Co-Inflow: Coarse-grained Information Flow Control for Java-like Languages

Jian Xiang, Stephen Chong
Harvard University, *{jxiang, chong}@seas.harvard.edu*

2021 IEEE Symposium on Security and Privacy (SP) | 978-1-7281-8934-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/SP40001.2021.00002

***Abstract*—Coarse-grained dynamic information-flow control (IFC) is a good match for imperative object-oriented programming languages such as Java. Java language abstractions align well with coarse-grained IFC concepts, and so Java can be cleanly extended with coarse-grained dynamic IFC without requiring significantly different design patterns or excessive security annotations, and without excessive performance overhead.**

We present Co-Inflow: an extension of Java with coarse-grained dynamic IFC. By careful design choices and defaults, a programmer typically needs to add very few annotations to a Java program to convert it to a Co-Inflow program with relatively good precision. Additional annotations can improve precision. We achieve this tradeoff between precision and annotation burden by instantiating and specializing recent advances in coarse-grained IFC for a Java-like setting, and by using *opaque labeled values*: a restriction of labeled values that the Co-Inflow runtime automatically and securely creates and uses.

We have captured the essence of Co-Inflow in a middle-weight imperative calculus, and proven that it provides a termination-insensitive non-interference security guarantee. We have a prototype implementation of Co-Inflow and use it to evaluate the precision, usability, and potential performance of Co-Inflow.

I. Introduction

The correctness and security of applications often rely on correctly enforcing appropriate restrictions on sensitive information. Increasingly sophisticated and mature language techniques are becoming available to track and control the use of sensitive information in applications (e.g., [1]–[12]). However, these information-flow control (IFC) techniques are often burdensome and invasive, requiring an application programmer to provide many security annotations or to adapt the program design to conform to the enforcement mechanism.

Coarse-grained dynamic information-flow control is a promising approach to track and control sensitive information in an application. It tracks information at the granularity of a *computational container*, for example, a lexically or dynamically scoped section of code. Compared to fine-grained IFC—which tracks information at the granularity of program variables—coarse-grained IFC typically has lower annotation burden and the potential for more efficient dynamic enforcement (since fewer entities are tracked) [6].

Moreover, in a language-based setting where the granularity of computational containers can be adjusted, coarse-grained IFC can be made as precise as fine-grained IFC [13], [14], meaning that we can obtain many of the benefits of coarse-grained IFC without sacrificing precision.

We argue that coarse-grained dynamic IFC is a good match for an imperative object-oriented programming language such as Java. Existing Java language abstractions align well with coarse-grained IFC concepts: method calls are natural computational containers; objects are natural data containers. As such, Java can be cleanly extended with coarse-grained dynamic IFC without requiring significantly different design patterns or excessive security annotations, and without excessive performance overhead.

To develop and explore this argument, we propose the Co-Inflow language: an extension of Java with coarse-grained dynamic IFC. Method calls are computational containers, which means that the programmer does not typically need to add annotations to indicate computational containers. Programmers can annotate additional computational containers, enabling finer-grain tracking of information if needed for precision.

Labeled values—which encapsulate a value and a security label that protects the information associated with the value—are crucial for the expressiveness and usefulness of coarse-grained dynamic IFC [3], [6], [15], [16]. Central to the usability of Co-Inflow, we use *opaque labeled values*: labeled values that the Co-Inflow system automatically constructs and destructs without programmer annotation. We have judiciously chosen when opaque labeled values are constructed to ensure that information-flow tracking in Co-Inflow is precise without programmer burden. Co-Inflow programmers can also use normal labeled values in their programs; opaque labeled values are used to get reasonable precision without burdening the programmer. To ensure security, programmers cannot examine opaque labeled values: their computations access opaque labeled values just like other values.

The key contribution of this paper is the design, implementation, and validation of coarse-grained dynamic information-flow control for a Java-like language such that we align the information-flow control mechanisms with existing language abstractions to achieve a good trade-off between programmer burden, precision, and performance.

DOI 10.1109/SP40001.2021.00002

In Section II we introduce the key ideas of coarse-grained dynamic IFC in a Java setting, and show key features of Co-Inflow, including opaque labeled values.

We capture the essence of Co-Inflow in the CIFC calculus, which we present in Section III. We prove that CIFC programs satisfy a strong information security property (Section IV). CIFC is the formal foundation for our design of Co-Inflow; Section V describes how we extend the concepts from CIFC to Co-Inflow.

We have a prototype implementation of Co-Inflow (via compilation to Java) which we describe in Section VI. We evaluate the feasibility and usefulness of Co-Inflow (Section VII) by porting existing Java applications (i.e., retrofitting strong information security guarantees to Java applications). Our experience indicates that Co-Inflow permits existing Java design patterns with little programmer annotation effort. For example, porting a 15,000 line Java health record application to Co-Inflow requires just 18 additional lines of code to ensure patients' records are accessible only to their assigned doctors.

We explore the precision of Co-Inflow using the IFSpec information-flow control benchmark [17]. Co-Inflow correctly detects all security violations in the benchmark, except for those involving reflection, a feature Co-Inflow doesn't support. Co-Inflow is about as precise (i.e., as few false positives for security enforcement) as other information-flow enforcement mechanisms. However, precision can be further improved by rewriting the benchmark programs to take advantage of Co-Inflow's features. We explore the performance overhead of our prototype implementation as well as perform experiments that indicate how well we might expect a more full-fledged implementation to perform.

II. Co-Inflow Overview

In this section, we present the key concepts of coarse-grained dynamic IFC through a simple example, and introduce the key features of Co-Inflow. Figure 1 shows a snippet of Java code that standardizes the formatting of phone numbers. For presentation purposes we show straight-line code that handles two people; more realistic code would loop over a collection of people.

A. Context labels and field labels

Coarse-grained information-flow control is based on the idea of *computational containers*: contexts in which computation occurs. Each computational container has a *context label*,[1] a security label that is an upper bound on all information that has flowed into the container. As more information enters the container, the context label is increased in order to remain an upper bound. In this work, every method invocation will be a computational container. That is, every stack frame is associated with a context label.

[1]In related work, this is also called a *floating label*, a *current label*, or a *program counter (pc) label*.

```
1  Person alice = ...;
2  Person bob = ...;
3                                          // 1.⊥
4  String a=alice.getPhoneNum();           // 4.L_Alice
5  String a1=formatNum(a);                 // 8.L_Alice
6  alice.setPhoneNum(a1);                  // 11.L_Alice
7
8  String b=bob.getPhoneNum();   // 14.L_Alice ⊔ L_Bob
9  String b1=formatNum(b);       // 18.L_Alice ⊔ L_Bob
10 bob.setPhoneNum(b1);
11
12 String formatNum(String pn)
13 {                    // 5.L_Alice  | 15.L_Alice ⊔ L_Bob
14   String pn1 = ...
15   ...                // 6.L_Alice  | 16.L_Alice ⊔ L_Bob
16   return pn1;        // 7.L_Alice  | 17.L_Alice ⊔ L_Bob
17 }
18
19 class Person {
20  String phoneNum;
21  int personId;
22   ...
23  String getPhoneNum()
24  {                   // 2.⊥        |  12.L_Alice
25   return phoneNum;   // 3.L_Alice  | 13.L_Alice ⊔ L_Bob
26  }
27  void setPhoneNum(String s)
28  {                   // 9.L_Alice  | 19.L_Alice ⊔ L_Bob
29   this.phoneNum = s;//10.L_Alice  | 20.L_Alice ⊔ L_Bob
30     // Program halts, since L_Alice ⊔ L_Bob ⋢ L_Bob
31  }
32   ...
33 }
```

Fig. 1. A phone formatting program. Comments indicate the current label at the program point during execution.

To see how context labels work, let's consider the execution of the code in Fig. 1. (Note that in this subsection we explain the key idea of context labels; the actual operation of Co-Inflow differs and is explained in Section II-C.) Comments on the right indicate the container's context label immediately after executing that line of code. Methods that are invoked twice have two context labels shown, corresponding to each of the two invocations. Numbers indicate the order of execution. Assume that initially (line 3) the context label is $\bot$, the most permissive security label, and that the method call `alice.getPhoneNum()` (line 4) returns data with label L_{Alice}.[2] Thus, after the call, the context label increases from $\bot$ to L_{Alice}, since information with label L_{Alice} entered the computational container.

Line 5 calls `formatNum`, which creates a new computational container for the callee's stack frame. Initially, the context label of the callee's container is L_{Alice}, the same as the caller's context label (since the callee's arguments and the decision to invoke the callee is information that flows into the callee's container, and the caller's context label is an upper bound

[2]For this example, we use labels L_{Alice} and L_{Bob} for Alice's and Bob's information respectively. In general, labels can be application specific.

```
1  Person alice = ...;
2  Person bob = ...;
3                                          // 1.⊥
4  Labeled<String> a =
5    toLabeled(alice.getPhoneNum(),
6              fieldLabelOf(alice)); // 5.⊥
7  Labeled<String> a1 =
8    toLabeled(formatNum(a),
9              fieldLabelOf(alice)); // 10.⊥
10 alice.setPhoneNum(a1);             // 13.⊥
11
12 Labeled<String> b =
13   toLabeled(bob.getPhoneNum(),
14             fieldLabelOf(bob));   // 17.⊥
15 Labeled<String> b1 =
16   toLabeled(formatNum(b),
17             fieldLabelOf(b));     // 22.⊥
18 bob.setPhoneNum(b1);               // 25.⊥
19
20 String formatNum(Labeled<String> pn)
21 {                                  // 6.⊥     | 18.⊥
22   String s = unlabel(pn); // 7.L_Alice | 19.L_Bob
23   ...                             // 8.L_Alice | 20.L_Bob
24   return pn1;                     // 9.L_Alice | 21.L_Bob
25 }
26
27 class Person {
28  String phoneNum;
29  int personId;
30  ...
31  String getPhoneNum() {    // 2.⊥     | 14.⊥
32   String pn = phoneNum;    // 3.L_Alice | 15.L_Bob
33   return pn;               // 4.L_Alice | 16.L_Bob
34  }
35  void setPhoneNum(Labeled<String> ls)
36  {                                 // 11.⊥    | 23.⊥
37   phoneNum = unlabel(ls); // 12.L_Alice | 24.L_Bob
38  }
39  ...
40 }
```

Fig. 2. The phone formatting program with labeled values

on this information). No additional information enters the container, and the method call returns data with label L_{Alice}. The context label of the caller container is already an upper bound on this information and doesn't need to be increased.

On line 6, we invoke `alice.setPhoneNum()`, which creates a new container with context label L_{Alice}. That method call stores the newly formatted phone number into a field of object `alice`. Each object has a *field label* that is an upper bound of the information stored in its fields. If an object's field label is ℓ, when data is read from a field of the object, it is treated as information with label ℓ. When data is stored to a field, the object's field label must be an upper bound of the context label. (The programmer may explicitly increase the field label of an object.) Assume the field label of object `alice` is L_{Alice}. The data being written to field `alice.phoneNum` is labeled L_{Alice}, the context label of the invocation of `alice.setPhoneNum`, so the field write succeeds.

Lines 8–10 format Bob's phone number. On line 8, the method call `bob.getPhoneNum()` returns data with label L_{Bob} (assuming the field label of object `bob` is L_{Bob}) and so after the call, the context label is $L_{Alice} \sqcup L_{Bob}$.[3] At line 10, when we invoke `bob.setPhoneNum(b1)`, the context label of the callee is $L_{Alice} \sqcup L_{Bob}$, which can't flow to the field label of `bob`, and execution halts (line 29) to avoid a possible security violation.

Context labels can soundly track and control the flow of information in a system. However, in some cases enforcement is imprecise. This happened in the example: although the context label was $L_{Alice} \sqcup L_{Bob}$ at line 29 before updating Bob's phone number, the data written to field `bob.phoneNum` depended only on information with label L_{Bob}. The program unnecessarily halts due to such imprecise tracking.

B. Labeled Values

The precision of coarse-grained information-flow tracking can be improved through the use of *labeled values* [3], which is a pair of a value v and a security label ℓ that is an upper bound on the information on which v depends. Intuitively, a labeled value can be passed around by a container without the container's context label being raised so long as the container does not examine or compute on the value v. A container can *unlabel* a labeled value, which raises the context label to at least ℓ and allows the container to compute with v.

In Co-Inflow, the class `Labeled` implements a labeled value, and primitive operations `toLabeled` and `unlabel` respectively create and destruct labeled values. Expression `toLabeled`(e, ℓ) creates a new computational container to evaluate expression e and protects its result with label value ℓ, allowing the programmer to control "label creep" [18] of the context labels.

Figure 2 shows the Figure 1 code modified to use labeled values. Line 6 constructs a labeled value using the `toLabeled` operator for the phone number returned by `getPhoneNum` and the field label of object `alice`. The operator `fieldLabelOf` retrieves the field label of the argument object.

Comments on the right show the context label as this example executes. Because line 6 constructs a labeled value (Alice's phone number, with label L_{Alice}), the context label remains $\bot$ after calling `alice.getPhoneNum()` and `fieldLabelOf(alice)`. The context labels of the calls to `formatNum` and `setPhoneNum` are $\bot$ at the beginning, and rise to L_{Alice} and L_{Bob} after the input labeled values are opened by `unlabel`. The context label of the caller remains $\bot$ throughout the execution, and Bob's phone number is successfully updated without raising the context label: labeled values enabled more precise tracking and control of information flow.

