JRIF: Reactive Information Flow Control for Java[*] Technical Report

Elisavet Kozyri Owen Arden Andrew C. Myers Fred B. Schneider
Department of Computer Science
Cornell University
{ekozyri,owen,andru,fbs}@cs.cornell.edu

October 24, 2015

Abstract

Classic information flow systems conservatively define the security label associated with a derived value to be at least as restrictive as the security label on any input to that derivation. Because restrictions on information flow do not necessarily change monotonically over time, this definition requires programmers to invoke downgrading operations. A *reactive information flow* (RIF) specification for a value v gives (i) allowed uses for v and (ii) the RIF specification for any value that might be directly or indirectly derived from v. RIF specifications thus specify how transforming a value might alter how the result may be used, and that is more expressive than existing approaches for controlling downgrading. We implement a type system for RIF specifications by extending Jif, a dialect of Java that supports a form of classic information flow. By implementing the JRIF language and compiler, we show how a classic information-flow type system can be easily replaced with a more expressive RIF type system. We built example applications with JRIF, and we provide insights into the benefits of RIF-based security labels.

1 Introduction

Many security policies of interest can be formulated in terms of information flow. A *confidentiality policy* specifies what information may propagate to various prin-

[*]Supported in part by AFOSR grants F9550-06-0019 and FA9550-11-1-0137, National Science Foundation grants 0430161, 0964409, and CCF-0424422 (TRUST), ONR grants N00014-01-1-0968 and N00014-09-1-0652, and grants from Microsoft.

cipals; an *integrity policy* specifies what principals have influenced certain pieces of information. The classic implementation is in terms of labels (e.g. [27, 39, 15, 21, 10, 37, 32]), which ultimately are interpreted in terms of sets of principals. For confidentiality, we might associate label $\{\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots\}$ with a piece of information v to assert that only principals $\{\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots\}$ may read v, as well as all values derived from v. For integrity, this label would assert that v, and all values derived from v, can be trusted only if $\{\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots\}$ can be trusted.

Schemes involving labeled information typically have some means to synthesize labels for results of computations. Moreover, the appropriate label for $\mathtt{x}$ after executing $\mathtt{x} := F(\mathsf{y}, \mathsf{z})$, might be less restrictive or more restrictive than the labels on y and z, depending on F—for example, statistical aggregation might produce information with a less-restrictive confidentiality label. For $\mathtt{x} := F(\mathsf{y}, \mathsf{z})$, having the label on $\mathtt{x}$ be more restrictive than the labels of y and z is certainly safe. But now labels become ever more restrictive, creating a mismatch between the label of $\mathtt{x}$ and the expected security policy for the result of F.

Programming languages have been developed to support labels through a type system (starting with Volpano *et al.* [38]). The type of a variable defines a label that is associated with any value that variable stores. Languages (starting with Myers [26]) have included explicit operations for programmers to assert that the output of a computation is less restrictive than its inputs[1]. For instance, by writing code like

$$\mathtt{winner} := \mathtt{declassify}(\mathit{majority}(\mathtt{x}_{\mathsf{p}}, \mathtt{x}_{\mathsf{q}}, \mathtt{x}_{\mathsf{r}}, ...), \mathtt{public})$$

the label associated with $\mathtt{winner}$ allows the majority of $\mathtt{x}_{\mathsf{p}}, \mathtt{x}_{\mathsf{q}}, \mathtt{x}_{\mathsf{r}}, ...$ to flow to any principal, even if variable $\mathtt{x}_{\mathsf{r}}$, say, has a label that allows the stored value to flow only to principal r.

The need for an explicit reclassification operation can be seen as a weakness in the language of labels. The set of principals a label denotes needs to be able to evolve as a computation proceeds, consistent with the computation history of the labelled value. A more expressive label language would specify not only allowed uses of a value, but also appropriate labels to associate with derived values. Prior work restricts downgrading [18, 23, 30, 31, 14] or enforces information erasure [12], but these approaches cannot specify an arbitrary label for a value based on the history of operations deriving the value.

This paper describes a type system that supports Reactive Information Flow specifications [20], which express changes in confidentiality or integrity that are

[1]For confidentiality, the output is allowed to flow to more principals than the inputs, and for integrity, the output is trusted more than the inputs.

instigated by operations that derive new values. A new Java dialect, JRIF, extends Java to support RIF specifications by modifying the Jif compiler and runtime. Experiences with using JRIF as compared to Jif [25, 28], which supports explicit operations for declassification and endorsement, are discussed. A public release of the source code for the JRIF compiler and runtime, along with example applications, will be available soon.

Section 2 defines RIF automata, finite-state automata that represent RIF specifications. In Section 3, we present JRIF, which has a type system based on RIF automata, and we discuss the security condition that type correctness implies. Section 4 demonstrates the practicality of JRIF through two real programs, and explores the advantages of JRIF compared to a classic labelled type system like Jif. Our modifications to Jif that are likely to generalize to other RIF implementations are discussed in Section 5, and Section 6 gives comparisons of JRIF to related work including other language-based models for controlling declassification and endorsement.

2 RIF Specifications as Automata

A RIF specification maps a sequence of operations on a value to a set of principals. For confidentiality, these principals are authorized to read the value. For integrity, these principals must be trusted in order for the value to be trusted; they may have directly or indirectly influenced the computation of the value.

Many operations have equivalent effects on the information content of their inputs.We assume that operations of interest to a programmer are annotated with identifiers that represent one of those classes of equivalent operations. We call these identifiers *reclassifiers* since the confidentiality or integrity of the outputs of the associated operations might differ from that of the inputs. For example, in the assignment

$$\mathtt{z} := [\mathtt{x}\ \mathit{mod}\ \mathtt{y}]_F$$

reclassifier F identifies a class to which modulo belongs. A sequence of operations may be represented by a sequence $\vec{F}$ of reclassifiers. Our RIF specifications map sequences of reclassifiers to sets of principals.

One representation for RIF specifications uses *RIF automata*, finite-state automata whose states are mapped to sets of principals and whose transitions are directed by reclassifiers. For a value v tagged with RIF automaton ρ, the set of principals $\mathcal{A}(\rho)$ in the initial state of ρ restricts the use of v.

Each value deriving from v with a history of reclassifiers $\vec{F}$ is also associated with a RIF automaton. Specifically, these derived values are associated with ρ after performing the indicated transitions in $\vec{F}$. We write this derived RIF automaton as $\mathcal{T}(\rho, \vec{F})$. So, if $\vec{F}$ is the sequence of reclassifiers involved in deriving v' from v, then $\mathcal{A}(\mathcal{T}(\rho, \vec{F}))$ is the set of principals that restricts the use of v'.

Finite-state automata are attractive because they compactly represent mappings from a possibly infinite number of sequences of reclassifiers to sets of principals. The number of states in a RIF automaton could be large but we have found in practice that relatively small RIF automata are capable of representing many policies of practical interest.

