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

February 12, 2016

Abstract

A *reactive information flow* (RIF) automaton for a value v specifies (i) allowed uses for v and (ii) the RIF automaton for any value that might be directly or indirectly derived from v. RIF automata thus specify how transforming a value alters how the result might be used. Such labels are more expressive than existing approaches for controlling downgrading. We devised a type system around RIF automata and incorporated it into Jif, a dialect of Java that supports a classic form of labels for information flow. By implementing a compiler for the resulting JRIF language, we demonstrate how easy it is to replace a classic information-flow type system by a more expressive RIF-based type system. We programmed two example applications in JRIF, and we discuss insights they provide into the benefits of RIF-based security labels.

*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.

1 Introduction

Static enforcement of information flow policies is attractive because it helps programmers to build software that is secure, catching security-critical errors at compile time. Such language-based enforcement mechanisms (e.g., [8, 27, 35, 40]) work by enriching traditional type systems with information flow labels; the labels specify confidentiality policies and sometimes integrity policies, too.

During program execution, information contained in inputs is transformed by operations in the code. The restrictions that programmers wish to impose on how the outputs of an operation are used may differ from the restrictions on the inputs to that operation. Consider a program that tallies votes for an election. Each vote would be considered confidential, but the winner is public. So, here, fewer restrictions are imposed on the output than on the inputs. As another example, consider a conference management application. The list of reviewers and the list of papers is likely to be considered public information, but the identity of each paper's reviewers should be kept confidential. In this case, more restrictions are imposed on the output of the operation that matches papers to reviewers than the restrictions imposed on its input.

Previous work has proposed techniques to distinguish restrictions on outputs from restrictions on inputs: information flow locks [6, 8], explicit expressions for declassification (for confidentiality) and endorsement (for integrity) [27, 29], capability-based mechanisms for downgrading security policies [23, 33, 41]. These approaches are unsatisfying, though, because changes to restrictions are not connected to the restrictions on the input or to the operation performed on that input. So in this prior work, restrictions can be replaced in arbitrary ways, regardless of whether some intended operation was performed. For example, an explicit declassification expression allows the label of an expression to be replaced with a new label in much the same way that a type-cast operation changes an expression from having one type to having another. Like a type-cast, explicit declassification operators undermine the static guarantees offered by type checking. Moreover, errors introduced by a misused declassification operation can be far more subtle than what an erroneous type-cast might cause.

This paper explores information flow labels that not only specify restrictions on allowed uses of a value but also specify restrictions on allowed uses of derived values. Reactive Information Flow automata (RIF automata) [22]are automata whose states represent restrictions and whose transitions are triggered by opera-

tions. RIF automata specify arbitrary changes to restrictions on allowed uses of values, and these changes are associated with transformations to that value. In particular, RIF automata specify how restrictions are transformed in step with how the information they protect is transformed. Thus, RIF automata make explicit the connection between information transformations and changes to restrictions.

We are not the first to contemplate information flow labels that encode allowable changes to restrictions. Previous approaches (e.g., [18, 25, 31, 32, 37]) that specified permitted changes to restrictions within security labels address transformations only between two kinds of restrictions. Most specify which operations may transform secret information into public information, although the opposite transformation (from public to secret) has also been studied [14]. In [14], transformations are triggered by run-time events instead of by applied operations. In contrast to these prior proposals, RIF automata specify transformations of restrictions across a wide spectrum and do so in terms of any specified operations for transforming protected data.

To better understand the utility of RIF automata, we implemented JRIF, a new dialect of Java for supporting information flow control. JRIF is based on the Jif [27, 29] compiler and runtime. The modifications to Jif were straightforward. Jif's labels, which are based on the Decentralized Label Model [28], were substituted with RIF automata, and the restrictiveness relation on labels was modified accordingly. Our experience in building JRIF gives confidence that other languages for information flow control could be extended similarly. JRIF has also provided us with a tool to explore how the expressiveness afforded by RIF automata compares to ordinary labels. A public release of the source code for the JRIF compiler and runtime, along with example applications is available at the JRIF web page [21].