C. Co-Inflow

The use of labeled values can improve precision, but reduces readability, and requires significant programmer effort to add

[3] $\ell_1 \sqcup \ell_2$ is the *join* of ℓ_1 and ℓ_2 and is as restrictive as both ℓ_1 and ℓ_2.

`toLabeled` annotations. Moreover, it may require changing many method signatures to pass and return labeled values, as happens in Figure 2.

To address this, the Co-Inflow runtime in essence inserts `toLabeled` and `unlabel` annotations automatically, without burdening the programmer. Indeed, Figure 1 is Java code, but when treated as Co-Inflow code, the implicit insertion of labeled values ensures that the runtime behavior is almost the same as the code in Figure 2, but without the need for programmer annotations or changed signatures.

Opaque labeled values There are differences between labeled values in Figure 2 and the labeled values that Co-Inflow implicitly inserts, which we refer to as *opaque labeled values*. In particular, for every computational container, Co-Inflow creates an opaque labeled value for its result. The opaque labeled values are then dynamically tracked in subsequent computations. Typically, in coarse-grained IFC languages such as LIO [3], [7], [15], the programmer must supply a label for a `toLabeled` operation (e.g., as in line 6 of Figure 2). Co-Inflow removes this annotation burden by using the context label as the label for opaque labeled values. Although easier (and potentially more precise) than a programmer-supplied label, the label of an opaque labeled value may itself reveal information. To prevent this covert channel, Co-Inflow does not allow code to inspect labels of opaque labeled values (which is why we call them *opaque*).[4]

In addition to creating new computational containers for method calls, Co-Inflow implicitly creates new computational containers for field reads and writes. That is, each field read and write is executed in its own container, which helps prevent context labels creeping upwards unnecessarily.

Co-Inflow's implicit insertion of computational containers and opaque labeled values allows a programmer to get the benefits of coarse-grained IFC with very few annotations. Co-Inflow aligns computational containers for IFC with the natural computational contexts of Java programs.

Object and Field Labels In Co-Inflow, each object has two labels associated with it: an *object label* and a *field label*. The field label is an upper bound on information stored in the object's fields. The object label is an upper bound on the information that has influenced the field label as well as an upper bound on meta-data about the object, such as its run-time class. The field label may increase during execution (through explicit operations) but the object label is fixed. We ensure that if a reference to an object is in a given computational container, then the context label of the container protects the object label, and thus the field label can be examined without ever needing to raise the context label.

[4]Previous work also allowed creation of labeled values without programmer-supplied labels [14], [19]. A minor novelty of our work is to make such labels opaque.

```
1 class Person {
2   int personId;
3   Person(...){ // constructor
4     ...
5     raiseFieldLabel(this,
6                     getLabelById(personId));
7   }
8   ...
9 }
```

Fig. 3. Labeling data appropriately with `raiseFieldLabel`

Co-Inflow has a primitive operation `raiseFieldLabel`(o, ℓ) that raises the field label of object o to at least label ℓ. Primitives `fieldLabelOf` and `getContextLabel` allow programmers to inspect an object's field label and the current context label.

Sources of Labels Co-Inflow programmers must ensure that data entering the system is labeled appropriately. The programmer can do this by using `raiseFieldLabel` to set the field label of objects appropriately. For example, the programmer might ensure that when objects are created from the results of database queries, their label is set appropriately. In the example above, this might be done in the constructor of the `Person` object, as shown in the code snippet of Figure 3.

Co-Inflow also has libraries that facilitate labeling data appropriately as it enters the system, for example, `InputStream` implementations that return labeled values and can be used to ensure that data entering the system is labeled appropriately. We also provide a signature mechanism to facilitate integration with legacy Java libraries (see Section VII).

III. Language Model

We present CIFC, an imperative calculus that captures the essence of Co-Inflow. Inspired by Middleweight Java [20], it enables us to prove formal security guarantees (Section IV).

A. Syntax

Figure 4 shows the syntax of CIFC. For syntactic element s, we write $\overline{s}$ to indicate a (possibly empty) list of s. A program is a list of class definitions and an expression to evaluate. A class definition is a class name, a list of field declarations, and a list of method declarations. A field declaration $cn\ f$ means that objects of the class contain a field named f whose value is either `null` or an object of class cn. The type of a field must be a class name. Methods take a single argument and return a single value. The body of a method is an expression. Valid expressions include variables, booleans, `null`, comparisons, field reads and writes, method calls, assignments, branches, and sequences.

Label-related expressions allow programmers to express and manipulate security labels. Metavariable ℓ ranges over constant security labels. Expression `toLabeled`(e_1, e_2) constructs a new labeled value, by evaluating e_1 in a new computational

P	::=	$\overline{cld}\,;\, e$	Program
cld	::=	**class** $cn\ \overline{fd}\ \overline{md}$	Class definition
fd	::=	$cn\ f$	Field declaration
md	::=	$\tau\ m(\tau\ x)\{e\}$	Method declaration
e	::=		Expression
		x	variable
		$\mid$ `true` $\mid$ `false`	boolean values
		$\mid$ `null`	null
		$\mid e == e$	comparison
		$\mid e.f$	field access
		$\mid e.m(e)$	method invocation
		$\mid$ `new` $cn()$	object creation
		$\mid x = e$	assignment
		$\mid e.f = e$	field write
		$\mid$ `if` e `then` e `else` e	condition branch
		$\mid e\,;\, e$	sequence
		$\mid le$	label related expression
le	::=		Label related expression
		ℓ	label values
		$\mid$ `toLabeled`(e, e)	labeled computation
		$\mid$ `unlabel`(e)	unlabel labeled data
		$\mid$ `labelOf`(e)	get label of labeled value
		$\mid$ `raiseFieldLabel`(e, e)	raise field label
		$\mid$ `fieldLabelOf`(e)	get field label
		$\mid$ `getContextLabel`$()$	get context label

cn Identifiers of class names
f Identifiers of field names
m Identifiers of method names
x Identifiers of variables

τ	::=		Type
		cn	class name
		$\mid$ `bool`	boolean type
		$\mid$ `Label`	label type
		$\mid$ `Labeled` τ	labeled type

Fig. 4. Core Syntax and Types of Co-Inflow Programs

container, and protecting the result with label e_2. Operators `unlabel` and `labelOf` consume a labeled value and return, respectively, the value that is labeled, and the label of the labeled value. Expression `raiseFieldLabel`(e_1, e_2) evaluates e_1 to an object, and raises its field label to at least the value of e_2. Expression `fieldLabelOf`(e) returns the field label of the object that e denotes. Expression `getContextLabel`$()$ returns the current context label.

B. Operational Semantics

We define a small-step operational semantics for CIFC. We extend the surface syntax of expressions (Figure 5) with additional expressions that arise during evaluation: object references ς, labeled values v^{ℓ}, opaque labeled values $v^{[\ell]}$, and `labelData`(e, ℓ) (used to implement construction of labeled values). Values in the language include `null`, label values, object references, labeled values, and opaque labeled values. In addition to the syntactic restrictions shown in Figure 5, we also require that opaque labeled values are not directly nested.

A *configuration* is either a pair $\langle \Delta, \mathcal{H} \rangle$ (where Δ is a stack of computational containers and $\mathcal{H}$ is a heap) or the special error configuration *Exception* which is used to indicate null-pointer

e	::=		Expressions
		$\ldots$	
		$\mid \varsigma$	object references
		$\mid v^{\ell}$	runtime `Labeled` values
		$\mid v^{[\ell]}$	opaque labeled values
		$\mid$ `labelData`(e, ℓ)	create labeled values
v	::=		Values
		ς	object reference
		$\mid \ell$	labels
		$\mid$ `null`	null
		$\mid v^{\ell}$	runtime `Labeled` values
		$\mid v^{[\ell]}$	opaque labeled values
κ	::=		Continuations
		$\bullet == e \mid v == \bullet \mid \bullet.f \mid \bullet.f = e \mid v.f = \bullet \mid \bullet.m(e)$	
		$\mid v.m(\bullet) \mid x = \bullet \mid$ `if` $\bullet$ `then` e `else` $e \mid \bullet\,;\, e$	
		$\mid$ `labelData`$(\bullet, \ell) \mid$ `labelOf`$(\bullet)$	
		$\mid$ `toLabeled`$(e, \bullet) \mid$ `unlabel`$(\bullet) \mid$ `fieldLabelOf`$(\bullet)$	
		$\mid$ `raiseFieldLabel`$(\bullet, e) \mid$ `raiseFieldLabel`$(v, \bullet)$	

Fig. 5. Deep syntax, values, and continuations

dereferences, security violations, and other run-time errors. In this formalism, error configurations halt execution; we discuss more precise handling of exceptions in Section V. A heap $\mathcal{H}$ is a function from *object references* ς to objects $\langle cn, \mathbb{F}, \ell_f, \ell_o \rangle$, where cn is the class name of the object, function $\mathbb{F}$ maps field names to values, label ℓ_f is the *field label* for the object (an upper bound on the information stored in the fields), and label ℓ_o is the *object label* for the object (an upper bound on the information associated with the object reference ς and the object's field label).

A computational container is a tuple (e, ρ, pc, θ) where e is the expression to evaluate, ρ is a stack of continuations (i.e., expressions with a hole; see Figure 5), pc is the container's floating context label, and environment θ maps local variables to values. A configuration has a stack of containers Δ where the top of the stack is the currently executing computation; we refer to the container at the top of the stack as the *current container*. We write $[]$ for the empty stack, and $s :: S$ for the stack with s pushed on stack S. A *final configuration* is a configuration of the form $\langle (v, [], pc, \theta) :: [], \mathcal{H} \rangle$, that is, where the container stack has just a single computational container with a value v and an empty continuation stack.

We give the semantics of CIFC as a small-step judgment $CT \vdash \langle \Delta, \mathcal{H} \rangle \rightarrow \langle \Delta', \mathcal{H}' \rangle$, where class table CT is a map from class names to class definitions. Given a list of class declarations, we can construct a corresponding class table in the obvious way (assuming that class names are distinct). Given a program $\overline{cld}\,;\, e$, the initial configuration is $\langle (e, [], \bot, \emptyset) :: [], \emptyset \rangle$, where $\bot$ is the most permissive security label, and $\emptyset$ denotes the empty heap and empty variable environment. The initial configuration is evaluated using the class table corresponding to the program's class declarations. The class table does not change during evaluation. Configurations should be *well-formed* (defined in Appendix B). For example, for every object $\langle cn, \mathbb{F}, \ell_f, \ell_o \rangle$ in the heap, the domain of field map $\mathbb{F}$ should be all and only the fields in the class definition for cn.

E-FieldRead
$$\frac{\delta = (v_o.f, \rho, pc, \theta) \qquad (\varsigma, \ell) = openOpaque(v_o) \\ \mathcal{H}(\varsigma) = \langle cn, \mathbb{F}, \ell_f, \ell_o \rangle \qquad pc' = pc \sqcup \ell_f \sqcup \ell_o \sqcup \ell \\ v = \mathbb{F}(f) \qquad \delta' = (v, [], pc', \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \delta :: \Delta, \mathcal{H} \rangle}$$

E-MethodCall
$$\frac{\delta = (v_o.m(v), \rho, pc, \theta) \qquad (\varsigma, \ell) = openOpaque(v_o) \\ \mathcal{H}(\varsigma) = \langle cn, \mathbb{F}, \ell_f, \ell_o \rangle \qquad pc' = pc \sqcup \ell \sqcup \ell_o \\ lookup\_md(CT, cn, m) = \tau_r\ m(\tau_a\ a)\{e_{body}\} \\ \theta' = [a \mapsto v] \qquad \delta' = (e_{body}, [], pc', \theta')}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \delta :: \Delta, \mathcal{H} \rangle}$$

E-FieldWrite
$$\frac{\delta = (v_o.f = v, \rho, pc, \theta) \qquad (\varsigma, \ell) = openOpaque(v_o) \\ (v', \ell') = openOpaque(v) \qquad \mathcal{H}(\varsigma) = \langle cn, \mathbb{F}, \ell_f, \ell_o \rangle \\ (pc \sqcup \ell \sqcup \ell') \sqsubseteq \ell_f \qquad \delta' = (v, \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H}[\varsigma \mapsto \langle cn, \mathbb{F}[f \mapsto v'], \ell_f, \ell_o \rangle] \rangle}$$

E-Return
$$\frac{\delta = (v_r, [], pc, \theta) \qquad (v, \ell) = openOpaque(v_r) \\ \delta' = (e, \rho, pc', \theta') \qquad \delta'' = (v^{[pc \sqcup \ell]}, \rho, pc', \theta')}{CT \vdash \langle \delta :: \delta' :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta'' :: \Delta, \mathcal{H} \rangle}$$

E-New
$$\frac{\delta = (\texttt{new}\ cn(), \rho, pc, \theta) \qquad \delta' = (\varsigma, \rho, pc, \theta) \\ \varsigma \notin dom(\mathcal{H}) \qquad \mathbb{F} = \{f \mapsto \texttt{null} \mid f \in fields(CT, cn)\}}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H}[\varsigma \mapsto \langle cn, \mathbb{F}, pc, pc \rangle] \rangle}$$

Fig. 6. Selected inference rules

Given a program $P = \overline{cld}; e$, we write $P \Downarrow \langle \Delta, \mathcal{H} \rangle$ to mean that the initial configuration for P can take zero or more steps to final configuration $\langle \Delta, \mathcal{H} \rangle$, using the class table corresponding to $\overline{cld}$.