Formally, a RIF automaton ρ is defined to be a 5-tuple $\langle Q, \Sigma, \delta, q_0, \lambda \rangle$, where:

– Q is a finite set of states,

– Σ is a finite set of reclassifiers,

– δ is a total, deterministic transition function $Q \times \Sigma \to Q$,

– q_0 is the initial state $q_0 \in Q$, and

– λ is a function from states to sets of principals.

So, functions $\mathcal{A}$ and $\mathcal{T}$ can now be defined as:

$$\mathcal{A}(\rho) \triangleq \lambda(q_0)$$
$$\mathcal{T}(\rho, \vec{F}) \triangleq \langle Q, \Sigma, \delta, \delta^*(q_0, \vec{F}), \lambda \rangle$$

where δ^* is the transitive closure of δ.

To specify both confidentiality and integrity, a value is tagged with a *c-automaton* ρ_c and an *i-automaton* ρ_i. A pair $\langle \rho_c,\ \rho_i \rangle$ of these automata forms a *RIF label*.[2] If a sequence $\vec{F}$ is applied to a value tagged with $\langle \rho_c,\ \rho_i \rangle$, then the resulting value is tagged with the RIF label

$$\langle \mathcal{T}(\rho_c, \vec{F}),\ \mathcal{T}(\rho_i, \vec{F}) \rangle$$

Changes to the confidentiality or integrity of a value have straightforward descriptions using RIF automata.

– For confidentiality, a reclassifier *triggers* a *declassification* when it causes a transition whose ending state is a superset of the principals contained in

[2]When the type of an automaton is clear from context, we will omit these subscripts.

its starting state. A reclassifier triggers a *classification* when it causes a transition whose ending state is a subset of the principals contained in its starting state.

- For integrity, these same two conditions imply that a reclassifier triggers a *deprecation* (a superset must be trusted) and an *endorsement* (a subset must be trusted), correspondingly.
- A reclassifier triggers a *reclassification* when the sets of principals in the starting and ending state of the corresponding transition have an arbitrary relation: one or more principals contained in the starting state might be excluded from the ending state, and/or one or more principals not contained in the starting state might be included in the ending state.

We illustrate how reclassifications are expressed using simple examples. Focusing on confidentiality, consider the following command, which assigns paper titles to reviewers, for a conference program committee, using a function $match$, which the programmer has annotated with the reclassifier $Mtch$.

$$\texttt{RevAsgn} := [match(\texttt{Titles}, \texttt{Revs})]_{Mtch} \tag{1}$$

Let A be the set of authors whose papers belong to the list `Titles`, and let R be the set of reviewers whose names appear in the list `Revs`. A possible policy for `Revs` is the c-automaton ρ_R in Figure 1. This automaton indicates that the value in `Revs` can be read initially by both the authors and the reviewers, but if an operation identified by reclassifier $Mtch$ is applied, then the resulting value (e.g., produced by $match$ in (1)) may only be read by reviewers (so authors cannot learn reviewer identity).

`Titles` might be tagged with c-automaton ρ_T in Figure 2. Then, the value in `RevAsgn` may be read by $\mathcal{A}(\mathcal{T}(\rho_R, Mtch)) \cap \mathcal{A}(\mathcal{T}(\rho_T, Mtch))$, which is R. So, a reclassification occurred—indeed, a classification. Notice, reclassifier $Mtch$ has different effect on the c-automata of `Revs` and `Titles`.

Now suppose that a publisher p receives a document `doc` that everyone trusts. The publisher is expected to publish `doc` in its entirety, but she is not supposed to publish an excerpt of `doc`[3], say by executing the following command since important context might be missing:

$$\texttt{exc} := [excerpt(\texttt{doc})]_{Xrpt} \tag{2}$$

[3]This example is inspired by `TruDocs` [36].

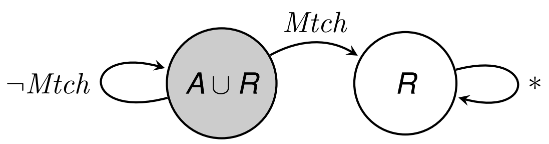


Figure 1: Automaton ρ_R. Initially, authors (A) and reviewers (R) are allowed to read, but after an operation identified as $Mtch$, the result is allowed to be read only by reviewers.

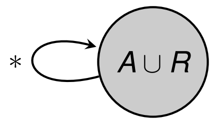


Figure 2: Automaton ρ_T. Read by authors (A) and reviewers (R) is allowed for this value.

The i-automaton in Figure 3 illustrates the desired integrity specification for doc. Initially, no principal must be trusted for doc to be trusted; but the result of isolating an excerpt out of doc is only as trusted as p. Here, reclassifier $Xrpt$ triggers a deprecation. Notice that, according to this i-automaton, if a *publicize* operation is instead applied to doc, then no principal must be trusted for the result to be trusted.

Restrictiveness Relation

Whenever a value will be stored into a variable, the RIF automaton that tags this variable should be at least as restrictive as the RIF automaton of the value. For c-automata, we say that ρ'_c is at least as restrictive as ρ_c, denoted $\rho_c \sqsubseteq_c \rho'_c$, if for all possible sequences of reclassifiers, the principals allowed to read the resulting value according to ρ'_c are also allowed by ρ_c. Relation $\sqsubseteq_c$ is formally defined as follows:

$$\rho_c \sqsubseteq_c \rho'_c \triangleq (\forall \vec{F}\colon \mathcal{A}(\mathcal{T}(\rho_c, \vec{F})) \supseteq \mathcal{A}(\mathcal{T}(\rho'_c, \vec{F}))) \tag{3}$$

For i-automata, ρ'_i is at least as restrictive as ρ_i, denoted $\rho_i \sqsubseteq_i \rho'_i$, if for all

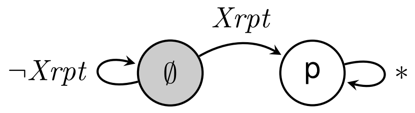


Figure 3: Automaton ρ_T of the document doc. When the excerpt operation (annotated with $Xrpt$) is applied, the result is deprecated to p.

possible sequences of reclassifiers, the principals that must be trusted according to ρ'_i include those that must be trusted according to ρ_i. So, relation $\sqsubseteq_i$ is defined as follows:

$$\rho_i \sqsubseteq_i \rho'_i \triangleq (\forall \vec{F}\colon \mathcal{A}(\mathcal{T}(\rho_i, \vec{F})) \subseteq \mathcal{A}(\mathcal{T}(\rho'_i, \vec{F}))) \tag{4}$$

We extend these restrictiveness relations to RIF labels by comparing components pointwise:

$$\langle \rho_c,\ \rho_i \rangle \sqsubseteq \langle \rho'_c,\ \rho'_i \rangle \triangleq \rho_c \sqsubseteq_c \rho'_c \wedge \rho_i \sqsubseteq_i \rho'_i.$$

