$. As a result, no rule applies after removing $\texttt{get}(h)$, so the adversary can distinguish t_{insec} from itself. In other words, t_{insec} is, indeed, insecure.

Private Halting Events. Finally, we turn to the case where an event the adversary cannot see halts the computation. In this case, the adversary should be unable to tell that the event took place, and therefore should not be able to distinguish a program with a secret halt from a program that never terminates. However, the adversary should still be able to distinguish it from any ITree that contains an event the adversary can see.

This intuition means that a private halting event should not be treated like a τ, as a private non-halting event is, but rather should be indistinguishable from *an infinite stream of* τ*s*. We formalize this approach with the rules presented in Figure 6 along with their symmetric analogues. EmpVisTau peels a single τ off the right ITree, leaving the private halting event on the left unmodified. EmpVisVisL does the same for a private event.

There are two interesting properties about these rules. First, unlike the rules for private events and τs that leave one side of the equivalence unmodified, these rules are coinductive, not inductive. This choice allows us to relate a private halting event to an entire nonterminating program, as long as that program has no public events. Indeed, no rule allows us to remove a private halting event, as there would be nothing left to compare. Second, EmpVisVisL has no requirement that B, the answer type of the not-necessarily-halting event, be non-empty. This choice avoids the need to explicitly handle the case where both ITrees contain private halts. If B is non-empty, then EmpVisVisL treats the event as a τ. If B is empty, then the first premise of the rule is trivially satisfied, which is desirable, as in that case both ITrees begin with a private halt event and should be equivalent.

4.2 The Metatheory of Indistinguishability

The `seutt` relation captures intuitions about when two ITrees are indistinguishable to some adversary, but using it requires a delicate mix of induction and coinduction. To both demonstrate the power of our definition and better support verification, we also develop a library of metatheory for indistinguishability. This library supports reasoning about cross-language toolchains without the need for explicit coinduction, as we will see when we verify the correctness of a security type system and compiler for Imp (Sections 5 and 6, respectively).

Indistinguishability as a PER Model. Recall from Section 2.1 that Sabelfeld and Sands [47] argue for indistinguishability forming a partial equivalence relation (PER). It would be nice if `seutt` always formed a PER, but because it is parameterized on an arbitrary relation for return values, that is not always the case. Instead, we prove generalized versions of transitivity and reflexivity. In particular, if we let $\overset{\leftrightarrow}{\mathcal{R}}$ denote the reverse relation of $\mathcal{R}$—that is, $\overset{\leftrightarrow}{\mathcal{R}}(x, y) \overset{\triangle}{\iff} \mathcal{R}(y, x)$—then the following theorems hold.

▶ **Theorem 3.** *For all $\mathcal{R}$, E, ρ, and ℓ, if $E; \rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}} t_2$, then $E; \rho \vdash_{ps} t_2 \approx^{\ell}_{\overset{\leftrightarrow}{\mathcal{R}}} t_1$.*

▶ **Theorem 4.** *If $E; \rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}_1} t_2$ and $E; \rho \vdash_{ps} t_2 \approx^{\ell}_{\mathcal{R}_2} t_3$ then $E; \rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}_1 \circ \mathcal{R}_2} t_3$.*

Note that if $\mathcal{R}$ is symmetric, then $\mathcal{R} = \overset{\leftrightarrow}{\mathcal{R}}$, and if $\mathcal{R}$ is transitive, then $\mathcal{R} \circ \mathcal{R} \subseteq \mathcal{R}$. These properties allow us to prove the following corollary.

▶ **Corollary 5.** *If $\mathcal{R}$ is a PER, then so is $E; \rho \vdash_{ps} - \approx^{\ell}_{\mathcal{R}} -$ for any E, ρ, and ℓ.*

ITree Combinators. ITrees are often defined using the combinators from Section 2.2, making it important to understand how indistinguishability interacts with those combinators. The definition of `seutt` directly describes how to relate simple programs defined using only `ret` and `trigger`, but they say nothing about larger ITrees built using bind and iteration.

Bind allows for the sequential composition of programs. We would like indistinguishable programs t_1 and t_2 followed by indistinguishable continuations k_1 and k_2 to compose into larger indistinguishable programs $t_1 \gg= k_1$ and $t_2 \gg= k_2$. The following theorem says that this result holds whenever the relation $\mathcal{R}_1$, securely relating t_1 and t_2, puts enough constraints on their possible outputs to ensure that k_1 and k_2 are always securely related at some relation $\mathcal{R}_2$.

▶ **Theorem 6.** *If $E; \rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}_1} t_2$ and for all values a, b, $\mathcal{R}_1(a, b)$ implies $E; \rho \vdash_{ps} k_1(a) \approx^{\ell}_{\mathcal{R}_2} k_2(b)$, then $E; \rho \vdash_{ps} t_1 \gg= k_1 \approx^{\ell}_{\mathcal{R}_2} t_2 \gg= k_2$.*

Iteration represents loops, which have two parts: an initial value, and a body that produces a value from the previous value. Indistinguishable initial values paired with indistinguishable bodies produce indistinguishable loops, as we can see in the following theorem.

▶ **Theorem 7.** *If $\mathcal{R}_1(a_1, b_1)$ and, for any a, b, $E; \rho \vdash_{ps} k_1(a) \approx^{\ell}_{\texttt{caseR}(\mathcal{R}_1,\mathcal{R}_2)} k_2(b)$ whenever $\mathcal{R}_1(a, b)$, then $E; \rho \vdash_{ps}$ `iter` $k_1\ a_1 \approx^{\ell}_{\mathcal{R}_2}$ `iter` $k_2\ b_1$.*

This rule is conceptually similar to a loop invariant from a Hoare-style logic. $\mathcal{R}_1$ is a property that is initially true and is preserved on each iteration except the final one, while the final iteration guarantees that $\mathcal{R}_2$ holds. The $\texttt{caseR}(\mathcal{R}_1, \mathcal{R}_2)$ function lifts two relations to a single relation over sum types such that $\mathcal{R}_1$ is applied to two left values, $\mathcal{R}_2$ is applied to two right values, and no other combination is related.

Relationship with Equivalence Up-To Taus. Recall that weak bisimulation of ITrees (`eutt`) requires two ITrees to contain the same pattern of interaction with their environment. Our notion of indistinguishability assumes that adversaries distinguish programs purely based on their interactions with the environment. One would thus expect that combining `eutt` with indistinguishability should result in indistinguishability. The following theorem shows this to be the case.

▶ **Theorem 8** (Mixed Transitivity)**.** *If both $E; \rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}_1} t_2$ and $E \vdash t_2 \approx_{\mathcal{R}_2} t_3$ then we can conclude that $E; \rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}_1 \circ \mathcal{R}_2} t_3$.*

This is a very powerful theorem. In particular, many program transformations preserve equality. That is, they take source programs with equivalent-up-to-taus ITree representations to target programs with the same property. Mixed transitivity tells us that compilers built from such transformations also preserve indistinguishability. For instance, since noninterference—the security property we are ultimately considering—is defined as a program being indistinguishable from itself, mixed transitivity supports a very simple proof that the compiler in Section 6 preserves noninterference. While this result might be surprising, it reflects the utility of ITrees and indistinguishability. By looking at which labels can distinguish an ITree from itself, we can discover where leaks are possible.