Selected inferences rules for the semantics are given in Figures 6 and 7, and selected auxiliary functions in Figure 8; remaining rules and auxiliary functions are in Appendix E.

The semantics ensures certain invariants hold during execution. As is standard in coarse-grained information-flow control, given a computational container $\delta = (e, \rho, pc, \theta)$, the context label pc is always an upper bound on the labels of any (unlabeled) value that appears in expression e, continuation stack ρ or the range of variable context θ. So, for example, rule E-Unlabel takes a labeled value v^{ℓ}, and unlabels it, ensuring that the context label is increased to be at least ℓ and allowing value v to be used in the current container.

Another invariant is that for any object $\langle cn, \mathbb{F}, \ell_f, \ell_o \rangle$ in the heap, the field label ℓ_f is an upper bound of the labels of any (unlabeled) value in the range of field map $\mathbb{F}$, and object label ℓ_o is an upper bound on information that may be learned by having a reference to this object, by the class cn of the object, and by the current value of the field label. The object label

E-ToLabeled
$$\frac{\delta = (\texttt{toLabeled}(e, v), \rho, pc, \theta) \\ e \text{ contains no assignments} \qquad (\ell_v, \ell) = openOpaque(v) \\ pc' = pc \sqcup \ell \qquad \delta' = (\texttt{labelData}(e, \ell_v), [], pc', \theta) \\ \delta'' = (\texttt{toLabeled}(e, v), \rho, pc', \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \delta'' :: \Delta, \mathcal{H} \rangle}$$

E-LabelData
$$\frac{\delta = (\texttt{labelData}(v, \ell_v), \rho, pc, \theta) \\ pc \sqsubseteq \ell_v \qquad \delta' = (v^{\ell_v}, \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-Unlabel
$$\frac{\delta = (\texttt{unlabel}(v'), \rho, pc, \theta) \qquad (v^{\ell}, \ell') = openOpaque(v') \\ pc' = pc \sqcup \ell \sqcup \ell' \qquad \delta' = (v, \rho, pc', \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-LabelOf
$$\frac{\delta = (\texttt{labelOf}(v'), \rho, pc, \theta) \qquad (v^{\ell}, \ell') = openOpaque(v') \\ pc' = pc \sqcup \ell' \qquad \delta' = (\ell, \rho, pc', \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-FieldLabelOf
$$\frac{\delta = (\texttt{fieldLabelOf}(v_o), \rho, pc, \theta) \\ (\varsigma, \ell) = openOpaque(v_o) \qquad \mathcal{H}(\varsigma) = \langle cn, \mathbb{F}, \ell_f, \ell_o \rangle \\ pc' = pc \sqcup \ell \sqcup \ell_o \qquad \delta' = (\ell_f, \rho, pc', \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-RaiseFieldLabel
$$\frac{\delta = (\texttt{raiseFieldLabel}(v_o, v_\ell), \rho, pc, \theta) \\ (\varsigma, \ell) = openOpaque(v_o) \qquad (\ell_f{}', \ell') = openOpaque(v_\ell) \\ \mathcal{H}(\varsigma) = \langle cn, \mathbb{F}, \ell_f, \ell_o \rangle \qquad pc \sqcup \ell \sqcup \ell' \sqsubseteq \ell_o \\ \ell_f \sqsubseteq \ell_f{}' \qquad \delta' = (v_o, \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H}[\varsigma \mapsto \langle cn, \mathbb{F}, \ell_f{}', \ell_o \rangle] \rangle}$$

Fig. 7. Selected inference rules for label-related expressions

$$openOpaque(v) \stackrel{def}{=} \begin{cases} (v_0, \ell) & \text{if } v = v_0^{[\ell]} \\ (v, \bot) & \text{otherwise} \end{cases}$$

Fig. 8. Selected auxiliary functions

and field label for a newly created object are set to the context label of the container that created it (see rule E-New). The object label remains fixed for the lifetime of the object but the programmer may raise the field label of an object during execution (using the `raiseFieldLabel` expression; see rule E-RaiseFieldLabel).[5] Rule E-FieldWrite shows the field label of an object must be an upper bound of information

[5]Having two labels for an object where the object label is fixed and the field label floats upwards is an instantiation of the ideas presented by Buiras et al. [16], adapted for an object-oriented setting. We use a single field label for all fields of an object, a design decision that trades off precision and usability.

written into a field of the object.

The programmer can create new computational containers using the `toLabeled`(e_1, e_2) expression. Rule E-TOLABELED pushes a new computational container onto the container stack to evaluate e_1. The expression `labelData` is used to construct a labeled value after e_1 is evaluated, checking that e_2 is a suitable label for the labeled value. Note that rule E-TOLABELED requires that expression e_1 does not contain any assignments to local variables. This is to ensure that any side-effects that occur during evaluation of e_1 do not inappropriately escape the scope of the computational container.

An innovation of this work is that new computational containers are also created for method calls and field reads, as demonstrated by rules E-METHODCALL and E-FIELDREAD. This automatic creation of computational containers fits naturally with Java-like computational patterns to reduce label creep without significant programmer annotation.

Similarly, while the programmer has the ability to explicitly create and manipulate labeled values (using expressions `toLabeled`, `unlabel`, and `labelOf`), the semantics of CIFC automatically creates *opaque labeled values* whenever a computational container finishes evaluation. In rule E-RETURN, where a container has finished evaluating an expression to value v_r, the parent container is given an opaque labeled value $v^{[pc \sqcup \ell]}$, where $(v, \ell) = \mathit{openOpaque}(v_r)$. Value v_r may itself be an opaque labeled value; auxiliary function $\mathit{openOpaque}$ (Figure 8) returns $(v_r, \bot)$ if v_r is not an opaque labeled value, and (v, ℓ) if v_r is opaque labeled value $v^{[\ell]}$. The inference rules automatically open opaque labeled values when the underlying value is needed (see, e.g., rule E-RETURN). As described in Section II-C, this provides much of the benefit of labeled values without the programmer needing to provide `toLabeled` and `unlabel` annotations. Note that opaque labels are not directly examinable by the programmer: she has no ability to determine whether a value is an opaque value and no ability to examine its label.

Because the inference rules are opening opaque labels and performing operations that are appropriately tainted by the labels of opaque labeled values, these rules are essentially creating (and immediately destroying) computational containers, again preventing context label creep without undue programmer annotation burden. For example, in rule E-FIELDWRITE, which evaluates field write $v_o.f = v$, if v_o and v are opaque labeled values with labels ℓ and ℓ', the rule updates the field as if it occurred in a container with context label $pc \sqcup \ell \sqcup \ell'$.

C. Type system

The type system of CIFC does *not* enforce security: security in CIFC is achieved via the run-time mechanisms that track and control the propagation of information during execution. The type system does ensure that the evaluation of CIFC does not get stuck.[6] Figure 4 gives the types used in CIFC. We have

[6]Note that evaluation may result in an error configuration, due to dereferencing null values or security violations.

$$\textsc{Unlabel}\quad \frac{CT, \Gamma, \mathcal{H} \vdash e : \texttt{Labeled}\ \tau}{CT, \Gamma, \mathcal{H} \vdash \texttt{unlabel}(e) : \tau}$$

$$\textsc{LabelOf}\quad \frac{CT, \Gamma, \mathcal{H} \vdash e : \texttt{Labeled}\ \tau}{CT, \Gamma, \mathcal{H} \vdash \texttt{labelOf}(e) : \texttt{Label}}$$

$$\textsc{FieldLabelOf}\quad \frac{CT, \Gamma, \mathcal{H} \vdash e : cn}{CT, \Gamma, \mathcal{H} \vdash \texttt{fieldLabelOf}(e) : \texttt{Label}}$$

$$\textsc{LabeledVal}\quad \frac{CT, \Gamma, \mathcal{H} \vdash v : \tau \qquad CT, \Gamma, \mathcal{H} \vdash \ell : \texttt{Label}}{CT, \Gamma, \mathcal{H} \vdash v^{\ell} : \texttt{Labeled}\ \tau}$$

$$\textsc{OpaqueLabeledVal}\quad \frac{CT, \Gamma, \mathcal{H} \vdash v : \tau \qquad CT, \Gamma, \mathcal{H} \vdash \ell : \texttt{Label}}{CT, \Gamma, \mathcal{H} \vdash v^{[\ell]} : \tau}$$

$$\textsc{Labeled}\quad \frac{CT, \Gamma, \mathcal{H} \vdash e_1 : \tau \qquad CT, \Gamma, \mathcal{H} \vdash e_2 : \texttt{Label}}{CT, \Gamma, \mathcal{H} \vdash \texttt{toLabeled}(e_1, e_2) : \texttt{Labeled}\ \tau}$$

Fig. 9. Selected typing rules for expressions

typing judgments for expressions, continuations, continuation stacks, containers, container stacks and configurations. The judgment for expressions has the form $CT, \Gamma, \mathcal{H} \vdash e : \tau$ where typing environment Γ maps variables to types. Figure 9 shows selected rules for this judgment. The typing system is mostly standard, although note that in rule OPAQUELABELEDVAL, an opaque labeled value $v^{[\ell]}$ has the same type as v, allowing opaque labeled values to be used wherever non-labeled values can be used. Additional rules for all type judgments are presented in Appendix D.

IV. SECURITY GUARANTEE

We prove that our calculus CIFC satisfies noninterference [18], [21]: a well-known strong information security property that, intuitively, guarantees that publicly-observable outputs of a system do not reveal any confidential information. The exact definition of what is observable leads to subtly different security guarantees. We focus on *termination-insensitive noninterference* (TINI) [21], a common variant that assumes the termination and timing behavior of the system is not observable. We do, however, assume that an adversary can observe significant portions of program configurations. This over-approximates what might be observable by an adversary in a setting that contains explicit output channels.

We assume that there is a label ℓ_A such that the adversary that is permitted to observe any and all information with label ℓ such that $\ell \sqsubseteq \ell_A$. Let $L = \{\ell \mid \ell \sqsubseteq \ell_A\}$ be the set of labels that the adversary may observe; intuitively, data and containers associated with these labels are publicly observable, or "low security". Let $H = \{\ell \mid \ell \not\sqsubseteq \ell_A\}$ be the labels that cannot flow to ℓ_A; these are the confidential, or "high security" labels.

To state TINI formally, we must define *low equivalence* of configurations. Intuitively, two $\langle \Delta_1, \mathcal{H}_1 \rangle$ and $\langle \Delta_2, \mathcal{H}_2 \rangle$ are low equivalent (written $\langle \Delta_1, \mathcal{H}_1 \rangle \approx_L \langle \Delta_2, \mathcal{H}_2 \rangle$) if the publicly-observable portions of the two configurations are the same, that is, the adversary would not be able to distinguish the configurations. A complete definition of low equivalence is given in Appendix C.

Using low equivalence, we can state our theorem that CIFC enforces TINI. If we have a program that we execute with two different initial high-security inputs and both executions terminate, then the final configurations are low equivalent. That is, the adversary cannot distinguish the results of the executions even though they compute with different high-security information. Without loss of generality, we assume that there is a distinguished class cn_{main} with a method $main$ that is invoked with the high-security values.

THEOREM. Let $\overline{cld}$ be a sequence of class declarations with distinguished class cn_{main} with method $main$. Let $\ell \in H$ be a high-security label, let e_1 and e_2 be expressions, and let

$$P_1 = \overline{cld}\,;\ \texttt{new}\ cn_{main}().main(\texttt{toLabeled}(e_1, \ell))$$

and

$$P_2 = \overline{cld}\,;\ \texttt{new}\ cn_{main}().main(\texttt{toLabeled}(e_2, \ell)).$$

If $P_1 \Downarrow \langle \Delta_1, \mathcal{H}_1 \rangle$ and $P_2 \Downarrow \langle \Delta_2, \mathcal{H}_2 \rangle$ then $\langle \Delta_1, \mathcal{H}_1 \rangle \approx_L \langle \Delta_1, \mathcal{H}_1 \rangle$.

The proof of Theorem IV is standard: we show that as the two executions advance, their configurations continue to be low equivalent. We have formalized CIFC and the proof of Theorem IV in Coq.[7] A sketch of the proof is in Appendix C.

V. FROM CIFC TO CO-INFLOW

The Co-Inflow language extends the ideas that proven secure in CIFC to cover a more realistic Java-like language. Although we have no formal proof of the security guarantees of Co-Inflow, the proof for CIFC provides assurance that the design of Co-Inflow correctly tracks and controls information flow.

Additional language features Java-like languages distinguish expressions from statements, and Co-Inflow has a form of `toLabeled` that accepts statements. Arrays are treated similarly to objects: each array is associated with an object label and a field label. The field label is an upper bound on the information conveyed by knowing the array's elements. Constructors are treated analogously to methods. Initializers are treated as code that runs inside the constructor's container. Lambda expressions are similar to methods: a new container is created for each invocation of the lambda-expression function. Every class has an object label and field label that protect its static fields: the field label is an upper bound on the information contained in the class's static fields. Static initializers and static methods are treated similarly to methods.