A unique supremum of two RIF automata always exists, because RIF automata and the restrictiveness relation form lattices. The set of c-automata and relation $\sqsubseteq_c$ define a lattice with bottom ρ_{P}, a one-state automaton on the set P (everyone can read), and top $\rho_{\emptyset}$, a one-state automaton on the empty set (no one can read). Join operator $\sqcup_c$ denotes the supremum of two c-automata. If ρ_c is c-automaton $\langle Q, \Sigma, \delta, q_0, \lambda_c \rangle$, and ρ'_c is c-automaton $\langle Q', \Sigma', \delta', q'_0, \lambda'_c \rangle$, then:

$$\rho_c \sqcup_c \rho'_c \triangleq \langle Q \times Q', \Sigma \cup \Sigma', \delta_\times, \langle q_0, q'_0 \rangle, \lambda_\times \rangle$$

where $\delta_\times$ and $\lambda_\times$ are defined for $\langle q, q' \rangle \in Q \times Q'$ as

$$\delta_\times(\langle q, q' \rangle, F) = \langle \delta(q, F),\ \delta'(q', F) \rangle$$
$$\lambda_\times(\langle q, q' \rangle) = \lambda_c(q) \cap \lambda'_c(q')$$

The intersection ($\cap$) in defining $\lambda_\times$ means c-automaton $\rho_c \sqcup_c \rho'_c$ allows principal p to read the corresponding result only if p is allowed to do so by both ρ_c and ρ'_c.

For integrity, the lattice is inverted: the top element is ρ_{P} (everyone must be trusted), and the bottom element is $\rho_{\emptyset}$ (no one must be trusted). This makes sense because a value tagged with $\rho_{\emptyset}$ can be safely considered to be tagged with ρ_{P} (i.e., $\rho_{\emptyset} \sqsubseteq \rho_{\mathsf{P}}$), but not vice versa. The join operation $\sqcup_i$ is identical to $\sqcup_c$ except for function $\lambda_\times$:

$$\lambda_\times(\langle q, q' \rangle) = \lambda_i(q) \cup \lambda'_i(q')$$

Here, union ($\cup$) in defining $\lambda_\times$ implies that, for every sequence of reclassifiers and all principals p, p must be trusted according to i-automaton $\rho_i \sqcup_i \rho'_i$ whenever p must be trusted according to either ρ_i or ρ'_i.

The set of RIF labels and the restrictiveness relation $\sqsubseteq$ also define a lattice, with top $\langle \rho_{\emptyset},\ \rho_{\mathsf{P}} \rangle$ (no one may read, everyone must be trusted), bottom $\langle \rho_{\mathsf{P}},\ \rho_{\emptyset} \rangle$ (everyone may read, no one must be trusted), and with join operator $\sqcup$ defined pointwise:

$$\langle \rho_c,\ \rho_i \rangle \sqcup \langle \rho'_c,\ \rho'_i \rangle \triangleq \langle \rho_c \sqcup \rho'_c,\ \rho_i \sqcup \rho'_i \rangle$$

Suppose x has label $\langle \rho_c,\ \rho_i \rangle$ and y has label $\langle \rho'_c,\ \rho'_i \rangle$. Then to obtain a label for the expression $x + y$, we can compute the supremum of the c-automata and the i-automata and use the resulting automata to tag the expression.

3 JRIF

Jif [25, 28] extends Java's types to incorporate information flow labels.[4] Jif handles information flows introduced by features of Java, including exception handling as well as allocation and updating of heap locations. Methods in Jif are annotated with labels to support compositional type checking and separate compilation. Jif also supports class declarations with label parameters and with a limited form of dependent types, which permits dynamic labels to be used for type checking. The Jif solver automatically infers the labels of local variable types, reducing the annotation burden on the programmer.

JRIF (Jif with Reactive Information Flow) replaces the labels in Jif with RIF labels. Our implementation preserves almost all of the abstractions provided by Jif, including label parameters, dependent label types, and label inference. Programmers tag fields, variables, and method signatures with RIF labels, and the JRIF compiler checks whether the program satisfies these specifications.

3.1 Syntax of RIF Labels and Annotated Expressions

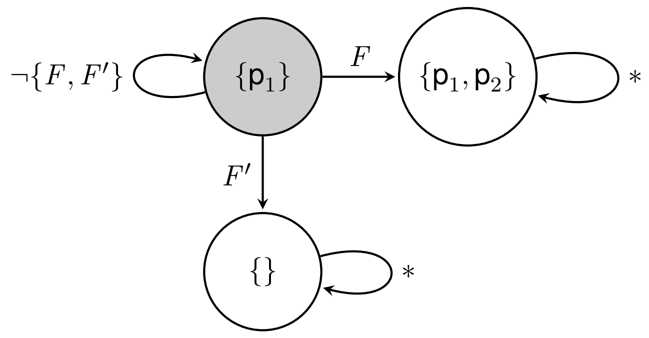
The JRIF syntax of a RIF label L is given in Figure 4. Using this syntax, the c-automaton in Figure 5a is coded as in Figure 5b. Reclassifications that are not specified in the syntax of a RIF label are taken to be transitions whose starting and ending states are identical.

JRIF programmers can annotate any expression with a reclassifier. The process of annotating expressions is relatively simple, and the type system quickly reveals mismatched assumptions between labels and reclassifiers. An alternative would be for the language definition to provide reclassifiers for all basic operations. However, this alternative would likely result in larger and harder to understand RIF automata and would be difficult to extend to user-defined functions. Thus, for honest-but-clumsy programmers, our approach seems like the best alternative. For simplicity of presentation, we continue to write $[\mathcal{E}]_F$ when expression $\mathcal{E}$ is annotated with reclassifier F even though the actual JRIF syntax is slightly different.

[4]Because Jif extends Java 1.4, it does not support Java Generics, introduced in Java 1.5.

$$\begin{array}{lcl}
L & ::= & \{A_c; A_i\} \\
A_c & ::= & \mathsf{c}\ [\mathit{ListOfTerms}] \\
A_i & ::= & \mathtt{i}\ [\mathit{ListOfTerms}] \\
\mathit{ListOfTerms} & ::= & T \mid T, \mathit{ListOfTerms} \\
T & ::= & \mathit{State} \mid \mathit{InitialState} \mid \mathit{Transition} \\
\mathit{State} & ::= & \mathit{ID} : \{\mathit{ListOfPrincipals}\} \\
\mathit{InitialState} & ::= & \mathit{ID}* : \{\mathit{ListOfPrincipals}\} \\
\mathit{Transition} & ::= & \mathit{ID} : \mathit{ID} \rightarrow \mathit{ID}
\end{array}$$

Figure 4: Syntax for labels, where ID represents an alphanumeric string.