4.3 Progress-Insensitive Indistinguishability

The type systems that enforce progress-sensitive noninterference are extremely restrictive. Thus, information-flow control literature mostly studies progress-*insensitive* type systems. These type systems enforce noninterference against adversaries who cannot see when a program has begun to silently loop forever. Intuitively, such adversaries believe that silently looping programs could break out of their loops at any moment, and so do not distinguish them from programs which have produced visible events.

In order to support such reasoning, we introduce `pi-seutt`, a progress-insensitive version of indistinguishability for ITrees. This leads to the following definition:

▶ **Definition 9** (`pi-seutt`)**.** *The relation* `pi-seutt`*, the progress-insensitive version of indistinguishability, is defined by modifying the definition of* `seutt` *by completely removing the rules for halting events (all rules in Figure 6) and making every other rule coinductive (this modifies* TauL *and* TauR *in Figure 4 as well as* PrivVisIndL *in Figure 5 and its not-presented symmetric counterpart).*

This relation is strictly more permissive than `seutt`, since it relates every ITree to silently diverging ITrees and private halts. These facts can be formalized in the following theorems:

▶ **Theorem 10.** *If $E; \rho \vdash_{ps} t_1 \approx^{\ell}_{\mathcal{R}} t_2$ then $E; \rho \vdash_{pi} t_1 \approx^{\ell}_{\mathcal{R}} t_2$.*

▶ **Theorem 11.** *Given any ITree t, $E; \rho \vdash_{pi} t_{spin} \approx^{\ell}_{\mathcal{R}} t$.*

▶ **Theorem 12.** *Given any ITree t, if e is a halting event, then* $E; \rho \vdash_{pi} \texttt{Vis}\ e\ k \approx^{\ell}_{\mathcal{R}} t$.

Just as with the progress-sensitive version of indistinguishability, we can show that indistinguishability plays well with the usual ITree combinators. This allows us to prove ITrees indistinguishable in many cases without resorting to hand-rolled coinduction.

▶ **Theorem 13.** *If* $E; \rho \vdash_{pi} t_1 \approx^{\ell}_{\mathcal{R}_1} t_2$ *and* $E; \rho \vdash_{pi} k_1(a) \approx^{\ell}_{\mathcal{R}_2} k_2(b)$ *whenever* $\mathcal{R}_1(a, b)$*, then* $E; \rho \vdash_{pi} t_1 \gg= k_1 \approx^{\ell}_{\mathcal{R}_2} t_2 \gg= k_2$.

▶ **Theorem 14.** *If* $\mathcal{R}_1(a_1, a_2)$ *and for any* a, a', $E; \rho \vdash_{pi} k_1(a) \approx^{\ell}_{\texttt{caseR}(\mathcal{R}_1,\mathcal{R}_2)} k_2(a')$ *whenever* $\mathcal{R}_1(a, a')$*, then* $E; \rho \vdash_{pi} \texttt{iter}\ k_1\ a_1 \approx^{\ell}_{\mathcal{R}_2} \texttt{iter}\ k_2\ a_2$.

Moreover, mixed transitivity again holds, allowing for simple proofs of compiler safety:

▶ **Theorem 15** (Mixed Transitivity)**.** *If both* $E; \rho \vdash_{pi} t_1 \approx^{\ell}_{\mathcal{R}_1} t_2$ *and* $E \vdash t_2 \approx_{\mathcal{R}_2} t_3$ *then we get* $E; \rho \vdash_{pi} t_1 \approx^{\ell}_{\mathcal{R}_1 \circ \mathcal{R}_2} t_3$.

Progress-insensitive indistinguishability behaves differently from the progress-sensitive sibling version in one important way: it does not form a PER. Because it relates a diverging ITree to every other ITree, `pi-seutt` is not transitive. This is not surprising, since progress-insensitive indistinguishability is not a PER [16, 43, 55]. It does, however, retain generalized symmetry, and a weakened but still-useful version of generalized transitivity:

▶ **Theorem 16.** *If* $E; \rho \vdash_{pi} t_1 \approx^{\ell}_{\mathcal{R}} t_2$ *then* $E; \rho \vdash_{pi} t_2 \approx^{\ell}_{\overset{\leftrightarrow}{\mathcal{R}}} t_1$.

▶ **Theorem 17.** *If* $E; \rho \vdash_{pi} t_1 \approx^{\ell}_{\mathcal{R}_1} t_2$, $E; \rho \vdash_{pi} t_2 \approx^{\ell}_{\mathcal{R}_2} t_3$*, and* t_2 *converges along all paths, then* $E; \rho \vdash_{pi} t_1 \approx^{\ell}_{\mathcal{R}_1 \circ \mathcal{R}_2} t_3$.

Where an ITree is considered convergent if it is either a `ret`, a τ followed by a convergent ITree, or a non-halting event followed by a continuation that converges for any input.

Unlike progress-sensitive indistinguishability, we can easily show that loops produce no events that are observable to some adversary at ℓ via `pi-seutt`. Suppose that we want to show that `iter` body a_0 emits no events that are observable to some adversary at ℓ. We can do so by showing that `iter` body a_0 and `ret` b are indistinguishable with some return relation $\mathcal{R}$. This shows that the body of the loop both emits no observable events and, if the loop terminates, it returns a value c where $\mathcal{R}(c, b)$. Importantly, we have not made any statement about whether the loop terminates; we have merely said that it will not produce events, regardless of its termination behavior. We formalize this in the following theorem:

▶ **Theorem 18.** *For any relation* $\mathcal{R}_{inv}$*, if*

$$\mathcal{R}_{inv}(a_0, b) \quad \textit{and} \quad \forall a, \mathcal{R}_{inv}(a, b) \implies E; \rho \vdash_{pi} \textit{body}\ a \approx^{\ell}_{\texttt{leftcase}(\mathcal{R}_{inv},\mathcal{R})} \texttt{ret}\ b,$$

then $E; \rho \vdash_{pi} \texttt{iter}\ \textit{body}\ a_0 \approx^{\ell}_{\mathcal{R}} \texttt{ret}\ b$*, where the relation* `leftcase` *is defined as follows:*

$$\texttt{leftcase}(\mathcal{R}_1, \mathcal{R}_2)(\texttt{inl}(a), b) = \mathcal{R}_1(a, b) \qquad \texttt{leftcase}(\mathcal{R}_1, \mathcal{R}_2)(\texttt{inr}(a), b) = \mathcal{R}_2(a, b)$$

4.4 Noninterference and Interpretation

Recall from Section 2.1 that we can define noninterference using an indistinguishability relation on programs by saying that a program is noninterfering if it is related to itself—given indistinguishable inputs, it will produce indistinguishable computations. We could define noninterference on ITrees using `seutt` (or `pi-seutt`), as they provide such indistinguishability relations by design. This approach produces a sensible definition, but one that assumes an extremely strong adversary.

Consider the following IMP program, where the h_is have label ℓ_h and the l_is have label ℓ_l:

$$\mathsf{if}\ (h_1 = 0)\ \mathsf{then}\ \{h_2 := l_1\}\ \mathsf{else}\ \{h_2 := l_2\}$$

Since the program writes only to secret variables, intuitively this program seems secure. However, according to `seutt`, it is not related to itself at ℓ_l since reading from l_1 and l_2 produce different `get` events with label ℓ_l. All adversaries have the power to observe *reads* of public state, not just writes.