[7] https://github.com/HarvardPL/CIFC/tree/master/coinflow

Control-flow constructs such as loops, breaks, continues, and returns do not pose any challenge: they are computations that execute within a container and do not need any special handling in Co-Inflow. Exceptions, however, can create information flows, as a container may exit either normally or exceptionally. Soundly tracking information flow from exceptions while allowing existing programming patterns is challenging. A common approach (which we take in the formalism of Section III) is to prevent exceptions from being caught. However, many Java programs rely extensively on exceptions, meaning that dramatically changing the behavior of exceptions (such as with delayed exceptions [22]) would require significant rewriting of Java programs. In our prototype implementation, we partially track information flow due to exceptions by raising the context label of the container that catches an exception to the context label of the container that threw the exception. This correctly tracks information flow due to thrown exceptions but may fail to soundly track information flow due to normal termination. We are currently investigating dynamic discovery of information flows due to exceptions—in the style of Shroff et al. [23]—which can provably bound the information leaked due to exceptional control flow while allowing legacy Java programming patterns.

Application-specific label lattices In general, the appropriate lattice of security labels to use will be specific to the application. For example, the lattice may be the powerset of application users, or a fixed and finite set of security levels. To enable this, a Co-Inflow application can define its own labels and the flow-to relation over labels. The classes that implement labels and the flows-to relation are specified at the beginning of execution of a Co-Inflow program, and cannot be changed during execution.

Sources and sinks It is important for an application programmer to be able to specify appropriate labels for data at *sources* (i.e., where information enters the application) and at *sinks* (i.e., where information leaves the application or is used in security-relevant operations). Co-Inflow provides mechanisms for programmers to easily indicate sources and sinks and the labels associated with them. That is, if a container accesses a source, its context label will be increased to the label associated with the source, and a container accessing a sink will be checked to ensure that its context label flows to the sink's label. Sources and sinks can be fields of objects, parameters to methods, or method return values.

Enforcement actions Security violations occur when inappropriately labeled data flows to a sink or an object's field, and when operations such as `raiseFieldLabel` and `labelData` do not satisfy their required label constraints. Co-Inflow can be configured to take one of three possible actions when a security violation occurs: (1) the security violation is logged but execution continues; (2) an exception is thrown, which can be caught and handled by the application; or (3) program execution is terminated. Option 1 is useful to audit the security of a system without intervening in execution, but may lead

to arbitrary information leakage. Option 2 provides the most flexibility, allowing programmers to handle violations appropriately, but as with other Co-Inflow exceptions, fails to track information flow due to normal termination (i.e., when the enforcement mechanism does not detect a security violation).

VI. Implementation

We have a prototype implementation of Co-Inflow via compilation to Java.[8] The implementation is about 3,500 lines of code and uses the Spoon library [24] for AST rewriting. The resulting Java program explicitly tracks the current label of containers and the field and object labels of objects, and constructs and destructs labeled values and opaque labeled values. To achieve this, it uses the Co-Inflow runtime system: a Java library comprising about 550 LoC; we describe it below.

Compilation After disambiguation and type checking, compilation of a Co-Inflow program goes through three stages.

First, an analysis is performed to ensure that there are no local side-effects in `toLabeled`'s computation (first argument). As in CIFC, this ensures that information does not inadvertently escape from a container.

Second, we perform a dataflow analysis to determine which expressions will evaluate to opaque labeled values, and thus, which expressions must be opened (i.e., unlabeled) before being used. Whereas the CIFC type system treats opaque labeled values the same as values and dynamically opens opaque labeled values as necessary, for Co-Inflow we precisely track which expressions will evaluate to opaque labeled values and statically insert coercions to unlabeled values. This is an intraprocecural analysis, as we ensure that all method arguments are opaque labeled values (although for efficiency and compatibility with Java methods, we pass labels of arguments via the Co-Inflow runtime system, described below).

Third and finally, we implement labeled values, opaque labeled values, containers, and Co-Inflow primitives by inserting calls to the Co-Inflow runtime system.

Runtime System Labeled values and opaque labeled values are represented as objects of the classes `Labeled<T>` and `OpaqueLabeled<T>` respectively. Each of these objects contains a value of parameter type `T` and a security label. The runtime system contains utility methods `toOpaqueLabeled` (which constructs an opaque labeled value whose label is the current context label) and `openOpaque` (which opens an opaque labeled value, returning the value and raising the context label as needed).

Co-Inflow uses a shadow stack to keep track of computational containers. The top of the stack is the current container. We insert calls to the Co-Inflow runtime to push and pop containers as appropriate: just before and after method calls, constructor calls, `toLabeled` expressions, and field reads.[9] Each container on the stack consists of a context label and (for containers for method and constructor calls) the labels of arguments to the callee. That is, all arguments to callees are treated as opaque labeled values, but instead of being represented as objects of class `OpaqueLabeled<T>`, the labels are passed on the shadow stack. This avoids changing the signatures of methods and constructors and allows correct dynamic dispatch of calls.

Every object in Co-Inflow requires an object label and a field label. During compilation we insert two instance fields into class declarations to track these. Accessing the object label or field label of an object is translated as accesses to these new fields. Before field reads we insert calls to the runtime system to raise the context label to the object's field label, and before field writes we check that the context label may flow to the object's field label. Although arrays are treated like objects in Co-Inflow, when compiling we cannot insert additional fields into arrays to track the object and field labels. Instead, the runtime system maintains a map from the array to the object and field labels for the array. This correctly tracks the object and field labels for arrays, but introduces additional overhead.

In Co-Inflow, initializers are treated as code that executes inside the container of the invoked constructor. To implement this, we rewrite field initializers and instance initializers as a private Java method that is invoked from the constructor.

Co-Inflow primitives (such as `labelOf`, `getContextLabel`, etc.) and source and sink annotations are simply translated to calls to the runtime system. Label checks are inserted as appropriate to correctly track and control information flow.

Expression `toLabeled`(e_1, e_2) is implemented by first evaluating e_2 in the current container, and then pushing a new container to evaluate e_1. Expression e_1 can be a Java expression (including Supplier Lambdas to allow execution of statements). After checking that the context label for the new container can flow to the label e_2, we construct a labeled value (the result of e_1 labeled with the result of e_2) and pop the container.

To track information flow arising from thrown exceptions, we modify catch blocks. When a catch block is executed, the container that threw the exception will still be at the top of the shadow stack; the catch block unwinds the shadow stack until the catch block's container is at the top, and raises its context label appropriately. This ensures that the context label of the container that catches the exception is at least as restrictive as the context label of the container that threw the exception.

Interaction with Legacy Java Code Our prototype implementation of Co-Inflow is based on source-code rewriting. Problems arise when interacting with legacy code for which the source code is not available for rewriting: Co-Inflow cannot precisely track information flow in code that is available only as bytecode. We could implement Co-Inflow as a byte-code rewriter (and thus be able to rewrite bytecode for which source code is not available), but would still encounter similar issues

[8] https://github.com/HarvardPL/Co-InflowPrototype

[9] We perform some optimizations and remove some pushes and pops when they are unnecessary. E.g., field reads may not require a push and pop.

for native code (i.e., code for which neither source code nor bytecode is available).

For objects that we cannot rewrite, we track their object and field labels using a map, similar to our mechanism for tracking these labels for arrays.

In order to handle legacy library methods, we allow programmers to specify the read effect and write effect of a method call. The read effect is a set of objects that must over-approximate the objects that the method may read and similarly the write effect must over-approximate the objects that the method may write. For example, a method `o.toString()` might have a read effect of `{o}`, and an empty write effect.

When we execute a legacy method m, we create a new container which serves as the computational container for the entire method call to m (including any other non-rewritten code it invokes). We raise container's context label to the join of field labels of all objects in the method's read effect before executing the call. If m calls into some rewritten code m', then information flow between m' and m will be correctly tracked. After the call finishes, we check that the context label for m flows to the field label of each object in the write effect. This conservatively approximates the information flows that may occur during the method call. We also provide a signature mechanism to specify sinks for legacy methods.

VII. Evaluation

We have evaluated Co-Inflow, focusing on validating its feasibility, precision, and usability. We ported three existing Java applications to Co-Inflow to evaluate feasibility and usability, measuring performance overhead and how many code changes were required. We choose these applications because they have interesting security policies that Co-Inflow can help enforce, and they are implemented in Co-Inflow-supported versions of Java. We evaluated precision of enforcement using the IFSpec core benchmark suite [17]. Performance evaluation was conducted on a Macbook Pro with an Intel Core i7 2.2Ghz processor and 16GB RAM, using Java version 8.

A. Human Resources Application

We ported an HR Management application[10] (HR) to Co-Inflow, which comprises approximately 4K lines of code (LoC). It supports basic human resource management, allowing users to add and search employees, change their work schedules and salaries. Users may search for employees' information, which is then displayed to the user.

In the HR case study, we want to enforce the security policy that only users that are managers can read information of employees. We use the powerset of user ids as security labels and allow flow from employee user ids to manager user ids. We insert calls to `raiseFieldLabel` on objects representing employee data to set the field label to the appropriate user id. We annotate user interface code as a sink, labeled with the user id of the currently logged-in user.

We ran two workflows on the application: (1) a manager searching and accessing employees' profiles; and (2) a (non-manager) employee searching and accessing employees' profiles. The first workflow contains only intended information flow whereas the second workflow violates the intended security policy. As expected, the Co-Inflow application permits the first workflow and intervenes to prevent showing employee details to the non-manager.

The programmatic construction of the security label lattice and the annotations to indicate the sources and sinks of information were the only changes we made to the application to port it from Java to Co-Inflow. In total, we added 20 LoC: 16 to implement the security labels; 1 to raise objects' field labels; and 3 to specify sinks. Note that the prototype allows programmers to indicate sinks using pattern matching; in this case, only the search result UI widget was marked as a sink.

B. Health Records Application

We ported HealthPlus,[11] a health records application implemented in approximately 15K LoC. Its functionality includes registration of patients, making appointments, storing patient records, pharmacy billing, and pharmacy stock controlling.

HealthPlus comes with a database of example personnel with different roles, e.g., doctors, pharmacists, and receptionists. Patients can make appointments with doctors. The application allows doctors to search for all patients' information.

We want to enforce the policy that a doctor can learn patient information only of patients assigned to that doctor. For security labels, we use the powerset of patient ids and doctor ids, with information-flow permitted from a patient to a doctor only if the doctor is assigned to the patient. Note that the information-flow relation thus depends on data that may be modified during execution.

Annotations are added to indicate the sources and sinks of information. A patient's information is labeled with her id after being loaded from the database. We also annotate user interface code as a sink, labeled with the id of the logged-in doctor. As expected, the Co-Inflow program is able to enforce the intended security policy, preventing doctors from learning information about patients they are not assigned to.

We added 18 LoC to the original system: 11 LoC to implement the security label lattice; 6 LoC to indicate sources of patient information; and 1 LoC to indicate sinks of information.

C. Roller Application

We ported Apache Roller[12] (version 2.5.x[13]), an open-source blog server. It can support thousands of users and blogs. In addition to basic blogging functionality, it features comment

[10] https://github.com/HamzaYasin1/HR-Management-System-in-Java-using-swing-framework

[11] https://github.com/heshanera/HealthPlus

[12] https://roller.apache.org/

[13] https://github.com/apache/roller/tree/roller-5.2.x

moderation, search, group blogging, and blogger-controlled layout and style. Each blog can have multiple collaborators with three permission levels: owner, editor, and drafter. It is implemented in approximately 48K LoC in Java 1.7.

We added two information-flow policies to the application. The first stops comments marked as spam from being persisted to the database. The second policy ensures that blogs with a confidential tag would not be leaked to public readers (i.e., unauthenticated users).

We build a product lattice of two information flow lattices to enforce the two policies. To track spam, we use a two-point lattice consisting of the labels *spam* and *valid*. To track confidential information, we also use a two-point lattice, but with labels *confidential* and *public*.

We add annotations to indicate the sources and sinks of information. For the first policy, a comment is labeled with the *spam* label based on Roller's spam-detecting heuristics. Database API calls are annotated as sinks that require *valid* data. For the second policy, a blog post is labeled *confidential* after being loaded from the database if it has a confidential tag. We annotate a POST response as a sink with label *confidential* or *public* based on whether the current user is authenticated. As expected, the Co-Inflow version enforces the intended security policies.

We added or modified a total of 41 LoC to enforce these policies. The lattices are implemented with 15 LoC and source/sink annotations take 14 LoC. Another 12 Loc are used to solve a precision issue: Co-Inflow protects all fields of an object with a single field label, which is efficient if all fields contain information with the same security label, but can be imprecise. Roller has an object that implements a cache for fast loading; the object also contains fields to record hit and miss statistics which are updated when both confidential and public pages are accessed, resulting in label creep for the entire cache object. We refactor the cache object to track information flow more precisely, by moving the statistics fields into their own object. This allows the cache itself to avoid label creep and prevent the false alarms caused by the statistics fields.

D. IFSpec benchmark

We used the IFSpec benchmark suite [17] to evaluate Co-Inflow's precision. IFSpec covers a broad range of different information flows found in real-world programs. It provides a core suite and extension suites. We evaluate Co-Inflow on the core suite of IFSpec, and the web vulnerability extension suite (which subsumes SecuriBench Micro[14]). The core suite contains 80 test cases covering various language features including class initializers, exceptions, reflection, aliasing, arrays, and library calls. The web vulnerability extension provides 152 test cases.