(a)

$$\mathsf{c}\ [q_0*{:}\{\mathsf{p}_1\},\ q_1{:}\{\mathsf{p}_1, \mathsf{p}_2\},\ q_2{:}\{\},\ F{:}q_0 \rightarrow q_1,\ F'{:}q_0 \rightarrow q_2]$$

(b)

Figure 5: Syntactic representation of a c-automaton

RIF labels are more expressive than Jif labels, so it is not surprising that RIF labels tend to be longer. Changes to confidentiality and integrity specified in RIF labels are not included in JIF (instead, additional code is required). Thus, reclassifications in JRIF have a concise description, whereas declassifications and endorsements in Jif are more verbose since they have a target label and, sometimes, must include the source label as well. Finally, a single JRIF reclassifier can trigger both a change in confidentiality and integrity; Jif requires a separate `declassify` and `endorse` to effect that same change.

3.2 Type checking

The JRIF type system infers the types of expressions and annotated expressions from the types of variables. The type of an expression $\mathcal{E}$ is the join of the RIF labels of all variables in $\mathcal{E}$. The type of an annotated expression $[\mathcal{E}]_F$ is the RIF label of $\mathcal{E}$ after performing an F transition.

Information flows may be explicit or implicit. An *explicit flow* occurs when information flows from one variable to another as in the assignment

$$\mathtt{x} := [\mathcal{E}]_F \,. \tag{5}$$

An *implicit flow* occurs when assignment takes place because of a conditional branch, as in the **if**-statement

$$\textbf{if } [\mathcal{E}]_F \textbf{ then } \mathtt{x} := [\mathcal{E}']_{F'} \textbf{ else } \mathtt{x} := [\mathcal{E}'']_{F''} \,. \tag{6}$$

Knowing the value of `x` implies whether the $\mathcal{E}$ evaluates to `true` or `false`.

JRIF, like other static information flow languages, controls implicit flows using a *program counter* (`pc`) label to represent the confidentiality and integrity of the control flow of the program. Assignment (5) is secure if both the `pc` label and the type of $[\mathcal{E}]_F$ can flow to the type of `x`. In other words, the RIF label of `x` is at least as restrictive as the `pc` label and the label of $[\mathcal{E}]_F$. When control flow branches, as in (6), the `pc` label increases to be at least as restrictive as the current `pc` label and the label of $[\mathcal{E}]_F$. This ensures that assignments in either branch are constrained to variables with RIF labels at least as restrictive as the label of $[\mathcal{E}]_F$.

3.3 What Type Safety Enforces

The RIF automaton associated with a variable specifies which principals may access that variable. A c-automaton restricts readers; an i-automaton restricts writers. But information flow control goes beyond such access control. In its general form, it concerns whether changes to values in a certain set of variables can cause changes to values in another set of variables—so-called noninterference [16]. For confidentiality, noninterference requires that changes to values in variables that principal p cannot read initially do not cause changes to values in variables that p can read during program execution. Equivalently, if the initial states of two program executions agree on values in variables that principal p can read, then these executions agree on the values that get assigned to variables that p can read. For integrity, noninterference requires that variables whose trust initially depends on trusting p do not cause changes to values in variables whose trust does not depend on p during program execution. Obviously, these conditions must hold for each principal p.

Reclassifications complicate a definition of information flow by changing what values are of concern during an execution. For example, if a reclassifier causes a transition that permits p to read the result of an operation on secret variables, then classic noninterference is violated, even though the RIF specifications of the variables are not. To accommodate reclassification in defining information flow, it suffices to partition execution into segments that each satisfy noninterference. Segments are delimited by reclassifications relative to a single principal p.[5]

Piecewise Noninterference (PWNI) for a principal p. A *piece* is an interval of a program's execution beginning with the initial state or a reclassification and ending with a declassification, endorsement, or termination.

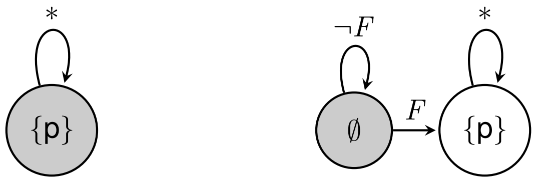
- If two pieces start at the same point of execution with states that agree on variables that p can read and on newly declassified values, then these pieces agree on the values that get assigned to variables that p can read throughout execution.
- If two pieces start at the same point of execution with states that agree on variables whose trust does not depend on p and on newly endorsed values, then these pieces agree on the values that get assigned to variables whose trust does not depend on p throughout execution.

To better understand what PWNI provides, consider the following command:

$$\mathsf{z} := [\mathtt{x}\ mod\ 4]_F;\ \mathsf{z} := \mathtt{x}\ mod\ 2 \tag{7}$$

Assume z is tagged with the c-automaton in Figure 6a, and $\mathtt{x}$ is tagged with the c-automaton in Figure 6b. This sequence of commands satisfies PWNI for principal

[5]A more careful definition of Piecewise Noninterference is found in [20].



(a) p is allowed to read (b) p is allowed to read after F

Figure 6

p. This is because if two pieces start with states that agree on the declassified value of $\mathtt{x}\ mod\ 4$, then these states will also agree on the value of $\mathtt{x}\ mod\ 2$. Thus, the two pieces will agree on both value being assigned to z. Notice, PWNI would not be satisfied if, for instance, the second assignment was $\mathtt{z} := \mathtt{x}\ mod\ 8$, because $\mathtt{x}\ mod\ 8$ leaks more information to p than what is allowed to be declassified in $\mathtt{x}\ mod\ 4$.

3.4 Dynamic labels and label parameters

Sometimes an information flow specification will become known only at execution time. It is also desirable to reuse code for multiple information flow specifications. Jif provides *dynamic labels* and class declarations parameterized with *label parameters* for that purpose. Label parameters are *reified*: each instance of a parameterized class definition contains runtime values for its label parameters. In most cases, dynamic labels and label parameters may be used interchangeably.

We adapted Jif's support for dynamic labels and label parameters to support RIF labels. Thus, RIF labels in JRIF may be instantiated as runtime values: they may be constructed programmatically, passed as method arguments, stored in fields and variables, and compared dynamically. Furthermore, the labels of static type declarations may range over label parameters and label-valued fields and variables.

Since the actual RIF label that a dynamic label or label parameter denotes is not known at compile time, the JRIF type system requires the programmer to insert checks that would prevent unsafe flows at runtime. JRIF programmers can compare the restrictiveness (§2) of dynamic RIF labels at runtime using expressions similar to those comparing dynamic Jif labels. In JRIF, it is also necessary to reason dynamically about transitions on RIF labels. For example, consider

$$\mathtt{y} := [\mathtt{x}\ mod\ 4]_F \tag{8}$$

where $\mathtt{x}$ is tagged with dynamic label ℓ_1, and $\mathtt{y}$ is tagged with dynamic label ℓ_2. This assignment statement is secure only when $\mathcal{T}(\ell_1, F) \sqsubseteq \ell_2$ holds, which depends on the values of ℓ_1 and ℓ_2.