The visibility of public read events is not the only problem. Using just `seutt` also means a computation cannot publicly depend on the result of reading a secret variable, even if a public value were written to that variable. For instance, the following program would also be considered insecure:

$$h := l\,;\mathsf{print}(\ell_l, h)$$

If h cannot change between assignments, this program is intuitively secure, but `seutt` at ℓ_l requires $\mathsf{print}(\ell_l, h)$ to produce the same output regardless of the value of h, which it clearly does not.

On uninterpreted ITrees, `seutt` models a system where both reads and writes are visible to anyone who can see the variable, and the value of a secret variable may silently change between a read and a write. This model makes perfect sense in some contexts—like distributed computation [28]—but we usually consider weaker adversaries.

We can remove these assumptions and model a weaker adversary by interpreting state, as we discussed in Section 2.4. Interpreting these programs would result in two meta-level functions (i.e., Coq functions) which take a state as input and produce an ITree returning an output state. For example in Section 2.4, we define the semantics of an IMP program c as an interpreted ITree—that is, as a function from states to ITrees—not as a single ITree with state events. We thus adjust our notions of indistinguishability and noninterference to account for this semantic construct.

Intuitively, we start with a family of relations $\mathcal{R}_{S,\ell}$ that describes when states are indistinguishable to an adversary at level ℓ and use it to define the following observational equivalence. For technical reasons, we require $\mathcal{R}_{S,\ell}$ to be an equivalence relation at all labels. For IMP, we use a relation $\cong^{\ell}_{\Gamma}$ which only requires states to agree on a variable x if the label of x flows to ℓ.

▶ **Definition 19** (Stateful Indistinguishability). *Two stateful computations p_1 and p_2 are* px-statefully indistinguishable under $\mathcal{R}_{S,\ell}$ and $\mathcal{R}$ at label ℓ *if, for every pair of states σ_1 and σ_2 such that $\mathcal{R}_{S,\ell}(\sigma_1, \sigma_2)$,*

$$E; \rho \vdash_{px} p_1\ \sigma_1 \approx^{\ell}_{\mathcal{R}_{S,\ell} \times \mathcal{R}} p_2\ \sigma_2$$

$$\textit{where } \mathcal{R}_{S,\ell} \times \mathcal{R}((\sigma_1', a_1), (\sigma_2', a_2)) \overset{\triangle}{\iff} \mathcal{R}_{S,\ell}(\sigma_1', \sigma_2') \textit{ and } \mathcal{R}(a_1, a_2)$$

As described above, stateful indistinguishability with $\cong^{\ell}_{\Gamma}$ defines security against an adversary who can observe public writes, but not secret writes or secret reads. This indistinguishability relation leads to a much more common definition of noninterference, and it is the one we will use in our case studies in Sections 5 and 6.

▶ **Definition 20** (Noninterference). *A stateful computation is* px-noninterfering *with state relations $\mathcal{R}_{S,\ell}$ and return relation $\mathcal{R}$ if, given any label ℓ, it is px-statefully indistinguishable from itself under state relation family $\mathcal{R}_{S,\ell}$ and return relation $\mathcal{R}$.*

$$\frac{\Gamma(x) \sqsubseteq \ell}{\Gamma \vdash x : \ell} \qquad \frac{}{\Gamma \vdash n : \ell} \qquad \frac{\Gamma \vdash e_1 : \ell_1 \qquad \Gamma \vdash e_2 : \ell_2}{\Gamma \vdash e_1 \odot e_2 : \ell_1 \sqcup \ell_2}$$

Figure 7 Typing rules for expressions in security-typed Imp.

Shared Typing Rules

$$[\textsc{Skip}]\ \frac{}{\Gamma; pc \vdash_{px} \mathsf{skip} \diamond \bot} \qquad [\textsc{If}]\ \frac{\Gamma \vdash_{px} e : \ell \quad \Gamma; pc \sqcup \ell \vdash_{px} c_1 \diamond \ell_{ex} \qquad \Gamma; pc \sqcup \ell \vdash_{px} c_2 \diamond \ell'_{ex}}{\Gamma; pc \vdash_{px} \mathsf{if}\ (e)\ \mathsf{then}\ \{c_1\}\ \mathsf{else}\ \{c_2\} \diamond \ell_{ex} \sqcup \ell'_{ex}}$$

$$[\textsc{Assign}]\ \frac{\Gamma \vdash_{px} e : \ell \quad pc \sqcup \ell \sqsubseteq \Gamma(x)}{\Gamma; pc \vdash_{px} x \coloneqq e \diamond \bot} \qquad [\textsc{Seq}]\ \frac{\Gamma; pc \vdash_{px} c_1 \diamond \ell_{ex} \qquad \Gamma; pc \sqcup \ell_{ex} \vdash_{px} c_2 \diamond \ell'_{ex}}{\Gamma; pc \vdash_{px} c_1 \,;\, c_2 \diamond \ell_{ex} \sqcup \ell'_{ex}}$$

$$[\textsc{Try}]\ \frac{\Gamma; pc \vdash_{px} c_1 \diamond \ell_{ex} \qquad \Gamma; pc \sqcup \ell_{ex} \vdash_{px} c_2 \diamond \ell'_{ex}}{\Gamma; pc \vdash_{px} \mathsf{try}\ \{c_1\}\ \mathsf{catch}\ \{c_2\} \diamond \ell'_{ex}} \qquad [\textsc{Print}]\ \frac{\Gamma \vdash_{px} e : \ell \qquad pc \sqcup \ell \sqsubseteq \ell'}{\Gamma; pc \vdash_{px} \mathsf{print}(e, \ell') \diamond \bot}$$

Progress-Sensitive Typing Rules

$$[\textsc{While-PS}]\ \frac{\Gamma \vdash_{ps} e : \bot \qquad \Gamma; \bot \vdash_{ps} c \diamond \bot}{\Gamma; \bot \vdash_{ps} \mathsf{while}\ (e)\ \mathsf{do}\ \{c\} \diamond \bot}$$

$$[\textsc{Throw-PS}]\ \frac{}{\Gamma; \bot \vdash_{ps} \mathsf{throw}(\bot) \diamond \bot}$$

Progress-Insensitive Typing Rules

$$[\textsc{While-PI}]\ \frac{\Gamma \vdash_{pi} e : \ell \quad \Gamma; pc \sqcup \ell \sqcup \ell_{ex} \vdash_{pi} c \diamond \ell_{ex}}{\Gamma; pc \vdash_{pi} \mathsf{while}\ (e)\ \mathsf{do}\ \{c\} \diamond \ell_{ex}}$$

$$[\textsc{Throw-PI}]\ \frac{pc \sqsubseteq \ell_{ex}}{\Gamma; pc \vdash_{pi} \mathsf{throw}(\ell_{ex}) \diamond \ell_{ex}}$$

Figure 8 Typing rules for commands in security-typed Imp.

5 Security Sensitive Type Systems For Imp

To see how to use this theory of indistinguishability and ITrees, we now provide an information-security guarantee for an example toolchain for Imp. We begin by verifying two information-flow type systems, and proceed with a simple compiler in Section 6. The two notions of noninterference—progress sensitive and progress insensitive—require slightly different type systems, so we use our ITrees-based semantics to formally verify that both enforce their respective notions of noninterference. As is common in such type systems, we assume $\mathcal{L}$ forms a join semilattice with a unique least element $\bot$ representing "completely public."