We proceed as follows. Section 2 defines RIF automata. In Section 3, we present JRIF and discuss its support for dynamic labels, parameterized types, and method constraints. Section 4 demonstrates the practicality of JRIF by describing two real programs (a Battleship game and a shared calendar application), and explores the advantages of JRIF compared to the labels in Jif. The implementation of JRIF is outlined in Section 5, and the security property that JRIF programs are expected to enforce is described in Section 6. Section 7 discusses how JRIF could be extended to support robustness. Section 8 gives comparisons of JRIF to related work, including other language-based models for controlling declassification and endorsement. Section 9 considers the relationship between reclassifiers and the

operators they annotate, and Section 10 concludes.

2 RIF Automata

A RIF automaton specifies restrictions on uses of values, and how the restrictions vary according to the history of operations involved in deriving these values.

- For confidentiality, restrictions specify which principals are allowed to read values.
- For integrity, restrictions specify whether values may be considered trusted based on which principals might have influenced them. Principals should be trusted in order for the values they modify to be trusted.

Different operations may have equivalent effects on information in their inputs. We assume that operations of interest to a programmer are annotated with identifiers that indicate 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. We employ the annotation $\texttt{reclassify}(e, \texttt{F})$ to associate a reclassifier `F` with an expression e. For example, in

$$\texttt{z} = \texttt{reclassify}(\texttt{x mod y}, \texttt{F})$$

reclassifier `F` identifies a class to which the `mod` operation belongs.

A RIF automaton is a finite-state automaton whose states are mapped to sets of principals and whose transitions are associated with reclassifiers. The meaning of a RIF automaton can be given as a function from sequences of reclassifiers to sets of principals (which correspond to the state reached after corresponding transitions are taken by processing the sequence of reclassifiers). A RIF automaton for confidentiality is called a *c-automaton*; for integrity, an *i-automaton*. To specify both confidentiality and integrity, a value is tagged with a ***RIF label***, which is a pair comprising a c-automaton and an i-automaton.

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

- Q is a finite set of automaton 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 automaton state $q_0 \in Q$, and

– $Prins$ is a function from states to sets of principals.

Finite-state automata compactly represent certain mappings from a possibly infinite number of sequences of reclassifiers to sets of principals. In theory, 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. By requiring transition function δ to be total, any sequence of reclassifiers is a valid sequence of transitions. Reclassifiers that do not trigger a transition between states (i.e., self-transitions) need not be specified explicitly, permitting compact representations of δ, as we illustrate in Section 3.

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 mapped to a superset of the principals mapped by its starting state. A reclassifier triggers a *classification* when it causes a transition whose ending state is mapped to a subset of the principals mapped by its starting state.

– For integrity, transitioning to a superset of principals triggers a *deprecation* (since a superset must now be trusted) whereas transitioning to a subset triggers an *endorsement* (because only a subset must be trusted).

– We use the term *reclassification* to describe all other relationships between starting and ending states.

The terminology introduced above is summarized in Figure 1.

We now illustrate with two simple examples how RIF automata express interesting information flow policies. Focusing on confidentiality, consider a system for paper reviewing. For each paper, three referees submit integer review scores. The system logs each referee's name with her submitted review score for a given paper. At the end, the system accepts the paper if the average review score is higher than some threshold `AcptThreshold`. A sensible confidentiality policy for each review score would be that (i) each review score can be read by the paper's authors and the referee, (ii) the pair matching a referee to her review score

	$S \subset E$	$S \supset E$
Confidentiality	Declassification	Classification
Integrity	Deprecation	Endorsement

Figure 1: Terminology for reclassification based on the relation between the sets of principals mapped by the starting (S) and ending (E) state of the corresponding transition.