To ensure that $\mathcal{T}(\ell_1, F) \sqsubseteq \ell_2$, the programmer would code

$$\textbf{if } \mathcal{T}(\ell_1, F) \sqsubseteq \ell_2 \textbf{ then } \mathtt{y} := [\mathtt{x}\ \mathit{mod}\ 4]_F \tag{9}$$

At compile time, this constraint provides the type system with the necessary information about ℓ_1 and ℓ_2. Specifically, the type system may assume $\mathcal{T}(\ell_1, F) \sqsubseteq \ell_2$ holds when the `then` clause starts executing since that code is executed only if the dynamic check succeeds. To execute the check, the runtime system constructs the automaton that results from an F transition on ℓ_1 and checks whether ℓ_2 is at least as restrictive.

In Jif, a declassification has the same effect on all labels. So in the Jif assignment

$$\mathtt{y} := \mathtt{declassify}(\mathtt{x}\ \mathit{mod}\ 4, \ell_2)$$

any value stored in `x` will be declassified[6]. To achieve the same effect as (9) in Jif, the programmer must use an auxiliary variable, say `d`, to control whether $\mathtt{x}\ \mathit{mod}\ 4$ should be declassified.

$$\textbf{if } \mathtt{d} \textbf{ then } \mathtt{y} := \mathtt{declassify}(\mathtt{x}\ \mathit{mod}\ 4, \ell_2) \tag{10}$$

Notice that it is the programmer's responsibility to ensure that `d` is inspected before any declassification of `x`; the type system provides no assistance about that. This is error-prone, compared with (9), where an error from the JRIF type system would alert the programmer that a dynamic check is necessary.

Dynamic labels and label parameters also relieve some of the annotation burden of writing and rewriting long JRIF labels as the functionality of a program evolves during development. With dynamic labels and label parameters, programmers can specify only the properties of RIF labels that are required for a given method or class to be secure. This in turn reduces the number of classes that must be modified when RIF labels change to accommodate new functionality.

3.5 Constraints

To reduce the need for redundant dynamic checks, a JRIF programmer may declare relationships between labels that must hold when the method is called. These

[6]Robust downgrading places restrictions on the label, but they are orthogonal to this discussion.

relations are called **where**-*constraints*. Consider the method

$$\begin{array}{l}\textbf{int}\{l_2\}\ \mathit{mod4}\,(\{l_1\}\ \mathtt{x})\ \textbf{where}\{\mathcal{T}(l_1,F) \sqsubseteq l_2\}\{\\ \quad \textbf{return}\ [\mathtt{x}\ \mathit{mod}\ 4]_F\ ;\\ \}\end{array}$$

for l_1 and l_2 dynamic labels or label parameters. The constraint **where**$\{\mathcal{T}(l_1, F) \sqsubseteq l_2\}$ permits the type system to assume $\mathcal{T}(l_1, F) \sqsubseteq l_2$ while type checking the body of method $\mathit{mod4}$. To ensure $\mathit{mod4}$ executes only when $\mathcal{T}(l_1, F) \sqsubseteq l_2$ holds, the type system need only check that the **where**-constraint holds at every call site naming $\mathit{mod4}$. For instance, the expression $[\mathtt{x}\ \mathit{mod}\ 4]_F$ in (9) could be replaced by a call to $\mathit{mod4}$.

4 Program Examples using JRIF

4.1 Battleship

The Jif distribution [28] includes an implementation of the Battleship game. Battleship is a good example because confidential information must be declassified over the course of the game. Integrity is important here, too. Ship coordinates are initially fixed and secret, but revealed when opponents guess their coordinates correctly. Also, players must not be able to change the position of their ships after the initial placements. Confidentiality or integrity policies in this game prevent a player from cheating.

We found that simple c-automata were expressive enough to specify confidentiality policies for the ship-coordinates of each player. That policy is:

- Values derived from the ship-coordinates selected by player p_1 should be read only by p_1, because opponent player p_2 is not allowed to learn the position of p_1's ships.
- The result of whether a ship of p_1 has been hit by the opponent player p_2 may be read by everyone, including p_2.

A c-automaton that expresses this policy appears in Figure 7, where Q is the reclassifier for the operation that checks whether an opponent's attack succeeded.

The integrity policy of ship-coordinates can be expressed using a simple i-automaton, too. Once p_1 selects the coordinates of her ships, they are as trusted as p_1. After ship-coordinates are chosen, they may not be changed during the game.

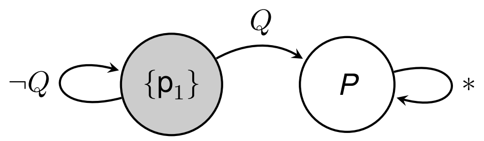


Figure 7: A c-automaton for ship-coordinates.

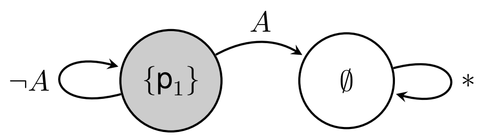


Figure 8: An i-automaton for ship-coordinates.

So, before the game actually starts, there is a game operation whose reclassifier raises the integrity of all ship-coordinates so that neither player can make changes. An i-automaton that expresses this policy is presented in Figure 8, where A is the reclassifier annotating the operation that accepts the initial coordinates.

The implementation of Battleship in JRIF was derived by making just a few modifications to the Jif implementation. We replaced Jif labels with RIF labels, and we replaced declassification or endorsement commands with reclassifications. Interestingly, methods in the Jif implementation that involved only label parameters and dynamic labels could be used without any modification in the JRIF implementation. We conjecture that any program written in Jif can be easily ported into JRIF.

Compared to Jif, JRIF provides better separation between program logic and information flow policies. Suppose a programmer decides some input value—the player's name, for instance—should not be declassified. In JRIF, this change involves modifying the RIF label declaration on any field storing the player's name. Specifically, the c-automaton of the updated label would be inspected and edited so it contains no transitions to automaton states that contain additional principals.

In contrast, to accommodate this change in Jif, the programmer must not only remove all declassification commands that involve the name field explicitly, but also must remove all declassification commands that involve any expressions to which it flows. Getting these deletions is error prone since the programmer must reason about the flow of information in the code.

We found the compilation and execution time of the JRIF version of Battleship to be comparable to the Jif version . On an Intel Core i5 (Intel Core i5 CPU

M 460 @ 2.53GHz $\times$ 4) processor with 4G of RAM, 7 seconds are required for compiling each version of Battleship (Jif version is $\sim$ 615 LOC, and JRIF version is $\sim$ 628 LOC). To measure the execution time, we ran a scripted Battleship game 1000 times in a loop. The Jif version executed 1000 games in 7 seconds while the JRIF version required 21 seconds. The increase is mostly due to the overhead of creating dynamic labels in JRIF, which currently requires more method calls than Jif's dynamic labels.

4.2 Shared Calendar

To further explore the expressive power of RIF labels, we developed a shared calendar application from scratch in JRIF. All four different kinds of reclassifications (declassification, classification, endorsement, and deprecation) are required in this application. Also, users may choose dynamic RIF labels to associate with values, so the same reclassifier could have different effects on values with different specifications.