5.1 Two Type Systems

Both type systems have two typing judgments: one for expressions and one for commands. The typing judgments for expressions take the form $\Gamma \vdash e : \ell$, where Γ is a map from variables to information flow labels, and ℓ is a label. The judgment says that e is well-typed and depends only on information at or below label ℓ. The typing rules for expressions, which are the same for both type systems, are presented in Figure 7.

The typing rules for commands are presented in Figure 8. As these rules differ between the progress-sensitive and progress-insensitive type systems, we annotate the turnstyles with *ps* for progress-sensitive rules, *pi* for progress-insensitive rules, and *px* for rules that are identical in both type systems.

The typing judgments for commands take the form $\Gamma; pc \vdash_{px} c \diamond \ell_{ex}$, where pc and ℓ_{ex} are information-flow labels. The pc label is a *program-counter label* that tracks the sensitivity of the control flow, while the second label ℓ_{ex} is an upper bound on the label of any exceptions c might raise. Note that the rules listed in Figure 8 do not include any way to type check an inlined Asm program. We address this concern in Section 5.3.

Program-counter labels are a standard technique to control *implicit information flows*—that is, information leaked by the control flow [46]. For example, consider the following program where h has label ℓ_h and l has label ℓ_l with $\ell_h \not\sqsubseteq \ell_l$:

$$\mathsf{if}\ (h = 0)\ \mathsf{then}\ \{l \coloneqq 0\}\ \mathsf{else}\ \{l \coloneqq 1\}$$

While l is only ever explicitly set to constant values, its final value clearly depends on the secret h. The pc label allows us to detect and eliminate these flows by tracking the sensitivity of the control flow. Specifically, the If rule requires the condition's label to flow to the pc in each branch, and the Assign rule requires the pc to flow to the label of the variable being assigned. In the above example, the label of the condition $h = 0$ is ℓ_h, so If requires c_1 and c_2 to type check with a pc where $\ell_h \sqsubseteq pc$. Since $\Gamma(l) = \ell_l$, Assign requires $pc \sqsubseteq \ell_l$. Transitivity of $\sqsubseteq$ thus requires $\ell_h \sqsubseteq \ell_l$, which it does not, so the program correctly fails to type check.

Exceptions can affect the control flow of a program, and therefore can also cause implicit flows of information. Consider the following program.

$$\mathsf{if}\ (h = 0)\ \mathsf{then}\ \{\mathsf{throw}(\ell_h)\}\ \mathsf{else}\ \{\mathsf{skip}\}\ ; l \coloneqq 1$$

Much like the previous example, this program only assigns l to a constant, yet it still leaks the value of h. We use a standard technique [33, 41] that relies on exception labels in the typing judgment. As previously mentioned, the exception label of a program c is an upper bound on the labels of any exception c might raise. To eliminate exception-based leaks, the Seq rule increases the pc label of the second command by the exception label of the first. The Try rule makes similar use of the exception label, increasing the pc in the catch block, as that command only executes if an exception is thrown.

The Skip rule is simple, as skip can never have an effect. Print produces a flow of information to an output channel labeled ℓ', so it checks that ℓ' may safely see both the expression being written and the fact that this command executed.

The rules for while loops and throw statements are different for the progress-sensitive and progress-insensitive type systems, so we handle them separately.

Progress-Sensitive While and Throw Rules. In a progress-sensitive setting, the adversary can observe nontermination. As a result, a program's termination behavior can only safely depend on completely public information. While-PS enforces this requirement in a standard, but highly restrictive way [56]: the loop condition and the pc of the context must both be the fully public label $\bot$. Moreover, any exceptions thrown in the body of the loop could also influence termination behavior, so those must be fully public as well.

Recall from Section 4 that a low observer cannot distinguish between an uncaught secret exception and an infinite loop. Thus non-public exceptions create the same implicit flows as while loops, so Throw-PS restricts exceptions in much the same way as While-PS restricts loops: everything must be fully public.

Progress-Insensitive While and Throw Rules. In a progress-insensitive setting, the adversary cannot see nontermination, so secrets can safely influence the termination behavior of a program. The WHILE-PI rule therefore allows loops with any *pc*. Since both the loop condition and any exceptions the loop body throws influence whether the body is run, WHILE-PI increases the *pc* in the loop body by both the loop guard label and the body's exception label.

For the same reason, THROW-PI is more permissive than its progress-sensitive counterpart. In particular, the label on the exception just needs to be at least as secret as the *pc* label.

5.2 Proving Security

Both type systems enforce their respective notions of noninterference (Definition 20). Unlike many existing proofs of noninterference, our proofs using ITrees proceed by simple induction over the syntax of IMP. This simplicity is made possible by the combination of two facts: our IMP semantics is given by simple induction using ITrees combinators, and those combinators interact with indistinguishability in predictable ways, as described by the metatheory of Section 4.

Type systems are inherently compositional: we are able to conclude that a program is secure knowing nothing about subprograms other than that they also type check. However, our semantic definition of noninterference is *not* fully compositional. To see this, consider the IMP program $p = l \coloneqq h\,; \mathsf{throw}(\ell)$. This program updates the state in an insecure way, assigning a high-security value to a low-security variable, and then throws a low-security exception. In fully interpreted programs, the updated state is part of the return value, but adversaries cannot observe that return value if an exception is thrown (see Section 3), making p semantically secure. However, if we catch the exception, the adversary once again can see the effect of the assignment $l \coloneqq h$. Thus, p does not compose securely.

In order for our type system to enforce security compositionally, it enforces two properties beyond noninterference. Each rules out programs which, like p above, are secure but do not compose securely. The first describes how state and exceptions interact in a secure setting, which will rule out the example program above. The second, called *confinement*, defines how effects are bound by the type system.

Interaction of Exceptions and State. Our first goal is to semantically rule out programs like p above, allowing us to reason compositionally about exception handlers. In order to do so, we need to reason about what state updates are performed before an exception is thrown. However, since in our semantics of IMP we interpret state events while leaving exceptions as ITree events, the result state of an IMP program is forgotten when an exception is thrown.

This correctly models our adversary, who cannot distinguish between private exceptions and silently diverging programs. But in order to achieve compositionality, we need to keep information about the final state before an exception is raised. We accomplish this with a condition on an alternative semantics for IMP programs. In this semantics, exceptions are interpreted into the standard sum type representation before state events are interpreted. This interpretation, $\texttt{interp}\ h_{prog}\ (\texttt{interp}\ h_{exc}\ [\![c]\!]_c)$, is a stateful function that returns a final state along with either a result of type unit or the label of an exception. We can inspect this final state to ensure that the program always takes indistinguishable states to indistinguishable states.

We formalize this property as follows, where the relation $\cong^{\ell}_{\Gamma}$ requires that states agree on a variable x only when $\Gamma(x) \sqsubseteq \ell$, as in Section 4.4.

▶ **Definition 21** (Exceptions-and-State Property). *A command c satisfies the* px–exceptions-and-state property *if* $\texttt{interp}\ h_{prog}\ (\texttt{interp}\ h_{exc}\ [\![c]\!]_c)$ *is statefully indistinguishable from itself under $\cong^{\ell}_{\Gamma}$ and $\top$ at every label ℓ.*

Note the use of $\top$ as the output relation means we ignore whether or not c threw an exception, while we still ensure that the final states are indistinguishable. Ignoring this information in this property is acceptable because it is captured by our standard noninterference condition.