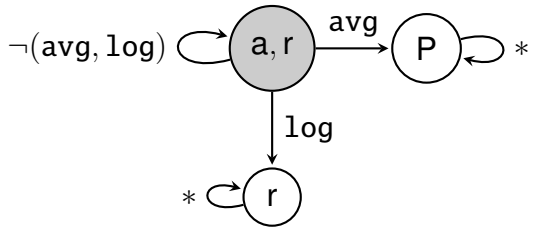


Figure 2: A c-automaton for a review score. Submitted review scores can be read by author a and referee r. The result of logging (log) each referee's score can be read only by that referee, and the final accept/reject result (avg) can be read by every principal.

may be read by the referee but not by the author (review scores are thus anonymous) (iii) the paper's final accept/reject result can be read by everyone. Notice that values covered by (i)–(iii) all derive at least partially from review scores. Figure 2 illustrates the corresponding c-automaton for review scores. Here, r denotes the referee, a a specific author, and P represents *public*, a set containing every principal. Reclassifier avg triggers a declassification, as specified by (i) and (iii), and reclassifier log triggers a classification, as specified by (i) and (ii). The asterisk "$*$" matches all reclassifiers and "$\neg$(avg, log)" matches all reclassifiers except for avg and log. Finally, gray indicates the initial state of the RIF automaton.

A program that implements the above system for paper reviewing is presented in Figure 3. Operation `concat(id,review)`, which matches a referee to her review score, is annotated with reclassifier log, because it triggers the classification specified by (i) and (ii). Operation `partSum/num > AcptThreshold`, which yields the final accept/reject result, is annotated with reclassifier avg, because it triggers the classification specified by (i) and (iii). Notice that annotated operations may appear both as assigned expressions and as conditional expressions.

```
partSum = 0;
num = 0;
while (num < 3){
  num = num +1;
  id = refereeId(num);
  review = refereeReview(num);
  record = reclassify(concat(id,review),log);
  partSum = partSum + review;
}
if (reclassify( partSum/num > AcptThreshold, avg))
  result = ‘‘accept’’;
else result = ‘‘reject’’;
```

Figure 3: Paper reviewing

As a second example[1], consider a publisher p receiving a document doc that everyone trusts. Suppose p has at her disposal two classes of operations. The first class, which is identified by reclassifier `publicize`, contains operations related to publishing documents. The second class, which is identified by reclassifier `Xcrpt`, contains operations for excerpting documents (e.g., `substring` operations). The publisher is expected to publish doc in its entirety, and so should not only publish excerpts. The i-automaton in Figure 4 illustrates the desired integrity label for doc.[2] Initially, no principal must be trusted for doc to be trusted; but the result of p excerpting doc is only as trusted as p (because p may be biasing the overall message by what p chooses for the excerpt). Here, reclassifier `Xcrpt` triggers the 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.

3 JRIF

Jif [27, 29] extends Java's types to incorporate information flow labels.[3] Jif handles information flows caused by features of Java, including exception handling

[1] This example is inspired by `TruDocs` [39].

[2] The transition function δ for this i-automaton is total, because we consider only two reclassifiers.

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

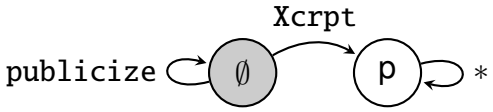


Figure 4: An i-automaton of the document doc. When the excerpt operation (annotated with Xcrpt) is applied, the result is deprecated to p.

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 a limited form of dependent types, which permits dynamic labels to be used for type checking. The Jif compiler's solver automatically infers labels for 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 JRIF implementation preserves almost all of the abstractions provided by Jif, including label parameters, dependent label types, and label inference. So, programmers can tag fields, variables, and method signatures with RIF labels, and the JRIF compiler checks whether the program satisfies these RIF labels.

3.1 Syntax of RIF Labels and Annotated Expressions

The JRIF syntax[4] of a RIF label ρ is given in Figure 5. The set of all principals P is represented by $\{\_\}$, and the empty set is represented by $\{\}$. Reclassifications that are not given explicitly in a RIF label are taken to be transitions whose starting and ending states are identical.