TABLE I
IFSPEC BENCHMARK RESULTS. TRUE POSITIVES, FALSE POSITIVES, TRUE NEGATIVES, AND FALSE NEGATIVES ARE REPORTED FOR CO-INFLOW WITH ONLY SOURCE AND SINK ANNOTATIONS ("ORIGINAL") AND WITH ADDITIONAL ANNOTATIONS FOR PRECISION.

	TP	FP	TN	FN
Original	141	47	41	3
Additional annotations	141	21	67	3

TABLE II
IFSPEC RESULTS FOR CORE SUITE AND WEB VULNERABILITY SUITE BY CATEGORY. NUMBER IN PARENTHESES INDICATE NUM. OF TEST CASES IN EACH CATEGORY. ONE TEST CASE CAN BE IN SEVERAL CATEGORIES.

	FP	TP	TN	FN
Core Suite (80)	24	35	18	3
Explicit flow (46)	14	18	12	2
Implicit flow (34)	10	17	6	1
Aliasing (11)	4	5	2	0
Arrays (12)	2	7	3	0
high-conditional (11)	6	3	2	0
library (7)	1	5	1	0
reflection (7)	0	1	3	3
simple (18)	7	6	5	0
exception (11)	2	6	2	1
casting (2)	1	1	0	0
class-initializer (7)	1	4	2	0
Web Vulnerability Suite (152)	23	106	23	0
Basic(47)	3	39	5	0
Collection(19)	5	14	0	0
DataStructure(8)	3	5	0	0
Factories(6)	1	3	2	0
Inter-procedual(23)	1	14	8	0
Predicate(9)	0	5	4	0
Reflection(4)	0	4	0	0
Sanitizers(6)	1	5	0	0
Session(4)	1	3	0	0
StrongUpdate(5)	2	1	2	0

We compile all test cases as Co-Inflow programs, using scripts to automatically insert sink and source annotations in accordance with the test case's specification.

The first row of Table I (labeled "Original") summarizes the results of the benchmark suite. Note that we did not add any annotations other than sources and sinks. The column marked TP (True Positives) indicates the test cases where Co-Inflow correctly detects an insecure information flow; FP (False Positive) is where Co-Inflow incorrectly reports an insecure information flow even though the test case is secure; TN (True Negative) is where Co-Inflow correctly reports no insecure information flows; and FN (False Negative) is where Co-Inflow incorrectly reports no insecure information flows even though the test case is insecure.

Table II presents detailed benchmark results by category.

Co-Inflow is correct for 182 out of the 232 test cases (=78%), with a precision[15] of 75%. The 3 false negatives are due to information flow via Java's reflection mechanism, which Co-Inflow does not track. The 47 false positives are due to imprecision in the tracking of information flows. Most are

[14] https://github.com/too4words/securibench-micro

[15] Precision is computed as $\frac{\#TP}{\#TP+\#FP}$.

caused either by coarse granularity (i.e., the default computational containers are not precise enough) or because sensitive data is stored in the field of an object that was created in a non-sensitive context. (Since we have not added any `raiseFieldLabel` annotations, the field label of objects is that the same as context label of the container that created the object.) We discuss the remaining causes of false positives below.

Co-Inflow is comparable in precision to other tools: Joana [25], a static fine-grained IFC tool for Java bytecode, has 74% precision and false negatives due to reflection.

Some of Co-Inflow's false positives can be addressed by additional annotations, such as inserting `raiseFieldLabel` to set the field label of objects correctly for the data intended to be stored in their fields or using finer-grained computational containers (by inserting `toLabeled` annotations and/or refactoring methods). We did this on all of the false positive test cases. The results are summarized in the second row of Table I (labeled "Additional annotations"). Of the 47 false positives, additional annotations converted 26 of them to true negatives, improving the precision of Co-Inflow to being correct on 208 out of the 232 test cases (=90%) with a precision of 87%.

The remaining 21 false positives can be placed into three categories: (1) computations that read sensitive data but whose results do not depend on the data due to, e.g., an arithmetic identity or casting a long value to an int; (2) arrays where elements are heterogeneously labeled (Co-Inflow uses a single label to protect all of an array's elements); (3) fields that contain sensitive data are over-written with non-sensitive data (Co-Inflow does not allow field labels to be lowered).

More details of our benchmark results are in Appendix A.

E. Performance

Our prototype implementation shows reasonable overhead for the case studies. For the HR application, we measured the performance of 4,000 searches of employee information (where each search requires loading information from the database and displaying the information on the UI). For every request, a computational container is created to load employee records to a set of objects; every object is labeled with the employee's id; and labels are checked when information is sent to a UI element. The Co-Inflow version has longer average latency (1.16ms vs. 1.01ms $=$ 15% oerhead) and 99th percentile latency (2.58ms vs. 2.07ms $=$ 25% overhead).

For the HealthPlus application we measured the performance of 4,000 searches of a patient's information by a doctor (where each search loads the patient's record from the database; stores the record in an object; and displays the record on the UI). For each request, a computational container is created for loading information; the field label of the object storing a patient's record is raised appropriately; and labels are checked when the object's contents are sent to UI elements. The Co-Inflow version has longer average latency (7.3ms vs. 6.4ms $=$ 14% overhead) and 99th percentile latency (10.9ms vs. 8.5ms $=$ 28% overhead).

For the Roller web application, we compare the performance of the Co-Inflow and Java versions using the benchmarking tool Apache JMeter.[16] Roller provides a JMeter script for stress testing that uses 5 threads and 1,000 requests which mimics real traffic by accessing various URLs (e.g., pages and feed) and introduces random delays between requests. We run the script on both the original Roller and Co-Inflow version. The Co-Inflow version has the same average throughput as the original (2.6 requests per second) but longer average latency (29ms vs. 23ms $=$ 26% overhead) and 99th percentile latency (80ms vs. 58ms $=$ 38% overhead).

Additional engineering can reduce this overhead significantly. To investigate the potential performance of a more full-fledged implementation (perhaps done as bytecode rewriting at class-load time), we hand-coded a phone-number formatting program (similar to Figure 1) in Java that explicitly represents, manipulates, and checks security labels. Unlike the Co-Inflow runtime implementation, context labels are represented as local variables in a method body, labels of method arguments are passed as additional arguments, and all object labels and field labels are represented as additional instance fields. Moreover, security labels are represented as bit-packed integers, with bit operations to implement meet and join. These modifications remove the need for a shadow stack and the map from objects to object labels and field labels and allow more efficient handling of security labels.

We tested the performance of this implementation by a loop whose body creates a new `Person` object, then calls `getPhoneNum`, `formatNum`, and then `setPhoneNum` with the newly formatted number. Method `formatNum` simply formats the input phone number by adding country code as a prefix. We compared our implementation that explicitly tracks labels with a Java version without any label tracking or checks. We run both implementations for 10^k iterations, for $k \in \{3, 6, 9, 12\}$. The performance difference between the implementations is negligible, indicating that a more full-fledged implementation could achieve very low overhead.

F. Developer effort

To port a Java application to Co-Inflow, a developer must understand the desired information flow restrictions of an application. She must then (1) implement a security label lattice that can express the intended information-flow restrictions and (2) identify and annotate the sources and sinks of sensitive information. As seen from the case studies above, the numbers of lines of code to accomplish these two tasks are relatively small. However, identifying all sources and sinks requires familiarity with the application. If a source or a sink is not correctly annotated, then Co-Inflow may fail to enforce the desired restrictions.

[16] https://jmeter.apache.org/

Assuming all sources and sinks are correctly annotated, the desired restrictions correctly encoded in the lattice, and that Co-Inflow's mechanisms are correctly tracking all relevant information flows, Co-Inflow will now soundly enforce the information-flow restrictions. That is, if no security error is raised at runtime, there is no security violation. However, Co-Inflow may be imprecise and raise a security error when there is no security violation. Code can be refactored to improve precision, such as factoring fields into their own objects (as we did in the Roller application), refactoring methods to have finer-grain containers (as we did in some IFSpec benchmarks), and proactively raising the field label of objects. To identify imprecision, the developer must exercise the application and discover situations where a security error is raised inappropriately. This iterative process relies on a reasonable test suite to exercise the application's functionality.

VIII. Related Work

Java IFC Java is a popular target language for IFC research, both source code [2], [26]–[29] and bytecode [25], [30], [31]. Jif [2] is a well-known tool for static fine-grained IFC. Programmers can annotate variables, values, and various program elements with security labels. Jif can be precise and efficient but may require significant programmer effort. By contrast, Co-Inflow causes and incurs runtime overhead, but requires relatively less annotation effort. Moreover, based on our experience, existing Java design patterns can be used in Co-Inflow without significant change, whereas porting Java programs to Jif often requires significant effort.

Laminar [28] is a dynamic IFC approach with similarities to Co-Inflow. Laminar provides a programming model to retrofit security policies onto existing programs. It introduces a lexically scoped *security region* which is protected by a secrecy label and integrity label. Programmers can create security regions to protect sensitive data. Co-Inflow by default protects every method call with a single label. Laminar relies on its OS and JVM modification to support dynamic IFC enforcement. In contrast, Co-Inflow is a programming language approach which is more flexibly deployable.

Aeolus [32] also implements coarse-grained dynamic IFC in Java, using threads as computational containers. A combination of runtime mechanisms and language restrictions limit information flow between containers. For example, the only memory that can be shared between containers must be special encapsulated objects that perform run-time label checks. These restrictions result in very low overhead label tracking, but Aeolus's programming model is a larger departure from Java's than the Co-Inflow approach.

Coarse-grained IFC Coarse-grained IFC ideas stem from security research in operating systems [33]–[35]. These ideas have also been applied to other domains, e.g., the web [7]–[9]; mobile applications [11], [12], and IoT [10]; and distributed systems [32], [36], [37]. LIO is a Haskell library that implements OS-like IFC enforcement in a language-based setting [3], [15]. Heule et al. [6] introduce a general framework for coarse-grained IFC in any programming language in which external effects can be controlled. Lu et al. [38] introduce a type system for enforcing nontransitive coarse-grained IFC policies for component-based software.

Co-Inflow is inspired by existing frameworks, particularly LIO. Our work adapts and specializes existing coarse-grained IFC work for Java. In addition, we use opaque labeled values as a practical programming mechanism, i.e., permitting labeled values where programmers do not explicitly provide a label. An early version of LIO (described by Stefan et al. [19]) also permitted the creation of labeled values without a programmer-supplied label, as does a calculus of Vassena et al. [14]. Unlike Co-Inflow, they allow a program to inspect the label of these labeled values. They ensure security by always raising the context label to the label of the inspected labeled value, whereas we prevent the code from inspecting the label of the opaque labeled value. Without also having non-opaque labels, this enforcement mechanism is unlikely to be a usable programming model due to unpredictability of context label changes, as described by Stefan et al. [19]. Indeed, this design was abandoned by LIO in favor of non-opaque labeled values.[17]

Fine- and coarse-grained IFC equivalence Recent work has shown that fine- and coarse-grained information flow control (IFC) systems are equivalent in terms of precision. Rajani et al. [13], [39] show that type systems for coarse- and fine-grained IFC are equivalent in precision and Vassena et al. [14] show a similar result for dynamic IFC mechanisms. These results encourage the adoption of coarse-grained IFC, as we can achieve the benefits (potentially less runtime overhead, better fit with existing language abstractions, more intuitive for programmers) without fear of losing precision.

IX. Conclusion

We introduce Co-Inflow, a language that demonstrates that coarse-grained dynamic IFC is a good match for imperative object-oriented programming languages such as Java. Co-Inflow aligns coarse-grained IFC mechanisms with existing Java language abstractions, allowing relatively precise tracking of information with few programmer annotations. This is accomplished in part by the use of *opaque labeled values*, which the Co-Inflow runtime system automatically and securely uses to increase precision in information-flow control. Programmers can add additional annotations to further improve precision.

Our prototype implementation of Co-Inflow, and our validation using existing Java programs and the IFSpec benchmark suite, indicate that coarse-grained dynamic IFC for Java-like language is feasible and is usable with potentially very low overhead. A Coq formalization of the core calculus CIFC and a proof of noninterference provides assurance that Co-Inflow correctly tracks information.

[17]The calculus of Vassena et al. [14] is not intended to be a practical programming model.

REFERENCES

[1] F. Pottier and V. Simonet, "Information flow inference for ML," *ACM Transactions on Programming Languages and Systems*, vol. 25, no. 1, pp. 117–158, 2003.

[2] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, "Jif: Java information flow," 2001–2016, software release. Located at http://www.cs.cornell.edu/jif.

[3] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible dynamic information flow control in Haskell," in *Proceedings of ACM Symposium on Haskell*, 2011, pp. 95–106.

[4] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "JSFlow: Tracking information flow in JavaScript and its APIs," in *Proceedings of ACM Symposium on Applied Computing*, 2014, pp. 1663–1671.

[5] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *Proceedings of ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2009, pp. 113–124.

[6] S. Heule, D. Stefan, E. Z. Yang, J. C. Mitchell, and A. Russo, "IFC inside: Retrofitting languages with dynamic information flow control," in *Proceedings of International Conference on Principles of Security and Trust*, 2015, pp. 11–31.

[7] P. Buiras, D. Vytiniotis, and A. Russo, "HLIO: Mixing static and dynamic typing for information-flow control in Haskell," in *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, 2015, pp. 289–301.

[8] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo, "Hails: Protecting data privacy in untrusted web applications," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 2012, pp. 47–60.