Confinement. Even with the exceptions-and-state property, implicit flows, like the motivating our use of pc labels, can still break compositionality. Confinement fixes this.

In the typing judgment for commands, the pc and ℓ_{ex} labels are both designed to constrain effects. If a command type checks with pc and ℓ_{ex}, it should have no effects visible *below* pc and no (uncaught) exceptions *above* ℓ_{ex}. Semantically, a program has no visible effects below pc if, for any label ℓ where $pc \not\sqsubseteq \ell$, it is indistinguishable from skip. For any uncaught exception terminating a ITree, we simply check that the exception's label flows to ℓ_{ex}. We formalize this idea into the following property called *confinement*.

▶ **Definition 22** (Confinement). *A command c is* px-confined to pc with ℓ_{ex} exceptions*, if, for all labels ℓ such that $pc \not\sqsubseteq \ell$, the following conditions hold.*

1. *c is indistinguishable from skip at ℓ: $\texttt{interp}\ h_{prog}\ [\![c]\!]_c$ and $\texttt{interp}\ h_{prog}\ [\![\textsf{skip}]\!]_c$ are px-statefully indistinguishable under $\cong^{\ell}_{\Gamma}$ and $=$ at ℓ.*
2. *c makes no modifications to the state visible at ℓ: $\texttt{interp}\ h_{prog}\ (\texttt{interp}\ h_{exc}\ [\![c]\!]_c)$ and $\texttt{interp}\ h_{prog}\ (\texttt{interp}\ h_{exc}\ [\![\textsf{skip}]\!]_c)$ are px-statefully indistinguishable under $\top$ and $=$ at ℓ.*
3. *For all initial state heap states h and register states r where c throws an exception, the label of that exception flows to ℓ_{ex}:*

$$E \vdash (\texttt{interp}\ h_{prog}\ (\texttt{interp}\ h_{exc}\ [\![c]\!]_c))(r,h) \approx_{=} \texttt{ret}\ (r', h', \texttt{inr}(\ell'_{ex})) \implies \ell'_{ex} \sqsubseteq \ell_{ex}$$

Together, these definitions restrict programs to those that compose securely, as required by the type system. With this compositionality property, we can prove that our type system enforces the conjunction of all three properties.

▶ **Theorem 23.** *If $\Gamma; pc \vdash_{px} c \diamond \ell_{ex}$, then c is px-noninterfering (Definition 20), satisfies the px–exceptions-and-state property, and is px-confined to pc with ℓ_{ex} exceptions.*

5.3 Semantic Typing and Inline Asm

Both type systems above enforce security, but are highly conservative. Many secure programs fail to type check, notably including any secure program with inlined Asm. To support our goal of cross-language security reasoning and address this concern without the need to introduce a type system for Asm, we provide a *semantic typing* [22] rule.

One would hope that the three conditions discussed above would be sufficient. However, the possibility of undefined Asm behavior (see Section 2.5) necessitates an additional condition. We thus introduce the notion of *inline validity*, which requires inlined Asm to depend only on the initial heap state, not the initial register state, thereby ruling out undefined behavior.

▶ **Definition 24** (Inline Validity). *An Asm program a is* inline-valid *if, given any two register states r_1 and r_2, and any heap states h, then a run with (r_1, h) and (r_2, h) produces the same changes to the heap. That is, if $p = \texttt{interp}\ h_{prog}\ (\texttt{interp}\ h_{exc}\ [\![a]\!]_{\texttt{asm}})$, then*

$$\texttt{printE} \vdash p(r_1, h) \approx_{\top \times =} p(r_2, h).$$

Note that any Asm program that only ever reads from a register after it has written to that register will satisfy this property. We also lift this definition to whole Imp programs by applying it separately to each inlined Asm block.

$$
\begin{array}{llcl}
\text{Registers} & r & ::= & \$0 \mid \$1 \mid \ldots \\
\text{Operands} & o & ::= & r \mid n \\
\text{Instructions} & i & ::= & \textsc{add}\ r_1 \leftarrow r_2, o \mid \textsc{sub}\ r_1 \leftarrow r_2, o \mid \textsc{mul}\ r_1 \leftarrow r_2, o \\
 & & \mid & \textsc{eq}\ r_1 \leftarrow r_2, o \mid \textsc{leq}\ r_1 \leftarrow r_2, o \mid \textsc{not}\ r \leftarrow o \\
 & & \mid & \textsc{mov}\ r_1 \leftarrow r_2 \mid \textsc{load}\ r \leftarrow x \mid \textsc{store}\ x \leftarrow r \mid \mathsf{print}(\ell, r) \\
\text{Branches} & b & ::= & \textsc{jmp}\ \mathrm{A} \mid \textsc{brz}\ r\ \mathrm{A1}\ \mathrm{A2} \mid \textsc{raise}\ \ell \\
\text{Blocks} & B & ::= & \mathrm{A} : i_1 ; \cdots ; i_n ; b \\
\text{Programs} & p & ::= & \textsc{Start} : i_1 ; \cdots ; i_n ; b \\
 & & & B_1 ; \cdots ; B_m
\end{array}
$$

Figure 9 Secure ASM syntax where x is a variable, A is an address, n is a natural number, and ℓ is an information-flow label.

▶ **Definition 25** (Validity). *c is a valid* Imp *program if any inlined* Asm *program it contains is an inline-valid* Asm *program.*

Including validity with our other semantic conditions is sufficient to guarantee security, so we can safely define the following semantic typing rule.

$$
[\textsc{Semantic}]\ \frac{\begin{array}{c} c \text{ is px-noninterfering} \\ c \text{ satisfies the px–exceptions-and-state property} \\ c \text{ is px-confined to } pc \text{ and } \ell_{ex} \\ c \text{ is valid (Definition 25)} \end{array}}{\Gamma; pc \vdash_{px} c \diamond \ell_{ex}}
$$

Adding this new rule to both type systems allows them to reason about multi-language programs including inline Asm and larger systems, even when the syntactic type system cannot reason about every component. Importantly, Semantic is sound from a security perspective. That is, Theorem 23 continues to hold for both extended type systems.

6 Preserving Noninterference Across Compilation

For a compiled language like Imp, noninterference is only part of the story. After all, rather than run Imp code directly, programmers instead compile Imp to Asm and run the Asm. Compilation can change programs significantly, and can introduce insecurity in the process. Thus, we need to ensure that the compiler translates noninterfering Imp programs into noninterfering Asm programs. We now turn our attention to the proof-engineering effort involved in providing such an assurance. In particular, we show that (a) adding exceptions and information-flow labels to Imp does not complicate the proof of compiler correctness, and (b) turning a proof of correctness into a proof of noninterference preservation is simple using mixed transitivity (Theorem 8).

Note that, to build our compiler, we had to fix the number of information-flow labels. We thus specialize our discussion of Imp from Section 5 to the two-point lattice $\mathcal{L} = \{\top, \bot\}$. Using any other finite lattice would require only minimal changes.

6.1 Asm, Its Semantics, and the Compiler

Figure 9 presents the syntax of Asm, the simple assembly language that our compiler targets. An Asm program is a sequence of *blocks*, where each block starts at some address A and consists of a sequence of straight-line instructions followed by a single jump. The first block must be at the special address START.

Most ASM instructions write to exactly one register, computing the written value from a combination of other registers and integer constants. For instance, ADD $\$0 \leftarrow \$1, 1$ takes the value of register \$1, adds one, and stores the result in register \$0. The MOV instruction copies the value of one register into another, while LOAD and STORE move information between registers and the heap. Finally, the PRINT instruction prints information to a stream, depending on the label ℓ.