Figure 6 illustrates how the c-automaton in Figure 2 is coded using JRIF syntax. The initial state, s, is distinguished by an asterisk * and maps to principals a and r. State t maps to the set of all principals P, denoted by _, and state u maps to principal r. Reclassifier avg triggers a transition from s to t, and the reclassifier log triggers a transition from s to u.

In JRIF, the process of annotating expressions with reclassifiers (writing **reclassify**$(e, \mathtt{F})$) is relatively simple, and the type checker quickly detects mismatched assumptions between RIF labels and reclassifiers in annotated expres-

[4]For clarity, the syntax presented in this paper is slightly simplifies the syntax used in JRIF implementation.

$$
\begin{array}{lcl}
\rho & ::= & \{\rho_c; \rho_i\} \\
\rho_c & ::= & \texttt{c}\ [\mathit{ListOfTerms}] \\
\rho_i & ::= & \texttt{i}\ [\mathit{ListOfTerms}] \\
\mathit{ListOfTerms} & ::= & T \mid T, \mathit{ListOfTerms} \\
T & ::= & \mathit{State} \mid \mathit{InitialState} \mid \mathit{Transition} \\
\mathit{State} & ::= & \mathit{ID} : \{\mathit{ListOfPrincipals}\} \\
\mathit{InitialState} & ::= & \mathit{ID}* : \{\mathit{ListOfPrincipals}\} \\
\mathit{Transition} & ::= & \mathit{ID} : \mathit{ID} \rightarrow \mathit{ID}
\end{array}
$$

Figure 5: Syntax for labels, where *ID* represents an alphanumeric string.

$$\texttt{c}[\texttt{s}*:\{\texttt{a},\texttt{r}\}, \texttt{t}:\{\_\}, \texttt{u}:\{\texttt{r}\}, \texttt{avg}:\texttt{s} \rightarrow \texttt{t}, \texttt{log}:\texttt{s} \rightarrow \texttt{u}]$$

Figure 6: Syntactic representation of a c-automaton

sions. Expressions not explicitly annotated trigger no transition on RIF labels.

```
boolean{c[q0*:{_}]} check (int{c[q0*:{_}]} in,
        int{c[q1*:{p},q2:{_},C:q1→q2]} pwd)
{
    boolean{c[q0*:{_}]} res=false;
    if (reclassify(in==pwd,C))
        res=true;
    return res;
}
```

Figure 7: Password check

A simple method for password checking written in JRIF is shown in Figure 7. Here, method `check` takes as arguments an input `in` and a password `pwd`; it checks if these two arguments are equal. Both arguments are integers (`int`), but they are tagged with different c-automata.[5] Input `in` is, for simplicity, considered public (all principals can read it). Password `pwd` can initially be read only by principal

[5]We focus only on confidentiality for this example.

`p` (the principal that picked this password), but the result of applying the equality check (annotated with reclassifier `C`) on `pwd` is public. Method `check` returns the boolean value that results from this equality check, which is considered public. JRIF's compiler decides whether this method is safe, based on typing rules we discuss next.

Changes to confidentiality and integrity specified in RIF labels are not expressible by Jif labels—instead, additional Jif code is required. 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 a change in both confidentiality and integrity; Jif requires a separate `declassify` and `endorse` to effect that same change.

Compared to Jif, JRIF better separates program logic from information flow policies. Suppose, for example, that a programmer decides that some input value—a game player's name—should not be declassified when formerly it was.

- In JRIF, this change to the program involves modifying the RIF label declaration on any field storing the player's name. The c-automaton of the label would be inspected and edited so that it contains no transitions to automaton states that map to additional principals.

- To accommodate this change in Jif, the programmer must not only find and remove all declassification commands that involve the name field explicitly, but she also must remove all declassification commands that involve any expressions to which the game player's name flows. Getting these deletions right is error prone, since the programmer must reason about the flow of information in the code—something the type system was supposed to do.