Operations involved in our shared calendar include:

– Create a personal or shared event.

– Invite a user to participate in a shared event.

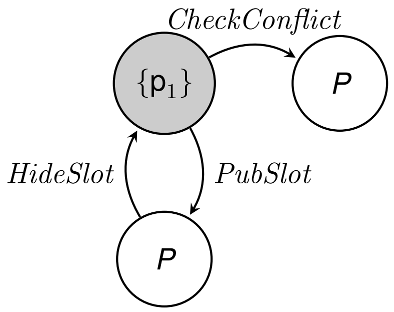
– Accept an invitation to participate in a shared event. Reclassifier: Accept

– Cancel a shared event. Reclassifier: Cancel

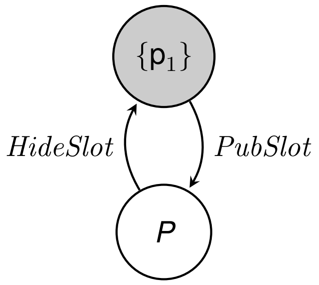
– Publish/hide an event date and time. Reclassifiers: $\mathit{PubSlot}$ and $\mathit{HideSlot}$

– Check and announce a conflict between personal events (not cancelled events) and an invitation for a new shared event. Reclassifier $\mathit{CheckConflict}$.

Figure 9 illustrates some c-automata for events created by principal p_1. The c-automaton in Figure 9a permits a full declassification triggered by the reclassifier $\mathit{CheckConflict}$, whereas the c-automaton in Figure 9b does not. Figure 10 gives an i-automaton for the events of p_1. Once an event with this RIF automaton is accepted, the event has top integrity since all of the attendees have agreed to participate (no one is able to modify this event thereafter). However, if the event is cancelled, it should still remain in the records of the user, but it should be treated with low integrity, because it should not influence the functionality of the user's calendar, say the result of $\mathit{CheckConflict}$ operation.



(a) A c-automaton that permits declassification for conflict checking.



(b) A c-automaton that does not permit declassification for conflict-checking.

Figure 9: Two RIF automata for event confidentiality.

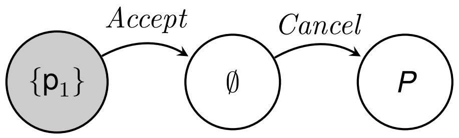


Figure 10: An i-automaton for an event. The self-loops are omitted; for instance, the result of applying *CheckConflict* to a canceled event has low integrity.

```
if $(\mathcal{T}(\texttt{lblEvt}, \mathsf{C}) \sqsubseteq \{c[q0{*} : P]; i[q1{*} : \emptyset]\}$
        $\&\&\ \mathcal{T}(\texttt{lblCal}, \mathsf{C}) \sqsubseteq \{c[q0{*} : P]; i[q1{*} : \emptyset]\})$
then if $([\texttt{cal.hasConflict}(\texttt{event}, \texttt{lblEvt}, \texttt{lblCal})]_{\mathsf{C}})$
        then result := true; //Conflict detected
        else result := false; //No conflict
```

Figure 11: Checking if the conflict is allowed to be declassified and endorsed, where C corresponds to reclassifier *CheckConflict*. Here, `lblEvt` is the dynamic label of the requested shared event `event`, `lblCal` is the dynamic label of events in the calendar `cal`, against which the conflict will be checked, and method `hasConflict` returns `true` if a conflict is detected.

We use dynamic labels and label parameters extensively in the shared calendar application. The event class has label parameters that are typically instantiated by dynamic labels. These RIF labels tag components of events, such as the time, date, duration, and description. Since events have parameterized types, the methods that manipulate events are also either parameterized or make use of dynamic labels in their type declarations. Figure 11 excerpts from the conflict-checking method. Here, the label of the event is dynamically checked to see whether it permits the check to be declassified and endorsed before performing the conflict check.

A user's events may be tagged with different dynamic labels. For example, a user might pick the c-automaton in Figure 9a for some events but pick the c-automaton in Figure 9b for others. Events can have different i-automata, too. Unshared events have the i-automaton in Figure 10, but accepted events can be treated with higher integrity and thus tagged with the i-automaton denoted by taking the $Accept$ transition. Additionally, the time slot of some events could be either hidden or public. To accommodate these heterogeneously labeled events, we store events in a data structure that makes it easier to aggregate events with different labels. The data structure has two fields: one is an event, and the other is a label. Before processing an event, its label is checked to prevent unspecified flows. Such data structures are common in Jif programs, and they are studied formally in [40].

5 Extending the Jif Compiler

Our strategy for extending Jif should extend to compilers for other information flow languages. This should not be so surprising: RIF labels expose the same interface to a type system as the native labels do. The difference is in how RIF labels evolve.

Our strategy for building JRIF involved three steps:

1. Add syntax for RIF labels and for annotating expressions with reclassifiers.
2. Add typing rules for annotated expressions (according to §3.2).
3. Modify the type checker to handle this more expressive class of labels:
   (a) implement the restrictiveness relation (§2) on RIF labels,
   (b) add an axiom stiputating that this relation is monotone with respect to transition relation $\mathcal{T}$,

Item (3b) is essential for supporting our richer language of label comparisons and **where**-constraints. For example, if a programmer introduces **where**-constraint

$l_2 \sqsubseteq l_1$ (or equivalently, checks dynamically that this relation holds), then the type checker should be able to deduce that $\mathcal{T}(l_2, F) \sqsubseteq \mathcal{T}(l_1, F)$ holds for every F.

In total, we added 6k lines of code to Jif (which is a total of 230k LOC), in order to get one compiler for JRIF. Out of the 494 Java classes used in Jif, we modified only 31 and added 48 new classes. Of these new classes, 37 are extensions of Jif classes; primarily abstract syntax tree nodes for labels, confidentiality and integrity policies, and code generation classes. Thus most of the effort in building JRIF focused on extending Jif's functionality rather than on building new infrastructure.

Some features of Jif are orthogonal to enforcing RIF labels, and we have ignored them for the time being. For instance, Jif uses authority and policy ownership to constrain how labels may be downgraded. Since RIF labels are concerned with what operation is applied to what value, authority and ownership is orthogonal to the enforcement of RIF specifications. As discussed below in §6.1, future versions of JRIF could extend the type system to support these constraints on downgrading as a complement to security guarantees offered by RIF specifications.

6 Related work

Expressive structures, like automata, have been used in prior work to represent information flow specifications. Program dependence graphs [17, 19], which represent data and flow dependencies between values have been used to specify declassifications that are allowed. Also, Rocha et al. [30, 31] employ *policy graphs* to specify sequences of functions that cause declassifications. However, these approaches do not handle arbitrary reclassifications; they handle only declassifications.

Declassifications are usually caused by *trusted processes* [4], which are permitted to violate noninterference. Several approaches that control declassification (e.g., [18]) employ some notion of trusted processes or code. Early versions of Jif used *selective downgrading* [29], which refines this idea with policies that are owned [9] by principals who may differ in which code they trust. These systems enforce a form of *intransitive* flow policy [33] since direct flows that do not involve the declassifying operation are prohibited.