Jumps are either direct jumps, conditional jumps, or exceptions. A direct jump JMP A immediately moves execution to the beginning of the block with address A. A conditional jump BRZ r A1 A2 move execution to A1 if register r contains zero and A2 otherwise. The RAISE ℓ branch raises an exception. Note that there is no equivalent of catching an exception. We assume that ASM programs always jump to either the address of one of the program's blocks or a special EXIT address.

Rather than representing ASM syntax directly in our Coq code, we take a more compositional approach and represent *sub–Control-Flow Graphs (sub-CFGs)*. These represent the structure of part of an ASM program. While a complete ASM program contains a unique START address, sub-CFGs may contain multiple addresses accessible to the outside. We refer to addresses which are accessible to the outside as *input* addresses. Likewise, sub-CFGs may jump to undefined addresses, whereas complete ASM programs always jump either to a defined address or EXIT. We refer to the undefined addresses a sub-CFG may jump to as its *output* addresses. Thus, a complete ASM program is a sub-CFG with exactly one input address (START) and exactly one output address (EXIT).

Intuitively, sub-CFGs execute starting at some input address, potentially jumping internally several times before they jump to some output address. To represent this pattern, we give sub-CFGs semantics as functions from an address to an ITree that return an address. That is, the semantics of a sub-CFG takes as input the input address at which to start executing, and produces an ITree that returns the output address the program jumps to. This structure is due to Xia et al. [58], and their semantic needed only minor changes to accommodate printing and exception-throwing.

In Xia et al.'s original compiler, IMP code always mapped to complete ASM programs. However, to accommodate exception throwing, our compiler has an extra step of indirection. We map IMP programs to sub-CFGs with exactly one input address but *three* output addresses. The first represents EXIT, as in a complete ASM program, while the second two represent the location of exception handler code. Thus, we compile $\mathsf{throw}(\ell)$ to a jump to the second address if $\ell = \bot$ and the third address if $\ell = \top$. To compile a try-catch command, we place one copy of the handler at the second address and a second copy at the third address. That means any exception will jump to the handler code, regardless of the label of the exception, matching the semantics we gave IMP in Section 3. Note that we still need separate addresses for each label to properly compile *uncaught* exceptions.

For inlined ASM code, we would hope to include it in the compiled code directly with no changes. Unfortunately, if inlined ASM throws an exception with a RAISE instruction, the surrounding IMP code can catch it, but embedding the RAISE unmodified in the compiled output would render the exception uncatchable. To support catching these exceptions, we process inlined ASM to replace RAISE instructions with jumps to the appropriate address. This change causes the inlined exception to properly jump to the handler code.

While the infrastructure described above translates IMP code into sub-CFGs, the end goal of our compiler is to translate complete IMP programs into complete ASM programs.

The final step uses the two output addresses for exceptions by linking the sub-CFG of the complete Imp program with *two different* handlers. The low-security exception handler raises a low-security exception, while the high-security exception handler raises a high-security exception. Thus, any Imp code that raises an exception compiles to a complete Asm program that raises that same exception, while Imp code that catches an exception compiles to a complete Asm program with equivalent control flow.

6.2 Compiler Correctness

We adapt Xia et al.'s [2020] proof of compiler correctness to account for the modifications we have made to Imp and Asm. We formalize correctness by comparing the source and the target programs—after interpretation—using weak bisimilarity. Intuitively, two stateful programs are weakly bisimilar if, whenever they are given *related* start states, the resulting ITrees are weakly bisimilar. We use a return relation $\mathcal{R}_{\text{env}}$. $\mathcal{R}_{\text{env}}$ ignores the register files and compares heaps using a relation $\cong$, which ensures that they map equal variables to equal values. We can now state the correctness theorem for the `compile` function.

▶ **Theorem 26.** *For any initial heap states h_1, h_2 such that $h_1 \cong h_2$, any register states r_1, r_2, and a valid Imp command c, the following equation holds*

$$\texttt{excE} \oplus \texttt{printE} \vdash \texttt{interp}\ h_{imp}\ \ [\![c]\!]_c\ \ (r_1, h_1) \approx_{\mathcal{R}_{env}} \texttt{interp}\ h_{asm}\ \ [\![\texttt{compile}(c)]\!]_{\texttt{asm}}\ \ (r_2, h_2)$$

where $\mathcal{R}_{env}((\_, h_1, \_), (\_, h_2, \_)) \iff h_1 \cong h_2$.

Notably, the changes necessary to adapt Xia et al.'s [2020] proof of correctness to our modified compiler are small and isolated. Most cases of the inductive proof, corresponding to existing language features, needed only cosmetic changes. The new language features required new, but conceptually uninteresting, cases.

6.3 Compiler Security

We finally turn to our ultimate goal: proving that our compiler preserves security. There are two important notions of security for our compiler, both of which require cross-language reasoning. The first is that secure source programs are indistinguishable—by all adversaries—from target programs. This property directly relates an Imp program to an Asm program. The second is that the compiler preserves noninterference. While noninterference itself is a property of a single program, *preserving* noninterference is a property of a translation between two languages, which requires cross-language reasoning.

In order to formalize the idea of a secure Imp program being indistinguishable from its compilation, we need to compare these programs, even though they come from different languages. Because we defined `seutt` purely semantically, we can use it as easily as if we were comparing programs in the same language. We use the return relation $\mathcal{R}^{\ell}_{\Gamma}$, which again ignores the register file and ensures that they map equal *visible* variables to equal values. The theorem then takes the following form.

▶ **Theorem 27.** *For any valid Imp program c, if* `interp` h_{prog} $[\![c]\!]_c$ *is noninterfering with state relation $\mathcal{R}^{\ell}_{\Gamma}$ and return relation $=$, and c is a valid Imp program, then the following* `seutt` *equation holds for any label ℓ, arbitrary register states r_1, r_2 and heap states h_1, h_2 such that $h_1 \cong^{\ell}_{\Gamma} h_2$.*

$$\texttt{excE} \oplus \texttt{printE} \vdash_{px} \texttt{interp}\ h_{prog}\ \ [\![c]\!]_c\ (r_1, h_1) \approx^{\ell}_{\mathcal{R}^{\ell}_{\Gamma}} \texttt{interp}\ h_{prog}\ \ [\![\texttt{compile}(c)]\!]_{\texttt{asm}}\ (r_2, h_2)$$

Our second theorem is simply that our compiler takes noninterfering IMP programs to noninterfering ASM programs.

▶ **Theorem 28** (Noninterference Preservation). *For a valid* IMP *program c, if* `interp` h_{prog} $[\![c]\!]_c$ *is noninterfering with state relations $\mathcal{R}_\Gamma^\ell$ and return relation $=$, then the same holds for its compilation. That is,* `interp` h_{prog} $[\![\texttt{compile}(c)]\!]_{\texttt{asm}}$ *is noninterfering with $\mathcal{R}_\Gamma^\ell$ and $=$. This result holds for both progress-sensitive and progress-insensitive noninterference.*

Notably, the proofs of both theorems follows directly from Theorem 26 and mixed transitivity, showing the utility of mixed transitivity for cross-language security reasoning.