[9] D. Stefan, E. Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, and D. Mazières, "Protecting users by confining JavaScript with COWL," in *Proceedings of USENIX Conference on Operating Systems Design and Implementation*, 2014, pp. 131–146.

[10] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash, "FlowFence: Practical data protection for emerging IoT application frameworks," in *Proceedings of USENIX Security Symposium*, 2016, pp. 531–548.

[11] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake, "Run-time enforcement of information-flow properties on Android," in *Proceedings of European Symposium on Research in Computer Security*, 2013, pp. 775–792.

[12] A. Nadkarni, B. Andow, W. Enck, and S. Jha, "Practical DIFC enforcement on Android," in *Proceedings of USENIX Security Symposium*, 2016, pp. 1119–1136.

[13] V. Rajani and D. Garg, "Types for information flow control: Labeling granularity and semantic models," in *Proceedings of IEEE Symposium on Computer Security Foundations*, 2018, pp. 233–246.

[14] M. Vassena, A. Russo, D. Garg, V. Rajani, and D. Stefan, "From fine- to coarse-grained dynamic information flow control and back," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 76:1–76:31, Jan. 2019.

[15] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Maziéres, "Addressing covert termination and timing channels in concurrent information flow systems," in *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, 2012, pp. 201–214.

[16] P. Buiras, D. Stefan, and A. Russo, "On dynamic flow-sensitive floating-label systems," in *Proceedings of IEEE Computer Security Foundations Symposium*, 2014, pp. 65–79.

[17] T. Hamann, M. Herda, H. Mantel, M. Mohr, D. Schneider, and M. Tasch, "A uniform information-flow security benchmark suite for source code and bytecode," in *Proceedings of Nordic Conference on Secure IT Systems*, 2018, pp. 437–453.

[18] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[19] D. Stefan, D. Mazières, J. C. Mitchell, and A. Russo, "Flexible dynamic information flow control in the presence of exceptions," *Journal of Functional Programming*, vol. 27, 2017.

[20] G. M. Bierman, M. Parkinson, and A. Pitts, "MJ: An imperative core calculus for Java and Java with effects," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-563, April 2003.

[21] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proceedings of IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[22] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett, "All your IFCException are belong to us," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. Los Alamitos, CA, USA: IEEE Computer Society, 2013, pp. 3–17.

[23] P. Shroff, S. F. Smith, and M. Thober, "Dynamic dependency monitoring to secure information flow," in *Proceedings of the 20th IEEE Computer Security Foundations Symposium*. IEEE Computer Society, 2007, pp. 203–217.

[24] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A library for implementing analyses and transformations of Java source code," *Software: Practice and Experience*, vol. 46, no. 9, pp. 1155–1179, 2016.

[25] J. Graf, M. Hecker, and M. Mohr, "Using JOANA for information flow control in Java programs - a practical guide," in *Proceedings of Working Conference on Programming Languages*, 2013, pp. 123–138.

[26] A. Banerjee and D. A. Naumann, "Stack-based access control and secure information flow," *Journal of functional programming*, vol. 15, no. 2, pp. 131–177, 2005.

[27] T. Amtoft, S. Bandhakavi, and A. Banerjee, "A logic for information flow in object-oriented programs," in *Proceedings of ACM Symposium on Principles of Programming Languages*, 2006, pp. 91–102.

[28] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel, "Laminar: Practical fine-grained decentralized information flow control," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 63–74.

[29] R. Küsters, T. Truderung, B. Beckert, D. Bruns, M. Kirsten, and M. Mohr, "A hybrid approach for proving noninterference of Java programs," in *Proceedings of IEEE Computer Security Foundations Symposium*, 2015, pp. 305–319.

[30] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *International Journal of Information Security*, vol. 8, no. 6, pp. 399–422, 2009.

[31] G. Barthe, D. Pichardie, and T. Rezk, "A certified lightweight non-interference Java bytecode verifier," in *Proceedings of European Symposium on Programming*, 2007, pp. 125–140.

[32] W. Cheng, D. R. Ports, D. Schultz, V. Popic, A. Blankstein, J. Cowling, D. Curtis, L. Shrira, and B. Liskov, "Abstractions for usable information flow control in Aeolus," in *Proceedings of USENIX Annual Technical Conference*, 2012, pp. 139–151.

[33] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris, "Labels and event processes in the Asbestos operating system," in *Proceedings of ACM Symposium on Operating Systems Principles*, 2005, pp. 17–30.

[34] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard OS abstractions," in *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles*, 2007, pp. 321–334.

TABLE III
CO-INFLOW ANNOTATIONS THAT ADDRESS FALSE POSITIVE OF IFSPEC BENCHMARKING RESULTS

Annotation	**Test Cases (26)**
Inserting `raiseFieldLabel`	Aliasing-InterProcedural-secure Aliasing-Simple-secure ScenarioBanking-Secure Webstore3 Static-Initializers-HighAccess-secure SecuriBench-Aliasing3 SecuriBench-Arrays3-secure SecuriBench-Basic17-secures SecuriBench-Basic29-secure SecuriBench-Collections10-secure SecuriBench-Collections2-secure SecuriBench-Datastructures2-secure SecuriBench-Inter12-secure
Inserting `toLabeled` and/or refactoring methods	Deepalias2 Webstore HighConditionalIncrementalLeak-secure IFLoop ScenarioPasswordSecure ImplicitListSizeNoLeak IFMethodContract2 ObjectSensLeak ReviewerAnonymity-NoLeak ExceptionalControlFlow1-secure SecuriBench-Basic30-secure SecuriBench-Datastructures1-secure SecuriBench-Factories3-secure

TABLE IV
FALSE POSITIVE OF IFSPEC BENCHMARKING RESULTS THAT CANNOT BE ADDRESSED BY CO-INFLOW ANNOTATIONS

Causes of FP	**Test Cases (21)**
A method reads sensitive data, but results do not depend on the data	Aliasing-ControlFlow-secure simpleConditionalAssignmentEqual simpleErasureByConditionalChecks BooleanOperations-secure IFMethodContract Polynomial ExceptionalControlFlow2-secure LostInCast
Heterogeneously label	ArrayIndexSensitivity-secure SecuriBench-Arrays10-secure SecuriBench-Arrays2-secure SecuriBench-Arrays5 SecuriBench-Arrays8-secure SecuriBench-Collections13-secure SecuriBench-Collections6-secure SecuriBench-Collections7-secure SecuriBench-Datastructures4 SecuriBench-Session2-secure
Sensitive data overriden by non-sensitive data	SecuriBench-Sanitizers3 SecuriBench-StrongUpdates3
Thread-local data.	SecuriBench-StrongUpdates5

[35] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in HiStar," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 2006, pp. 263–278.

[36] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières, "Securing distributed systems with information flow control," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA: USENIX Association, 2008, pp. 293–308.

[37] M. V. Pedersen and S. Chong, "Programming with flow-limited authorization: Coarser is better," in *4th IEEE European Symposium on Security and Privacy*. Piscataway, NJ, USA: IEEE Press, Jun. 2019.

[38] Y. Lu and C. Zhang, "Nontransitive security types for coarse-grained information flow control," in *Proceedings of 33rd IEEE Symposium on Computer Security Foundations*, 2020, pp. 199–213.

[39] V. Rajani, I. Bastys, W. Rafnsson, and D. Garg, "Type systems for information flow control: the question of granularity," *ACM SIGLOG News*, vol. 4, no. 1, pp. 6–21, 2017.

APPENDIX A
IFSPEC BENCHMARKING RESULTS

Detailed results are shown in Tables III and IV.

APPENDIX B
WELL-FORMEDNESS OF CONFIGURATIONS

Judgment $CT \vdash \mathcal{H} : ok$ ensures the heap is well-formed, i.e., (a) objects in a heap are correctly addressed: they are pointing to objects whose class definitions are valid; (b) class names mentioned in the heap are in the class table; and (c) all fields in every heap object are valid: the value of a field is either a valid object reference or null.

Judgment $CT, \mathcal{H} \vdash \theta : ok$ ensures all variables map to valid values; $\vdash e : ok$, $\vdash \kappa : ok$, and $\vdash \rho : ok$ respectively ensure expressions, continuations, and continuation stacks have correct syntax; $CT, \mathcal{H} \vdash (e, \rho, pc, \theta) : ok$ ensures that every component of the container is well-formed; $CT \vdash \langle \Delta, \mathcal{H} \rangle : ok$ ensures that every container in stack Δ is well-formed.

APPENDIX C
NON-INTERFERENCE PROOF

Definitions of low equivalence are shown in Figure 10.

To prove TINI, we need a few lemmas proving low equivalence is preserved under different combinations of reduction behaviours. Lemmas 1, 2, 3, 4 shows these combinations.

LEMMA 1: *L-L* PRESERVES $\approx_L$. If $\Sigma_1.pc \sqsubseteq L$, $\Sigma_2.pc \sqsubseteq L$, $CT \vdash \Sigma_1 \rightarrow \Sigma_1'$, $CT \vdash \Sigma_2 \rightarrow \Sigma_2'$, and $\Sigma_1 \approx_L^{\phi} \Sigma_2$, then exists ϕ', $\Sigma_1' \approx_L^{\phi'} \Sigma_2'$.

LEMMA 2: *H-H2H* PRESERVES $\approx_L$. If $\Sigma_1.pc \not\sqsubseteq L$, $\Sigma_2.pc \not\sqsubseteq L$, $CT \vdash \Sigma_1 \rightarrow \Sigma_1'$, $\Sigma_1'.pc \not\sqsubseteq L$, and $\Sigma_1 \approx_L^{\phi} \Sigma_2$, then exists ϕ', $\Sigma_1' \approx_L^{\phi'} \Sigma_2$.

LEMMA 3: *H-H2H* PRESERVES $\approx_L$. If $\Sigma_1.pc \not\sqsubseteq L$, $\Sigma_2.pc \not\sqsubseteq L$, $CT \vdash \Sigma_2 \rightarrow \Sigma_2'$, $\Sigma_2'.pc \not\sqsubseteq L$, and $\Sigma_1 \approx_L^{\phi} \Sigma_2$, then exists ϕ', $\Sigma_1 \approx_L^{\phi'} \Sigma_2'$.

LEMMA 4: *H2L-H2L* PRESERVES $\approx_L$. If $\Sigma_1.pc \not\sqsubseteq L$, $\Sigma_2.pc \not\sqsubseteq L$, $CT \vdash \Sigma_1 \rightarrow \Sigma_1'$, $CT \vdash \Sigma_2 \rightarrow \Sigma_2'$, $\Sigma_1'.pc \sqsubseteq L$, $\Sigma_2'.pc \sqsubseteq L$, and $\Sigma_1 \approx_L^{\phi} \Sigma_2$, then exists ϕ' so that $\Sigma_1' \approx_L^{\phi'} \Sigma_2'$.

PROOF. All four lemmas can be proven using case analysis on the reduction rules $\rightarrow$.

LEMMA 5: *LT-LT*. If Σ_1 is terminated, $\Sigma_1.pc \sqsubseteq L$, and $\Sigma_1 \approx_L^{\phi} \Sigma_2$, then Σ_2 is terminated.

PROOF. By definition of low equivalence of configurations.

$$\textsc{Addresses-H} \approx_L \quad \frac{\mathcal{H}_1(\varsigma_1) = \langle cn, \mathbb{F}_1, \ell_f, \ell_o\rangle \quad \mathcal{H}_2(\varsigma_2) = \langle cn, \mathbb{F}_2, \ell_f, \ell_o\rangle \quad \ell_o \sqsubseteq L \quad \ell_f \not\sqsubseteq L}{\phi[\varsigma_1, \mathcal{H}_1 \approx_L^\phi \varsigma_2, \mathcal{H}_2]}$$

$$\textsc{Addresses-L} \approx_L \quad \frac{\begin{array}{c}\mathcal{H}_1(\varsigma_1) = \langle cn, \mathbb{F}_1, \ell_f, \ell_o\rangle \quad \mathcal{H}_2(\varsigma_2) = \langle cn, \mathbb{F}_2, \ell_f, \ell_o\rangle \quad \ell_o \sqsubseteq L \quad \ell_f \sqsubseteq L \\ \forall f.\ (\mathbb{F}_1(f) = \texttt{null} \iff \mathbb{F}_2(f) = \texttt{null}) \quad \vee \quad (\mathbb{F}_1(f) = \varsigma_{f_1} \wedge\ \mathbb{F}_2(f) = \varsigma_{f_2} \\ \wedge \mathcal{H}_1(\varsigma_{f_1}) = \langle cn_{f1}, \mathbb{F}_{f_1}, \ell_{f_{f_1}}, \ell_{o_{f1}}\rangle \quad \wedge \mathcal{H}_2(\varsigma_{f_2}) = \langle cn_{f2}, \mathbb{F}_{f_2}, \ell_{f_{f_2}}, \ell_{o_{f2}}\rangle \\ \wedge (\phi(\varsigma_{f_1}) = \varsigma_{f_2} \wedge \ell_{o_{f1}} \sqsubseteq L \wedge \ell_{o_{f2}} \sqsubseteq L \quad \wedge \ell_{o_{f1}} = \ell_{o_{f2}} \wedge cn_{f1} = cn_{f2}) \\ \vee \quad (\ell_{o_{f1}} \not\sqsubseteq L \wedge \ell_{o_{f2}} \not\sqsubseteq L) \quad)\end{array}}{\phi[\varsigma_1, \mathcal{H}_1 \approx_L^\phi \varsigma_2, \mathcal{H}_2]}$$