Many models have been proposed for expressing and enforcing policies that permit principled disclosures of sensitive information, but PWNI is the first to consider classifications and protect the new secrets that classifications yield. *Gradual release* (GR) [2] requires that designated declassifications be the only points where an attacker's knowledge about initial secret variables increases. GR does not specify what secret values can be declassified. This is expressed in: *conditional gradual release* [3], *delimited release* [34], and *relaxed noninterference* [22].

Chong and Myers [14] introduce information flow specifications that use conditions on program state as a basis for deciding when a value may be declassified or should be erased.[7] RIF automata can be modified to express such specifications, by associating reclassifiers with conditions on states in some simple predicate language. Such an approach would generalize the policies expressible by Chong and Myers [14] since downgrading and erasure policies permit only linear sequences of conditions whereas automata admit more general structures.

Sabelfeld et al. [35] introduce a four-dimension categorization (what, where, who, when) of declassification. In JRIF, a reclassifier that causes a declassification indicates "what" will be declassified and "where" in the program.

Broberg et al. [8] characterize dynamic policies based on a three-level *hierarchy of control*. Using the authors' terminology, the components of RIF automata are described as follows: automata states are Level 0 controls, function $\mathcal{A}$ is a Level 1 control (a *determining function*), function $\mathcal{T}$ is a Level 2 control (a *meta policy*). However, whereas Broberg et al. [8] consider these controls to be the same for all values in a program, in JRIF this hierarchy is different for each RIF automaton.

6.1 RIF and Robustness

Another approach to enforcing principled disclosures is *robust downgrading* [13] used in Jif [28] and Fabric [24, 1]. Robust downgrading requires that downgrades occur only in high-integrity contexts, preventing untrusted principals from influencing what (and whether) sensitive information is disclosed. A significant advantage of robust downgrading is that, unlike selective downgrading, it provides an end-to-end security guarantee: an untrusted principal cannot cause a robust program to disclose information.

JRIF enforces an end-to-end security guarantee distinct from, and orthogonal to, robust downgrading. Whereas robust downgrading ensures attackers cannot control the decision to downgrade or (in the case of declassification) the information being declassified, JRIF guarantees that disclosed values are produced only by specified sequences of operations. However, RIF specifications are compatible with robust downgrading. JRIF's type system could be extended with a robust enforcement mechanism to enforce RIF specifications and robustness simultaneously. There are two options.

[7]Chong and Myers [11] also proposes extension to operations other than declassification and erasure.

One option is to extend JRIF's type system to restrict where reclassifiers may appear. To enforce robust downgrading, a reclassifier that causes a declassification to p or an endorsement of p would be restricted to code where control flow integrity does not require trust in p. In other words, p should not be able to control whether the operation associated with the reclassifier occurs. Furthermore, in the case of declassification, the integrity of the value being declassified should also not require trust in p. These constraints are a straightforward adaptation of Jif's rules for robust downgrading that use the starting and ending state of a RIF automaton transition to identify declassifications or endorsements.

Another option further generalizes robust downgrading but preserves the first option as a special case. Rather than implicitly identifying transitions that trigger declassification and endorsement, RIF automata could be extended by requiring each transition to be annotated with a set of principals that may influence the reclassification. The JRIF type system would restrict each reclassifier to appear in code whose control flow integrity is higher than the integrity specified in all transitions that reclassifier triggers. Furthermore, the integrity of the inputs to the operation the reclassifier annotates should be higher than the integrity specified in all transitions the reclassifier triggers in c-automata. The Jif-style approach from the first option can be implemented by specifying that the integrity on all declassifying or endorsing transitions are based on which principals are included or excluded. The extra expressiveness of the second approach seems as though it might have interesting applications, but we leave exploration of these to future work.

6.2 Capability-based systems

Many recent systems for information flow control are based on capabilities, including Flume [21], HiStar [39], Asbestos [15], Aeolus [10], Laminar [32], and LIO [37]. We focus our discussion on Flume, but a similar approach should apply to other systems.

Flume extends standard operating system abstractions with information flow control. Confidentiality and integrity policies are represented in Flume with unforgeable tokens, called *tags*. System resources are annotated with *labels*, which are collections of tags. Each process has an associated *process label*, which conservatively tracks the confidentiality and integrity policy on the process's memory. When a process performs input operations on sensitive data, the restrictiveness of the process label is raised by adding that resource's tags to the label. Output operations are constrained to affect resources with labels that are at least as restrictive as the current process label. For instance, if a process reads a secret file, any subsequent attempt to write to a public file will receive an error.

This mechanism alone is usually too restrictive; certain outputs of a program might not actually depend on any secret data, or the purpose of the program may actually be to release secret data in a controlled way. Thus, Flume also assigns to each process a set of capabilities that specify which tags it is permitted to add or remove from its process label. For instance, to add or remove a tag t, a process must have capability t^+ or t^-, respectively. Removing a tag from the process label is equivalent to declassification or endorsement.

Consider the following scenario. Alice has two files: `diary.txt`, where she keeps a personal journal, and `pwds.db`, where she stores passwords. Both files contain very sensitive information, so she adds a tag, *secret*, to their labels. She gives her editor the $secret^+$ capability, but not $secret^-$. This capability enables the editor to read `diary.txt`, but prevents it from outputting its contents to the network or to a file lacking the *secret* tag. In order to read the password file, she gives her password manager the $secret^+$ capability, but also the $secret^-$ capability so that the passwords can be used to log in to remote hosts.

Unfortunately, this gives the password manager more power than Alice may have intended since it may both read file `diary.txt` and export it to the network. In Flume, Alice's only option is to create separate tags for each file to distinguish secrets that should never be exported and to carefully assign capabilities to processes accordingly.

Extending Flume with RIF specifications would provide a better option. As in Jif, we can replace Flume labels with RIF automata, but where the states of these automata are sets of tags. Thus, each system resource is associated with a RIF automaton, and the process label is a RIF automaton that is at least as restrictive as the current process's memory. Instead of permitting processes to directly add or remove tags, processes receive capabilities for performing transitions on the process label's RIF automaton. Output operations are constrained to resources whose RIF automata are at least as restrictive as the process label.

RIF specifications for Flume would allow Alice to express her policies more directly. For `diary.txt`, she assigns a RIF automaton with a single state: *secret*. For `pwds.db`, she assigns an automaton with two states, *secret* and *public*, and a transition between them called *login*. Then granting the *login* capability to her password manager does not allow it leak `diary.txt`, because that file's automaton remains in the *secret* state after the *login* transition.

6.3 Paragon and Paralocks

The Paragon [7] programming language has information flow control based on Paralocks [6, 5]. Paragon policies are expressed in terms of information sources and sinks called *actors* and guarded by predicates called *locks*.