7 Related Work

Goguen and Meseguer [15] introduced noninterference to formalize confidentiality; that is, the intuitive notion that secret data does not leak to an adversary. Volpano et al. [57] enforce progress-insensitive noninterference with a type system, and Volpano and Smith [56] modify the type system to be progress-sensitive. These results led to a long line of work introducing noninterference to an increasing complicated settings [e.g., 1, 4, 31, 33, 34, 41, 42, 45, 46, 52, 54, 62, 65]. Proving the security of these varied type systems led to complicated arguments for noninterference, but also gave rise to an informal library of proof techniques. This work fits into a tradition of proof techniques for noninterference via models.

Most models view noninterference either as a trace (hyper)property or as the result of an indistinguishability relation. These perspectives are not mutually exclusive; we can view two programs as indistinguishable if they produce equivalent traces. Their focus, however, can be quite different. Trace-based models view noninterference as a 2-safety hyperproperty [12]. That is, noninterference can be falsified using finite prefixes of two traces. Specifically, for any interfering program there are two inputs that differ only on secrets but produce distinguishable events after a finite number of steps.

Indistinguishability models focus more on building compositional relations. Pioneered by Abadi et al. [1] and Sabelfeld and Sands [47], these models use PERs and define secure programs as those that are self-related. Two such approaches have yielded recent notable results. First, logical-relations techniques [44] inductively assign each type a binary relation. By constructing the relation to reflect the security requirements of the type, logical relations can reason about information flow control and noninterference [16, 43, 55]. Second, bisimulation approaches directly match up program executions to define indistinguishability [13, 49].

This work straddles these methods. ITrees intuitively collect all possible traces of a program into one infinite data structure. Our binary indistinguishability relation on ITrees is thus combining the hyperproperty model of noninterference with the indistinguishability model. Moreover, our indistinguishability relation is built on top of weak bisimulation. To give meaning to a type system, we also build a small logical relation connecting types to our bisimulation arguments.

To remain practical, many languages provide only progress-insensitive guarantees [e.g., 28, 29, 41, 57], despite the fact that termination channels alone can leak arbitrary amounts of data [6]. Techniques for enforcing progress-sensitive guarantees [46, 56] exist, but have seen little use. Recent work attempts to unify the two by explicitly considering termination leaks as declassifications [11]. Like other models of noninterference [16], `seutt` is naturally progress-sensitive, giving a strong guarantee. We include the progress-insensitive `pi-seutt` to give ITree-based semantics to more-practical systems as well.

A few other works provide machanized proofs of noninterference using different techniques [3, 17, 53]. However, each verifies existing paper proofs [53] or mechanizes an existing proof technique designed for a single-language setting [3, 17](e.g., parametricity [3] or logical relations [17]). This work is unique among mechanizations of noninterference in its use denotational semantics designed to support multi-language settings.

Originally defined by Xia et al. [58], ITrees are based on free monads and their derivatives [23, 24, 51]. This gives rise to a natural interpretation of effects via monad transformers [20, 27] that behave like algebraic-effect handlers [10, 35, 36, 38, 39, 48]. The information-flow community also studies effects deeply since they can leak information. Traditionally, information-flow languages use a program-counter label to reason about effects, as we saw in Section 5. Recent work by Hirsch and Cecchetti [18] connects program-counter labels with monads, giving the former semantics using the latter.

Secure compilation is a very active research area. For instance, Barthe et al. [8] show how to securely compile to a low-level ASM-like target language. However, they use a type system for the target language to enforce security. Other efforts focus on particular language features, such as cryptographic constant time [9]. Moreover, until recently, most work on secure compilation focused on fully-abstract compilation [26]. Unfortunately, Abate et al. [2] recently showed that full abstraction is not sufficient to guarantee preservation of hyperproperties like noninterference. Our Mixed Transitivity theorems (Theorems 8 and 15) show that *equivalence-preserving* compilation does preserve noninterference.

Beyond work on secure compilation, most work on noninterference does not address multiple interacting languages. In one notable exception, Focardi et al. [14] examine the relationship between a process-calculus–based notion of security and simple imperative language with information-flow control, similar to IMP. They translate their version of IMP into CCS and show that they preserve IMP's security guarantees. However, their work contains only pencil-and-paper proofs, rather than formally verifying their translation or its security.

Finally, this work focuses on an approach for verifying language toolchains, but running any program requires hardware. Most language-based security and verification work assumes the hardware is predictable and reliable, but cannot enforce security. Hardware enforcement of information-security properties [59, 64] provides dynamic enforcement of properties like noninterference at the cost of space and power usage. Combining these mechanisms with our approach could reduce the overhead of hardware enforcement for verified-secure programs and provide a means to guarantee that interactions with unverified programs remain safe.

8 Conclusion

This paper uses ITrees to reason semantically about noninterference. Our main technical contributions are two new indistinguishability relations on ITrees that we use to define noninterference—one progress sensitive and one progress insensitive—and their metatheory. While both noninterference definitions are coinductive, our metatheory library supports verifying properties of a language toolchain with no direct use of coinduction.

The two indistinguishability relations describe security in many settings, and we plan to include them in the ITrees library. Importantly, because they do not place any restrictions on the events in an ITree, they can be used for reasoning about a variety of language features. However, we recognize that many variations of noninterference appear in the literature, depending on the adversarial model and desired language features. For instance, *declassification* allows private information to be made public in controlled circumstances, creating a need for more complicated security conditions. We hope that the relations studied here both become the basis of verification efforts larger than our case study *and* that they serve as a starting point for further exploration of indistinguishability relations for ITrees.

References

1 Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 1999. doi: 10.1145/292540.292555.

2 Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, and Jérémy Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *IEEE Computer Security Foundations Symposium (CSF)*, 2019. doi: 10.1109/CSF.2019.00025.

3 Maximilian Algehed and Jean-Philippe Bernardy. Simple noninterference from parametricity. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019. doi: 10.1145/3341693. URL `https://doi.org/10.1145/3341693`.

4 Maximilian Algehed and Alejandro Russo. Encoding dcc in haskell. In *Workshop on Programming Languages and Analysis for Security (PLAS)*, 2017. doi: 10.1145/3139337.3139338.

5 Owen Arden, Jed Liu, and Andrew C. Myers. Flow-limited authorization. In *IEEE Computer Security Foundations Symposium (CSF)*, July 2015. doi: 10.1109/CSF.2015.42.

6 Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *European Symposium on Research in Computer Security (ESORICS)*, pages 333–348. Springer, 2008. doi: 10.1007/978-3-540-88313-5_22.

7 Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. *SIGPLAN Not.*, 49(1):165–178, January 2014. ISSN 0362-1340. doi: 10.1145/2578855.2535839. URL `https://doi.org/10.1145/2578855.2535839`.

8 Gilles Barthe, Amitabh Basu, and Tamara Rezk. Security types preserving compilation. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 2–15, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24622-0.

9 Gilles Barthe, Benjamin Greégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant time". In *IEEE Computer Security Foundations Symposium (CSF)*, 2018. doi: 10.1109/CSF.2018.00031.

10 Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, January 2015.

11 Johan Bay and Aslan Askarov. Reconciling progress-insensitive noninterference and declassification. In *IEEE Computer Security Foundations Symposium (CSF)*, June 2020. doi: 10.1109/CSF49147.2020.00015.

12 Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security (JCS)*, 18(6):1157–1210, 2010. doi: 10.3233/JCS-2009-0393.