$$\textsc{Heap} \approx_L \quad \frac{\begin{array}{c}\forall \varsigma_1\ \varsigma_2.[\phi(\varsigma_1) = \varsigma_2 \implies \phi[\varsigma_1, \mathcal{H}_1 \approx_L^\phi \varsigma_2, \mathcal{H}_2]] \\ \wedge \quad \forall \varsigma, \varsigma \notin dom(\mathcal{H}_1) \implies \varsigma \notin dom(\phi) \quad \wedge \quad \forall \varsigma, \varsigma \notin dom(\mathcal{H}_2) \implies \varsigma \notin dom(\phi^{-1}) \\ \wedge \quad \forall \varsigma. \mathcal{H}_1(\varsigma) = \langle cn, \mathbb{F}, \ell_f, \ell_o\rangle \wedge \ell_o \not\sqsubseteq L \implies \varsigma \notin dom(\phi) \quad \wedge \quad \forall \varsigma. \mathcal{H}_2(\varsigma) = \langle cn, \mathbb{F}, \ell_f, \ell_o\rangle \wedge \ell_o \not\sqsubseteq L \implies \varsigma \notin dom(\phi^{-1})\end{array}}{\mathcal{H}_1 \approx_L^\phi \mathcal{H}_2}$$

$$\textsc{Expr}\approx_L\textsc{-object-L} \quad \frac{\mathcal{H}_1(\varsigma_1) = \langle cn, \mathbb{F}_1, \ell_f, \ell_o\rangle \quad \mathcal{H}_2(\varsigma_2) = \langle cn, \mathbb{F}_2, \ell_f, \ell_o\rangle \quad \ell_o \sqsubseteq L \quad \phi(\varsigma_1) = \varsigma_2}{\langle \varsigma_1, \mathcal{H}_1\rangle \approx_L^\phi \langle \varsigma_2, \mathcal{H}_2\rangle}$$

$$\textsc{Expr}\approx_L\textsc{-object-H} \quad \frac{\mathcal{H}_1(\varsigma_1) = \langle cn, \mathbb{F}_1, \ell_{f_1}, \ell_{o1}\rangle \quad \mathcal{H}_2(\varsigma_2) = \langle cn, \mathbb{F}_2, \ell_{f_2}, \ell_{o2}\rangle \quad \ell_{o1} \not\sqsubseteq L \quad \ell_{o2} \not\sqsubseteq L}{\langle \varsigma_1, \mathcal{H}_1\rangle \approx_L^\phi \langle \varsigma_2, \mathcal{H}_2\rangle}$$

$$\textsc{Expr}\approx_L\textsc{-labeled-L} \quad \frac{\langle e_1, \mathcal{H}_1\rangle \approx_L^\phi \langle e_2, \mathcal{H}_2\rangle \quad \ell \sqsubseteq L}{\langle e_1^\ell, \mathcal{H}_1\rangle \approx_L^\phi \langle e_2^\ell, \mathcal{H}_2\rangle}$$

$$\textsc{Expr}\approx_L\textsc{-labeled-H} \quad \frac{\ell \not\sqsubseteq L}{\langle e_1^\ell, \mathcal{H}_1\rangle \approx_L^\phi \langle e_2^\ell, \mathcal{H}_2\rangle}$$

$$\textsc{Expr}\approx_L\textsc{-OpaqueLabeled-L} \quad \frac{\langle e_1, \mathcal{H}_1\rangle \approx_L^\phi \langle e_2, \mathcal{H}_2\rangle \quad \ell \sqsubseteq L}{\langle e_1^{[\ell]}, \mathcal{H}_1\rangle \approx_L^\phi \langle e_2^{[\ell]}, \mathcal{H}_2\rangle}$$

$$\textsc{Expr}\approx_L\textsc{-OpaqueLabeled-H} \quad \frac{\ell_1 \not\sqsubseteq L \quad \ell_2 \not\sqsubseteq L}{\langle e_1^{[\ell_1]}, \mathcal{H}_1\rangle \approx_L^\phi \langle e_2^{[\ell_2]}, \mathcal{H}_2\rangle}$$

$$\textsc{Variable State} \approx_L \quad \frac{\forall x. \theta_1(x) = v_1 \wedge \theta_2(x) = v_2 \wedge \langle v_1, \mathcal{H}_1\rangle \approx_L^\phi \langle v_1, \mathcal{H}_2\rangle}{\langle \theta_1, \mathcal{H}_1\rangle \approx_L^\phi \langle \theta_2, \mathcal{H}_2\rangle}$$

$$\textsc{Expression Stack} \approx_L \quad \frac{\langle e_1, \mathcal{H}_1\rangle \approx_L^\phi \langle e_2, \mathcal{H}_2\rangle \quad \langle \rho_1, \mathcal{H}_1\rangle \approx_L^\phi \langle \rho_2, \mathcal{H}_2\rangle}{\langle e_1 :: \rho_1, \mathcal{H}_1\rangle \approx_L^\phi \langle e_2 :: \rho_2, \mathcal{H}_2\rangle}$$

$$\textsc{Container} \approx_L \quad \frac{pc \sqsubseteq L \quad \langle e_1, \mathcal{H}_1\rangle \approx_L^\phi \langle e_2, \mathcal{H}_2\rangle \quad \langle \rho_1, \mathcal{H}_1\rangle \approx_L^\phi \langle \rho_2, \mathcal{H}_2\rangle \quad \langle \theta_1, \mathcal{H}_1\rangle \approx_L^\phi \langle \theta_2, \mathcal{H}_2\rangle}{\langle (e_1, \rho_1, pc, \theta_1), \mathcal{H}_1\rangle \approx_L^\phi \langle (e_2, \rho_2, pc, \theta_2), \mathcal{H}_2\rangle}$$

$$\textsc{Configuration-Empty} \quad \frac{\mathcal{H}_1 \approx_L^\phi \mathcal{H}_2}{\langle [], \mathcal{H}_1\rangle \approx_L^\phi \langle [], \mathcal{H}_2\rangle}$$

$$\textsc{Configuration-L} \quad \frac{\delta_1.pc \sqsubseteq L \quad \delta_2.pc \sqsubseteq L \quad \mathcal{H}_1 \approx_L^\phi \mathcal{H}_2 \quad \langle \delta_1, \mathcal{H}_1\rangle \approx_L^\phi \langle \delta_2, \mathcal{H}_2\rangle \quad \langle \Delta_1, \mathcal{H}_1\rangle \approx_L^\phi \langle \Delta_2, \mathcal{H}_2\rangle}{\langle \delta_1 :: \Delta_1, \mathcal{H}_1\rangle \approx_L^\phi \langle \delta_2 :: \Delta_2, \mathcal{H}_2\rangle}$$

$$\textsc{Configuration-H} \quad \frac{\Delta_1.pc \sqsubseteq L \quad \Delta_2.pc \sqsubseteq L \quad \forall i(1 \le i \le m), \delta_i.pc \not\sqsubseteq L \quad \forall j(1 \le j \le n), \delta'_j.pc \not\sqsubseteq L \quad \langle \Delta_1, \mathcal{H}_1\rangle \approx_L^\phi \langle \Delta_2, \mathcal{H}_2\rangle}{\langle \delta_1 \ldots \delta_m :: \Delta_1, \mathcal{H}_1\rangle \approx_L^\phi \langle \delta'_1 \ldots \delta'_n :: \Delta_2, \mathcal{H}_2\rangle}$$

Fig. 10. Definitions of low equivalence of expressions, continuations, containers, and configurations

LEMMA 6: *HT-H2H*. If Σ_1 is terminated, $\Sigma_1.pc \not\sqsubseteq L$, $\Sigma_2.pc \not\sqsubseteq L$, $\Sigma_1 \approx_L^\phi \Sigma_2$ and $CT \vdash \Sigma_2 \to \Sigma_2'$, then $\Sigma_2'.pc \not\sqsubseteq L$.

PROOF. By case analysis on the reduction rules $\to$ and the definition of low equivalence.

Now we prove TINI.

PROOF. By strong induction on the number of *total steps* that two executions take to reach final configurations, and applying lemmas 1, 2, 3, 4, 5, and 6.

APPENDIX D
TYPE SYSTEM

The judgement $\vdash CT : ok$ ensures that all loaded classes are well-typed, i.e., all method declarations in a class are well typed: it's body's type matches its declared return type.

Figure 11 shows remaining typing judgements for expressions. Some of these rules use auxiliary functions defined in Definition 1. Figure 12 shows typing judgements of continuations. Figure 13 shows typing judgements of a container, container stack, and configuration.

DEFINITION 1: AUXILIARY FUNCTIONS.

$$lookup\_md(CT, cn, m) \stackrel{def}{=} \tau_r\ m(\tau\ x)\{e\}$$
where $CT(cn) = \textbf{class}\ cn\ \overline{fd}\ \overline{md}$ and $\tau_r\ m(\tau\ x)\{e\} \in \overline{md}$

$$lookup\_fd(CT, cn, f) \stackrel{def}{=} cn'\ f$$
where $CT(cn) = \textbf{class}\ cn\ \overline{fd}\ \overline{md}$ and $cn'\ f \in \overline{fd}$

$$fields(CT, cn) \stackrel{def}{=} \{f \mid cn'\ f \in \overline{fd}\}$$
where $CT(cn) = \textbf{class}\ cn\ \overline{fd}\ \overline{md}$

APPENDIX E
OPERATIONAL SEMANTICS

Remaining reduction rules are presented in Figure 14.

COMPARE
$$\frac{CT, \Gamma, \mathcal{H} \vdash e_1 : cn \quad CT, \Gamma, \mathcal{H} \vdash e_2 : cn}{CT, \Gamma, \mathcal{H} \vdash e_1 == e_2 : \texttt{bool}}$$

FIELDREAD
$$\frac{CT, \Gamma, \mathcal{H} \vdash e : cn \quad \mathit{lookup\_fd}(CT, cn, f) = cn'\ f;}{CT, \Gamma, \mathcal{H} \vdash e.f : cn'}$$

METHODCALL
$$\frac{CT, \Gamma, \mathcal{H} \vdash e : cn \quad CT, \Gamma, \mathcal{H} \vdash e_1 : \tau \quad \mathit{lookup\_md}(CT, cn, m) = \tau'\ m(\tau\ a)\{e_{body}\}}{CT, \Gamma, \mathcal{H} \vdash e.m(e_1) : \tau'}$$

NEWEXP
$$\frac{CT(cn) = \textbf{class}\ cn\ \overline{fd}\ \overline{md}}{CT, \Gamma, \mathcal{H} \vdash \texttt{new}\ cn() : cn}$$

NULL
$$\frac{}{CT, \Gamma, \mathcal{H} \vdash \texttt{null} : cn}$$

BOOLEAN
$$\frac{}{CT, \Gamma, \mathcal{H} \vdash \texttt{true/false} : \texttt{bool}}$$

VAR
$$\frac{\Gamma(x) = \tau}{CT, \Gamma, \mathcal{H} \vdash x : \tau}$$

ASSIGNMENT
$$\frac{\Gamma(x) = \tau \quad CT, \Gamma, \mathcal{H} \vdash e : \tau}{CT, \Gamma, \mathcal{H} \vdash x = e : \tau}$$

IF
$$\frac{CT, \Gamma, \mathcal{H} \vdash e : \texttt{bool} \quad CT, \Gamma, \mathcal{H} \vdash e_1 : \tau \quad CT, \Gamma, \mathcal{H} \vdash e_2 : \tau}{CT, \Gamma, \mathcal{H} \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

SEQUENCE
$$\frac{CT, \Gamma, \mathcal{H} \vdash e_1 : \tau \quad CT, \Gamma, \mathcal{H} \vdash e_2 : \tau'}{CT, \Gamma, \mathcal{H} \vdash e_1\ ;\ e_2 : \tau'}$$

OBJID
$$\frac{\mathcal{H}(\varsigma) = \langle cn, \mathbb{F}, \ell_f, \ell_o \rangle \quad CT(cn) = \textbf{class}\ cn\ \overline{fd}\ \overline{md}}{CT, \Gamma, \mathcal{H} \vdash \varsigma : cn}$$

LABEL
$$\frac{}{CT, \Gamma, \mathcal{H} \vdash \ell : \texttt{Label}}$$

CONTEXTLABEL
$$\frac{}{CT, \Gamma, \mathcal{H} \vdash \texttt{getContextLabel}() : \texttt{Label}}$$

LABELDATA
$$\frac{CT, \Gamma, \mathcal{H} \vdash e : \tau \quad CT, \Gamma, \mathcal{H} \vdash \ell : \texttt{Label}}{CT, \Gamma, \mathcal{H} \vdash \texttt{labelData}(e, \ell) : \texttt{Labeled}\ \tau}$$

Fig. 11. Typing rules for expressions

COMPARE.LEFT.HOLE
$$\frac{CT, \Gamma, \mathcal{H} \vdash e : cn}{CT, \Gamma, \mathcal{H} \vdash \bullet == e : cn \to \texttt{bool}}$$

COMPARE.RIGHT.HOLE
$$\frac{CT, \Gamma, \mathcal{H} \vdash e : cn}{CT, \Gamma, \mathcal{H} \vdash e == \bullet : cn \to \texttt{bool}}$$