Consequently, changes in information flow policies cause fewer changes in JRIF programs than those caused in JIf programs.

3.2 Label checking

Label checking in JRIF is performed by a procedure that decides whether the restrictions imposed by one RIF label are not weaker than the restrictions imposed by another RIF label. This is a *restrictiveness relation* between RIF labels, and it is analogous to the subtyping relation in ordinary type systems. Whenever a value

will be stored into a variable, the RIF label that tags this variable must be at least as restrictive as the RIF label on the value because, otherwise, the restrictions imposed by the value's RIF label might be violated (e.g., more principals may read values than those allowed by that RIF label) as execution proceeds.

We formalize the restrictiveness relation first for RIF automata and then for RIF labels. Let R map each RIF automaton to the set of principals mapped by its initial state[6], and let T map a RIF automaton and a *sequence* $\vec{\mathtt{F}}$ *of reclassifiers* to the RIF automaton obtained by taking the corresponding sequence of transitions.[7] For c-automata, we define ρ'_c to be at least as restrictive as ρ_c, denoted $\rho_c \sqsubseteq_c \rho'_c$, if for all possible sequences of reclassifiers, principals allowed to read the resulting value according to ρ'_c are also allowed by ρ_c. Relation $\sqsubseteq_c$ is thus formally defined as follows:

$$\rho_c \sqsubseteq_c \rho'_c \;\triangleq\; (\forall \vec{\mathtt{F}}\colon \mathtt{R}(\mathtt{T}(\rho_c, \vec{\mathtt{F}})) \supseteq \mathtt{R}(\mathtt{T}(\rho'_c, \vec{\mathtt{F}}))). \tag{1}$$

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

$$\rho_i \sqsubseteq_i \rho'_i \;\triangleq\; (\forall \vec{\mathtt{F}}\colon \mathtt{R}(\mathtt{T}(\rho_i, \vec{\mathtt{F}})) \subseteq \mathtt{R}(\mathtt{T}(\rho'_i, \vec{\mathtt{F}}))). \tag{2}$$

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

$$\{\rho_c; \rho_i\} \sqsubseteq \{\rho'_c; \rho'_i\} \triangleq (\rho_c \sqsubseteq_c \rho'_c) \;\wedge\; (\rho_i \sqsubseteq_i \rho'_i).$$

The least restrictive RIF label is denoted with $\{\}$; it allows all principals to read values, and it requires no principal to be trusted. RIF label $\{\rho_c\}$ imposes restrictions on confidentiality (according to ρ_c), but it imposes no restriction on integrity (no principal is required to be trusted). Similarly, RIF label $\{\rho_i\}$ imposes restrictions on integrity, but it imposes no restriction on confidentiality.

The RIF labels inferred by the JRIF compiler for an expression are at least as restrictive as the RIF labels of all variables in this expression. In particular, the c-automaton of an expression allows principals to read derived values only if these principals are allowed to do so by all c-automata of variables in that expression. JRIF constructs such a c-automaton by taking the product of all c-automata of

[6] $\mathtt{R}(\langle Q, \Sigma, \delta, q_0, \mathit{Prins}\rangle) \triangleq \mathit{Prins}(q_0)$

[7] $\mathtt{T}(\langle Q, \Sigma, \delta, q_0, \mathit{Prins}\rangle, \vec{\mathtt{F}}) = \langle Q, \Sigma, \delta, \delta^*(q_0, \vec{\mathtt{F}}), \mathit{Prins}\rangle$, where δ^* is the transitive closure of δ.

the used variables, assigning the intersection of the allowed principals at each state. For integrity, the i-automaton of an expression requires principals to be trusted whenever these principals are required to be trusted by some i-automata of variables in that expression. Again, JRIF constructs such an i-automaton by taking the product of all i-automata of the used variables, assigning the union of the required principals at each state.