Paragon policies determine whether a flow of information to an actor is permitted. When a lock is *open*, flow is permitted. Paragon uses a type-and-effect system to track lock state at each program point, and the compiler statically verifies that all flows are permitted. While Paragon policies can express simple sequences of operations, Paragon's policy language is not expressive enough to encode arbitrary finite-state automata. Furthermore, encoding sequences directly as Paragon policies is somewhat cumbersome and detracts from the otherwise elegant declarative policy language Paragon provides. Thus, extending Paragon with RIF specifications would provide more expressive information flow specifications to developers.

One could extend Paragon's policy language so that the initial state of a RIF automaton defines the lock currently being enforced. Locks specify the set of actors to which information may flow. A transition in a RIF automaton would cause a new lock to be enforced. This design composes the enforcement of policies based on lock-state with the enforcement of RIF specifications in an interesting way. It allows developers to express policies that permit the lock predicates that are enforced to evolve based on the sequence of operations that derived the labeled value.

7 Conclusion

We defined JRIF, a Java dialect and compiler that implements Reactive Information Flow specifications based on finite-state automata. RIF labels specify the allowed uses of the values they are associated with, along with the RIF label to associate with a derived value. JRIF was a straightforward extension of the Jif compiler and runtime, demonstrating that RIF specifications are easily applied to existing languages that did not anticipate them, but do support information flow types.

JRIF's type system is more expressive than classic information flow type systems. JRIF allows programmers to specify rich policies based on the sequence of operations used to derive a value. Existing systems can emulate such policies in the state and control flow of a program, but doing so makes the code more complex and provides few security guarantees. In contrast, JRIF's type system enforces end-to-end security guarantees that ensure a value derived from inputs is protected according to a policy that corresponds to the sequence of operations involved in this derivation.

We evaluated JRIF by programming two examples: an implementation of Battleship, and a shared calendar application. Our implementation of Battleship demonstrates that applications developed with Jif may be ported easily to JRIF; the shared calendar demonstrates separation between policies and program logic that JRIF enables.

References

[1] O. Arden, M. D. George, J. Liu, K. Vikram, A. Askarov, and A. C. Myers. Sharing mobile code securely with information flow control. In *IEEE Symp. on Security and Privacy*, pages 191–205, May 2012.

[2] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *IEEE Symp. on Security and Privacy*, pages 207–221, May 2007.

[3] A. Banerjee, D. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *IEEE Symp. on Security and Privacy*, pages 339–353, 2008.

[4] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975. Available as DTIC AD-A023 588.

[5] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Programming Languages and Systems*, pages 180–196. Mar. 2006.

[6] N. Broberg and D. Sands. Paralocks—role-based information flow control and beyond. In *37th ACM Symp. on Principles of Programming Languages (POPL)*, Jan. 2010.

[7] N. Broberg, B. van Delft, and D. Sands. Paragon for practical programming with information-flow control. In *11th ASIAN Symposium on Programming Languages and Systems, APLAS 2013*, pages 217–232. Springer, 2013.

[8] N. Broberg, B. van Delft, and D. Sands. The anatomy and facets of dynamic policies. In *IEEE Symp. on Computer Security Foundations*. IEEE, 2015.

[9] H. Chen and S. Chong. Owned policies for information security. In *17th IEEE Computer Security Foundations Workshop*, June 2004.

[10] W. Cheng, D. R. K. 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 *2012 USENIX Annual Technical Conference*, June 2012.

[11] S. Chong and A. C. Myers. Security policies for downgrading. In *Proc. 11th ACM Conference on Computer and Communications Security*, pages 198–209, Oct. 2004.

[12] S. Chong and A. C. Myers. Language-based information erasure. In *18th IEEE Computer Security Foundations Workshop*, pages 241–254, June 2005.

[13] S. Chong and A. C. Myers. Decentralized robustness. In *19th IEEE Computer Security Foundations Workshop*, pages 242–253, July 2006.

[14] S. Chong and A. C. Myers. End-to-end enforcement of erasure and declassification. In *IEEE Symp. on Computer Security Foundations*, pages 98–111, June 2008.

[15] 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 *20th ACM Symp. on Operating System Principles (SOSP)*, Oct. 2005.

[16] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.

[17] 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*, 8(6):399–422, 2009.

[18] B. Hicks, D. King, P. McDaniel, and M. Hicks. Trusted declassification: high-level policy for a security-typed language. In *1st ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 65–74. ACM, 2006.

[19] A. Johnson, L. Waye, S. Moore, and S. Chong. Exploring and enforcing security guarantees via program dependence graphs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–302, New York, NY, USA, June 2015. ACM Press.

[20] E. Kozyri and F. B. Schneider. Reactive information flow specifications: Foundation and types. In prep.

[21] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *21st ACM Symp. on Operating System Principles (SOSP)*, 2007.

[22] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *32nd ACM Symp. on Principles of Programming Languages (POPL)*, Long Beach, CA, Jan. 2005.

[23] P. Li and S. Zdancewic. Practical information-flow control in web-based information systems. In *18th IEEE Computer Security Foundations Workshop*, pages 2–15, 2005.

[24] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *22nd ACM Symp. on Operating System Principles (SOSP)*, pages 321–334, Oct. 2009.

[25] A. C. Myers. JFlow: Practical mostly-static information flow control. In *26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, Jan. 1999.

[26] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *16th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, Oct. 1997.

[27] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, Oct. 2000.

[28] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif 3.0: Java information flow. Software release, `http://www.cs.cornell.edu/jif`, July 2006.

[29] F. Pottier and S. Conchon. Information flow inference for free. In *5th ACM SIGPLAN Int'l Conf. on Functional Programming*, ICFP '00, pages 46–57, 2000.

[30] B. Rocha, S. Bandhakavi, J. den Hartog, W. Winsborough, and S. Etalle. Towards static flow-based declassification for legacy and untrusted programs. In *IEEE Symp. on Security and Privacy*, pages 93–108, 2010.

[31] B. Rocha, M. Conti, S. Etalle, and B. Crispo. Hybrid static-runtime information flow and declassification enforcement. *Information Forensics and Security, IEEE Transactions on*, 8(8):1294–1305, 2013.

[32] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical fine-grained decentralized information flow control. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2009.

[33] J. Rushby. Noninterference, transitivity and channel-control security policies. Technical Report CSL-92-02, SRI, Dec. 1992.

[34] A. Sabelfeld and A. C. Myers. A model for delimited release. In *2003 International Symposium on Software Security*, number 3233 in Lecture Notes in Computer Science, pages 174–191. Springer-Verlag, 2004.

[35] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17(5):517–548, Oct. 2009.

[36] F. B. Schneider, K. Walsh, and E. G. Sirer. Nexus Authorization Logic (NAL): Design rationale and applications. *ACM Trans. Inf. Syst. Secur.*, 14(1):8:1–8:28, June 2011.

[37] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, September 2011.

[38] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[39] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006.

[40] L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2–3), Mar. 2007.