FIELDREAD.OBJ.HOLE
$$\frac{\mathit{lookup\_fd}(CT, cn, f) = cn'\ f}{CT, \Gamma, \mathcal{H} \vdash \bullet.f : cn \to \tau}$$

METHODCALL.OBJ.HOLE
$$\frac{CT, \Gamma, \mathcal{H} \vdash x : \tau \quad \mathit{lookup\_md}(CT, cn, m) = \tau'\ m(\tau\ a)\{e_{body}\}}{CT, \Gamma, \mathcal{H} \vdash \bullet.m(x) : cn \to \tau'}$$

LABELDATA.HOLE
$$\frac{}{CT, \Gamma, \mathcal{H} \vdash \texttt{labelData}(\bullet, \ell) : \tau \to \texttt{Labeled}\ \tau}$$

METHODCALL.ARG.HOLE
$$\frac{CT, \Gamma, \mathcal{H} \vdash e : cn \quad \mathit{lookup\_md}(CT, cn, m) = \tau'\ m(\tau\ a)\{e_{body}\}}{CT, \Gamma, \mathcal{H} \vdash e.m(\bullet) : \tau \to \tau'}$$

UNLABEL.HOLE
$$\frac{}{CT, \Gamma, \mathcal{H} \vdash \texttt{unlabel}(\bullet) : \texttt{Labeled}\ \tau \to \tau}$$

LABELOF.HOLE
$$\frac{}{CT, \Gamma, \mathcal{H} \vdash \texttt{labelOf}(\bullet) : \texttt{Labeled}\ \tau \to \texttt{Label}}$$

FIELDLABELOF.HOLE
$$\frac{}{CT, \Gamma, \mathcal{H} \vdash \texttt{fieldLabelOf}(\bullet) : cn \to \texttt{Label}}$$

ASSIGNMENT.HOLE
$$\frac{\Gamma(x) = \tau}{CT, \Gamma, \mathcal{H} \vdash x = \bullet : \tau \to \tau}$$

SEQUENCE.HOLE
$$\frac{CT, \Gamma, \mathcal{H} \vdash e : \tau}{CT, \Gamma, \mathcal{H} \vdash \bullet\ ;\ e : \tau' \to \tau}$$

FIELDWRITE.OBJ.HOLE
$$\frac{\mathit{lookup\_fd}(CT, cn, f) = cn'\ f \quad CT, \Gamma, \mathcal{H} \vdash e : cn'}{CT, \Gamma, \mathcal{H} \vdash \bullet.f = e : cn \to cn'}$$

FIELDWRITE.VAL.HOLE
$$\frac{\mathit{lookup\_fd}(CT, cn, f) = cn'\ f \quad CT, \Gamma, \mathcal{H} \vdash e : cn}{CT, \Gamma, \mathcal{H} \vdash e.f = \bullet : cn' \to cn'}$$

IF.HOLE
$$\frac{CT, \Gamma, \mathcal{H} \vdash e_1 : \tau \quad CT, \Gamma, \mathcal{H} \vdash e_2 : \tau}{CT, \Gamma, \mathcal{H} \vdash \texttt{if}\ \bullet\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \texttt{bool} \to \tau}$$

Fig. 12. Typing rules for continuations

VARIABLE STATE
$$\frac{\forall x\ \tau, \Gamma(x) = \tau \quad \theta(x) = v \quad CT, \Gamma, \mathcal{H} \vdash v : \tau}{CT, \Gamma, \mathcal{H} \vdash \theta : ok}$$

CONTINUATION-STACK
$$\frac{CT, \Gamma, \mathcal{H} \vdash \kappa : \tau \to \tau' \quad CT, \Gamma, \mathcal{H} \vdash \rho : \tau' \to \tau''}{CT, \Gamma, \mathcal{H} \vdash (\kappa :: \rho) : \tau \to \tau''}$$

CONTAINER
$$\frac{e \text{ is free of assignment} \quad CT, \Gamma, \mathcal{H} \vdash e : \tau \quad CT, \Gamma, \mathcal{H} \vdash \theta : ok \quad CT, \Gamma, \mathcal{H} \vdash \rho : \tau \to \tau'}{CT, \Gamma, \mathcal{H} \vdash (e, \rho, \ell, \theta) : \tau \to \tau'}$$

STACK-NIL
$$\frac{}{CT, \Gamma, \mathcal{H} \vdash [] : \tau \to \tau}$$

CONTAINER-STACK
$$\frac{CT, \Gamma, \mathcal{H} \vdash \delta : \tau \to \tau' \quad CT, \Gamma, \mathcal{H} \vdash \Delta : \tau' \to \tau''}{CT, \Gamma, \mathcal{H} \vdash \delta :: \Delta : \tau \to \tau''}$$

CONFIGURATION
$$\frac{CT, \Gamma, \mathcal{H} \vdash \delta : \tau \to \tau' \quad CT, \Gamma, \mathcal{H} \vdash \Delta : \tau' \to \tau'' \quad \vdash CT : ok}{CT, \Gamma \vdash \langle \mathcal{H}, \delta :: \Delta \rangle : \tau \to \tau''}$$

Fig. 13. Typing rules for continuation stack, container, container stack, and configuration

E-VAR
$$\frac{v = \theta(x) \quad \delta = (x, \rho, pc, \theta) \qquad \delta' = (v, \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-CTX-RETURN
$$\frac{\delta = (v, \kappa :: \rho, pc, \theta) \quad \delta' = (\kappa[v], \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-COMP-CTX1
$$\frac{\delta = (e_1 == e_2, \rho, pc, \theta) \quad \delta' = (e_1, \bullet == e_2 :: \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-COMP-CTX2
$$\frac{\delta = (v == e_2, \rho, pc, \theta) \quad \delta' = (e_2, v == \bullet :: \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-FIELDREAD-CTX
$$\frac{\delta = (e.f, \rho, pc, \theta) \quad \delta' = (e, \bullet.f :: \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-FIELDREAD-EXCEPTION
$$\frac{\delta = (v.f, \rho, pc, \theta) \quad (\texttt{null}, \ell) = openOpaque(v)}{CT \vdash \langle \delta :: \Delta; \mathcal{H} \rangle \rightarrow Exception}$$

E-METHODCALL-CTX1
$$\frac{\delta = (e_1.m(e_2), \rho, pc, \theta) \quad \delta' = (e_1, \bullet.m(e_2) :: \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-METHODCALL-CTX2
$$\frac{\delta = (v.m(e_2), \rho, pc, \theta) \quad \delta' = (e_2, v.m(\bullet) :: \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-METHODCALL-EXCEPTION
$$\frac{\delta = (v.m(v'), \rho, pc, \theta) \quad (\texttt{null}, \ell) = openOpaque(v)}{CT \vdash \langle \delta :: \Delta; \mathcal{H} \rangle \rightarrow Exception}$$

E-LABELDATA-CTX-VAL
$$\frac{\delta = (\texttt{labelData}(e, \ell), \rho, pc, \theta) \quad \delta' = (e, \texttt{labelData}(\bullet, \ell) :: \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-LABELDATA-EXCEPTION
$$\frac{\delta = (\texttt{labelData}(v, \ell_v), \rho, pc, \theta) \qquad pc \not\sqsubseteq \ell_v}{CT \vdash \langle \delta :: \Delta; \mathcal{H} \rangle \rightarrow Exception}$$

E-UNLABEL-CTX
$$\frac{\delta = (\texttt{unlabel}(e), \rho, pc, \theta) \quad \delta' = (e, \texttt{unlabel}(\bullet) :: \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-LABELOF-CTX
$$\frac{\delta = (\texttt{labelOf}(e), \rho, pc, \theta) \quad \delta' = (e, \texttt{labelOf}(\bullet) :: \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-RAISEFIELDLABEL-CTX-OBJ
$$\frac{\delta = (\texttt{raiseFieldLabel}(e, e'), \rho, pc, \theta) \quad \delta' = (e, \texttt{raiseFieldLabel}(\bullet, e') :: \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-RAISEFIELDLABEL-CTX-LABEL
$$\frac{\delta = (\texttt{raiseFieldLabel}(v, e'), \rho, pc, \theta) \quad \delta' = (e', \texttt{raiseFieldLabel}(v, \bullet) :: \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-RAISEFIELDLABEL-EXCEPTION
$$\frac{\delta = (\texttt{raiseFieldLabel}(v, v'), \rho, pc, \theta) \quad (\texttt{null}, \ell') = openOpaque(v)}{CT \vdash \langle \delta :: \Delta; \mathcal{H} \rangle \rightarrow Exception}$$

E-RAISEFIELDLABEL-INFORMATIONLEAK
$$\frac{\begin{array}{c}\delta = (\texttt{raiseFieldLabel}(v_o, v_\ell), \rho, pc, \theta) \\ (\varsigma, \ell) = openOpaque(v_o) \\ (\ell_f{}', \ell') = openOpaque(v_\ell) \\ \mathcal{H}(\varsigma) = \langle cn, \mathbb{F}, \ell_f, \ell_o \rangle \\ pc \sqcup \ell \sqcup \ell' \not\sqsubseteq \ell_f \vee \ell_f \sqsubseteq \ell_f{}'\end{array}}{CT \vdash \langle \delta :: \Delta; \mathcal{H} \rangle \rightarrow Exception}$$

E-FIELDWRITE-INFORMATIONLEAK
$$\frac{\begin{array}{c}\delta = (v_o.f = v, \rho, pc, \theta) \\ (\varsigma, \ell_r) = openOpaque(v_o) \\ (\varsigma', \ell_r') = openOpaque(v) \\ \mathcal{H}(\varsigma) = \langle cn, \mathbb{F}, \ell_f, \ell_o \rangle \\ (pc \sqcup \ell_r \sqcup \ell_r') \not\sqsubseteq \ell_f\end{array}}{CT \vdash \langle \delta :: \Delta; \mathcal{H} \rangle \rightarrow Exception}$$

E-TOLABELED-CTX
$$\frac{\begin{array}{c}\delta = (\texttt{toLabeled}(e, e_1), \rho, pc, \theta) \\ e \text{ contains no assignments} \\ \delta = (e_1, \texttt{toLabeled}(e, \bullet) :: \rho, pc, \theta)\end{array}}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-ASSIGNMENT-CTX
$$\frac{\delta = (x = e, \rho, pc, \theta) \quad \delta' = (e, x = \bullet :: \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-VARASSIGN
$$\frac{\delta = (x = v, \rho, pc, \theta) \quad \delta' = (v, \rho, pc, \theta[x \mapsto v])}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-FIELDWRITE-CTX1
$$\frac{\delta = (e_1.f = e_2, \rho, pc, \theta) \quad \delta' = (e_1, \bullet.f = e_2 :: \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-FIELDWRITE-CTX2
$$\frac{\delta = (v.f = e_2, \rho, pc, \theta) \quad \delta' = (e_2, v.f = \bullet :: \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-FIELDWRITE-EXCEPTION
$$\frac{\delta = (v.f = v', \rho, pc, \theta) \quad (\texttt{null}, \ell) = openOpaque(v)}{CT \vdash \langle \delta :: \Delta; \mathcal{H} \rangle \rightarrow Exception}$$

E-FIELDLABELOF-CTX
$$\frac{\delta = (\texttt{fieldLabelOf}(e), \rho, pc, \theta) \quad \delta = (e, \texttt{fieldLabelOf}(\bullet) :: \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-IF-CTX
$$\frac{\delta = (\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2, \rho, pc, \theta) \quad \delta' = (e, \texttt{if } \bullet \texttt{ then } e_1 \texttt{ else } e_2 :: \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-SEQ-CTX
$$\frac{\delta = (e_1 \,;\, e_2, \rho, pc, \theta) \quad \delta' = (e_1, (\bullet \,;\, e_2) :: \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-SEQ
$$\frac{\delta = (v \,;\, e_2, \rho, pc, \theta) \quad \delta' = (e_2, \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-COMP
$$\frac{\begin{array}{c}\delta = (v_1' == v_2', \rho, pc, \theta) \qquad (v_1, \ell_1) = openOpaque(v_1') \qquad (v_2, \ell_2) = openOpaque(v_2') \qquad \ell_{o_1} = getObjL(v_1, \mathcal{H}) \\ \ell_{o_2} = getObjL(v_2, \mathcal{H}) \qquad pc' = pc \sqcup \ell_1 \sqcup \ell_2 \sqcup \ell_{o_1} \sqcup \ell_{o_2} \qquad v = \begin{cases} \texttt{true} & \text{if } v_1 = v_2 \\ \texttt{false} & \text{if } v_1 \neq v_2 \end{cases} \qquad \delta' = (v, \rho, pc', \theta)\end{array}}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

E-GETCONTEXTLABEL
$$\frac{\delta = (\texttt{getContextLabel}(), \rho, pc, \theta) \qquad \delta' = (pc, \rho, pc, \theta)}{CT \vdash \langle \delta :: \Delta, \mathcal{H} \rangle \rightarrow \langle \delta' :: \Delta, \mathcal{H} \rangle}$$

$$getObjL(v, \mathcal{H}) \stackrel{def}{=} \begin{cases} \ell_o & \text{if } \exists \varsigma, v = \varsigma \\ & \quad \wedge \mathcal{H}(\varsigma) = \langle cn, \mathbb{F}, \ell_f, \ell_o \rangle \\ \bot & \text{otherwise} \end{cases}$$

Fig. 14. Remaining Operational Semantics Rules