The RIF label of an annotated expression $\textbf{reclassify}(e, \texttt{F})$ is the RIF label of expression e after performing an `F` transition. Specifically, if $\rho = \{\rho_c; \rho_i\}$ is the RIF label of e, then $\texttt{T}(\rho, \texttt{F}) \triangleq \{\texttt{T}(\rho_c, \texttt{F}); \texttt{T}(\rho_i, \texttt{F})\}$ is the RIF label of $\textbf{reclassify}(e, \texttt{F})$. This rather simple rule is what gives RIF labels their expressive power.

Information flows can be explicit or implicit. An *explicit flow* occurs when information flows from one variable to another due to an assignment

$$\texttt{x} = \textbf{reclassify}(e, \texttt{F}). \tag{3}$$

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

$$\begin{aligned} &\textbf{if } (\textbf{reclassify}(e, \texttt{F})) \ \texttt{x} = \textbf{reclassify}(e', \texttt{F}') \\ &\quad \textbf{else } \texttt{x} = \textbf{reclassify}(e'', \texttt{F}''). \end{aligned} \tag{4}$$

Knowing the value of `x` after statement (4) completes will tell whether 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 (3) is secure if both the `pc` label and the RIF label of $\textbf{reclassify}(e, \texttt{F})$ can flow to the RIF label of `x`. In other words, the RIF label of `x` is at least as restrictive as the `pc` label and the RIF label of $\textbf{reclassify}(e, \texttt{F})$. When control flow branches, as in (4), the `pc` label is increased to being at least as restrictive as the current `pc` label and the RIF label of $\textbf{reclassify}(e, \texttt{F})$. This increase ensures that assignments in either branch are constrained to variables with RIF labels at least as restrictive as the RIF label of $\textbf{reclassify}(e, \texttt{F})$.

JRIF implements label checking rules for all basic Java features, including method overloading, class inheritance, and exceptions. The formal description for all rules implemented in JRIF is out of scope for this paper. However, the ideas

that underlie these rules (i.e., restrictiveness relation, explicit and implicit flow control) are based on the rules just explained for explicit and implicit flows.

We illustrate label checking by returning to method `check` from Figure 7. This method successfully compiles in JRIF, because:

- the c-automaton of `res` is at least as restrictive as the c-automata of `in` and `pwd` after their taking a `C` transition, and
- the c-automaton of the return value is at least as restrictive as the c-automaton of `res`.

More JRIF examples can be found on the JRIF web page [21].

3.3 Dynamic labels and label parameters

Sometimes an information flow specification becomes known only at execution time. Or we might desire to reuse the same code in connection with 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 in JRIF. So, 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 in JRIF 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 into code the checks that would prevent unsafe flows at runtime. JRIF programmers insert comparisons of the restrictiveness of dynamic RIF labels at runtime by writing 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

$$\texttt{y} = \textbf{reclassify}(\texttt{x}\ \texttt{mod}\ 4, \texttt{F}) \tag{5}$$

where x is tagged with dynamic label l1, and y is tagged with dynamic label l2. This assignment statement is secure only when $\mathtt{T}(\mathtt{l1}, \mathtt{F}) \sqsubseteq \mathtt{l2}$ holds, which depends on the values of l1 and l2.

To ensure that $\mathtt{T}(\mathtt{l1}, \mathtt{F}) \sqsubseteq \mathtt{l2}$ holds, the programmer would code

$$\textbf{if } (\mathtt{T}(\mathtt{l1}, \mathtt{F}) \sqsubseteq \mathtt{l2})\ \mathtt{y} = \textbf{reclassify}(\mathtt{x}\ \mathtt{mod}\ 4, \mathtt{F}) \tag{6}$$

At compile time, constraint $\mathtt{T}(\mathtt{l1}, \mathtt{F}) \sqsubseteq \mathtt{l2}$ informs the type system about the necessary relationship between l1 and l2, because the type system may assume $\mathtt{T}(\mathtt{l1}, \mathtt{F}) \sqsubseteq \mathtt{l2}$ 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 RIF label that results from an F transition on l1 and checks whether l2 is at least as restrictive. This check also illustrates an interesting property of RIF labels: the same reclassifier may have different effects on different labels. Figure 8 shows a snippet in JRIF that uses command (6) to ensure that the modulo of x is assigned to y only if the effect of the F reclassifier on label l1 permits such a flow. Otherwise, it returns an error value. If the programmer were to omit this check, method mod4 would no longer typecheck.