13 Riccardo Focardi, Carla Piazza, and Sabina Rossi. Proof methods for bisimulation based information flow security. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2002.

14 Riccardo Focardi, Sabrina Rossi, and Andrei Sabelfeld. Bridging language-based and process calculi security. In *FoSSaCS*, 2005. doi: 10.1007/978-3-540-31982-5_19.

15 Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy (S&P)*, 1982. doi: 10.1109/SP.1982.10014.

16 Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. Mechanized logical relations for termination-insensitive noninterference. *Proc. ACM Program.*

Lang., 5(POPL), January 2021. doi: 10.1145/3434291. URL `https://doi.org/10.1145/3434291`.

17 Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. Mechanized logical relations for termination-insensitive noninterference. *Proc. ACM Program. Lang.*, 5(POPL), jan 2021. doi: 10.1145/3434291. URL `https://doi.org/10.1145/3434291`.

18 Andrew K. Hirsch and Ethan Cecchetti. Giving semantics to program-counter labels via secure effects. *Proceedings of the ACM on Programming Languages*, 5(POPL), January 2021. doi: 10.1145/3434316.

19 Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2013. doi: 10.1145/2429069.2429093.

20 Mauro Jaskelioff. Modular monad transformers. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 64–79, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-00590-9.

21 Limin Jia and Steve Zdancewic. Encoding information flow in Aura. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 17–29, 2009.

22 Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 2015. doi: 10.1145/2676726.2676980.

23 Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 94–105, 2015. doi: 10.1145/2804302.2804319. URL `http://doi.acm.org/10.1145/2804302.2804319`.

24 Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible effects: an alternative to monad transformers. In *ACM SIGPLAN Notices*, volume 48, pages 59–70. ACM, 2013.

25 Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. From c to interaction trees: Specifying, verifying, and testing a networked server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2019. doi: 10.1145/3293880.3294106.

26 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi: 10.1145/1538788.1538814. URL `http://doi.acm.org/10.1145/1538788.1538814`.

27 Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 1995. doi: 10.1145/199448.199528.

28 Jed Liu, Owen Arden, Michael D. George, and Andrew C. Myers. Fabric: Building open distributed systems securely by construction. 25(4–5):319–321, May 2017. doi: 10.3233/JCS-0559.

29 Tom Magrino, Jed Liu, Owen Arden, Chin Isradisaikul, and Andrew C. Myers. Jif 3.5: Java information flow. Software release, 2016. URL `https://www.cs.cornell.edu/jif`.

30 Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2018. URL `http://coq.inria.fr`. Version 8.8.1.

31 Mae P. Milano and Andrew C. Myers. MixT: A language for mixing consistency in geodistributed transactions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018. doi: 10.1145/3192366.3192375.

32 Eugenio Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23, June 1989. Full version, titled *Notions of Computation and Monads*, in Information and Computation, 93(1), pp. 55–92, 1991.

33 Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January 1999. doi: 10.1145/292540.292561.

34 Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *IEEE Symposium on Security and Privacy (S&P)*, 1998. doi: 10.1109/SECPRI.1998.674834.

35 Gordon Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures*, pages 1–24, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45315-4.

36 Gordon Plotkin and John Power. Notions of computation determine monads. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures*, pages 342–356, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-45931-6.

37 Gordon D Plotkin. *A structural approach to operational semantics*. Aarhus university, 1981.

38 Gordon D. Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003.

39 Gordon D Plotkin and Matija Pretnar. Handling Algebraic Effects. *Logical Methods in Computer Science*, 9(4), December 2013. doi: 10.2168/LMCS-9(4:23)2013. URL `https://lmcs.episciences.org/705`.

40 Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. Liquid information flow control. *Proceedings of the ACM on Programming Languages*, 4(ICFP), August 2020. doi: 10.1145/3408987.

41 François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(1):117–158, January 2003. doi: 10.1145/596980.596983.

42 Willard Rafnsson and Andrei Sabelfeld. Compositional information-flow security for interactive systems. In *IEEE Computer Security Foundations Symposium (CSF)*, 2014. doi: 10.1109/CSF.2013.8.

43 Vineet Rajani and Deepak Garg. Types for information flow control: Labeling granularity and semantic models. In *IEEE Computer Security Foundations Symposium (CSF)*, 2018. doi: 10.1109/CSF.2018.00024.

44 John Reynolds. Types, abstraction and parametric polymorphism. *Information Processing*, 1983.

45 Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight information-flow security in haskell. In *ACM SIGPLAN Haskell Symposium*, 2008. doi: 10.1145/1411286.1411289.

46 Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003. doi: 10.1109/JSAC.2002.806121.

47 Andrei Sabelfeld and David Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001. doi: 10.1023/A:1011553200337.

48 Tom Schrijvers, Maciej Piróg, Nicolas Wu, and Mauro Jaskelioff. Monad transformers and algebraic effects: What binds them together. Technical Report CW699, Department of Computer Science, KU Leuven, 2016.

49 Geoffery Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Computer Security Foundations Workshop (CSFW)*, 2003. doi: 10.1109/CSFW.2003.1212701.

50 Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Disjunction category labels. In *Nordic Conference on Security IT Systems (NordSec)*, October 2011. doi: 10.1007/978-3-642-29615-4_16.

51 Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4): 423–436, 2008. doi: 10.1017/S0956796808006758.

52 Tsa-ching Tsai, Alejandro Russo, and John Hughes. A library for secure multi-threaded information flow in haskell. In *IEEE Computer Security Foundations Symposium (CSF)*, 2007. doi: 10.1109/CSF.2007.6.

53 Marco Vassena and Alejandro Russo. On formalizing information-flow control libraries. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, page 15–28, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450345743. doi: 10.1145/2993600.2993608. URL `https://doi.org/10.1145/2993600.2993608`.

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

55 Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, and Deian Stefan. From fine- to coarse-grained dynamic information flow control and back. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2019. doi: 10.1145/3290389.

56 Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *IEEE Computer Security Foundations Workshop (CSFW)*, June 1997. doi: 10.1109/CSFW.1997.596807.

57 Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security (JCS)*, 4(3), 1996. doi: 10.3233/JCS-1996-42-304.

58 Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: Representing recursive and impure programs in coq. *Proceedings of the ACM on Programming Languages*, 4(POPL), January 2020. doi: 10.1145/3371119.

59 Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. Using information flow to design an ISA that controls timing channels. In *IEEE Computer Security Foundations Symposium (CSF)*, June 2019. doi: 10.1109/CSF.2019.00026.

60 Yannick Zakowski, Paul He, Chung-Kil Hur, and Steve Zdancewic. An equational theory for weak bisimulation via generalized parameterized coinduction. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, January 2020.

61 Yannick Zakowski, Calvin Beck, Irene Yoon, Ilya Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, compositional, and executable formal semantics for llvm ir. *Proceedings of the ACM on Programming Languages*, 5(ICFP), 2021.

62 Steve Zdancewic and Andrew C Myers. Secure information flow via linear continuations. *Higher-Order and Symbolic Computation*, 15(2-3), 2002. doi: 10.1023/A:1020843229247.

63 Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. *Communications of the ACM*, 54(11):93–101, November 2011. doi: 10.1145/2018396.2018419.

64 Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2012. doi: 10.1145/2254064.2254078.

65 Lantian Zheng and Andrew C. Myers. End-to-end availability policies and noninterference. In *IEEE Computer Security Foundations Workshop (CSFW)*, 2005. doi: 10.1109/CSFW.2005.16.