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

$$\mathtt{y} = \textbf{declassify}(\mathtt{x}\ \mathtt{mod}\ 4, \mathtt{l2})$$

causes any value stored in x to be declassified.[8] To achieve the same effect as (6) in Jif, the programmer would have to use an additional variable, say d, to control whether x mod 4 should be declassified.

$$\textbf{if } (\mathtt{d})\ \mathtt{y} = \textbf{declassify}(\mathtt{x}\ \mathtt{mod}\ 4, \mathtt{l2}) \tag{7}$$

But it now becomes the Jif programmer's responsibility to ensure that d is inspected before any declassification of x; the Jif type system provides no assistance about that. This program construction is error-prone, compared with (6), 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 when 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

[8] Robust downgrading places restrictions on the label, but that is orthogonal to this discussion.

```
int{l2} mod4(label{} l1, label{} l2, int{l1} x)
{

    int{l2} y=-1;
    if (T(l1,F) ⊑ l2) y=reclassify(x mod 4,F);
    return y;
}

void foo()
{
    label{} l1 =
     new label {c[q1*:{a},q2:{_},F:q1→q2];
                i[q1*:{a},q2:{},F:q1→q2]};
    label{} l2 =
     new label {c[q1*:{_}]; i[q1*:{}]};
    int{l1} x = Random();
    int{l2} z;

    z = mod4 (l1,l2,x);
}
```

Figure 8: Dynamic labels and dynamic check

method or class to be secure. This, in turn, reduces the number of classes that must be modified if RIF labels must be changed to accommodate new functionality.

3.4 Constraints

To reduce the need for redundant dynamic checks, a JRIF programmer may declare relationships between labels that must hold when a specific method is called. These relationships are called **where**-*constraints*. These constraints already exist in Jif, but they are being extended here to accommodate the expressiveness of RIF labels. Consider method

$$\begin{array}{l}\texttt{int}\{\texttt{l2}\}\ \texttt{mod4}(\{\texttt{l1}\}\ \texttt{x})\ \texttt{where}\{\texttt{T}(\texttt{l1},\texttt{F}) \sqsubseteq \texttt{l2}\}\{\\ \quad \texttt{return reclassify}(\texttt{x mod } 4,\texttt{F});\\ \}\end{array}$$

for l1 and l2 dynamic labels or label parameters. Constraint $\texttt{where}\{\texttt{T}(\texttt{l1},\texttt{F}) \sqsubseteq \texttt{l2}\}$ permits the type system to assume $\texttt{T}(\texttt{l1},\texttt{F}) \sqsubseteq \texttt{l2}$ while type checking the body of method mod4. To ensure mod4 executes only when $\texttt{T}(\texttt{l1},\texttt{F}) \sqsubseteq \texttt{l2}$ holds, the type system need only check that the **where**-constraint holds at every call site naming mod4. For instance, the expression $\texttt{reclassify}(\texttt{x mod } 4,\texttt{F})$ in (6) could be replaced by a call to method mod4.

4 Program Examples using JRIF

4.1 Battleship

The Jif distribution [29] includes an implementation of the Battleship game. Battleship is a good example because both confidentiality and integrity are important to prevent cheating. Over the course of the game, confidential information is declassified. 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 initial placements.

A rather simple c-automaton suffices to specify the confidentiality policy for the ship-coordinates of each player. That policy is:

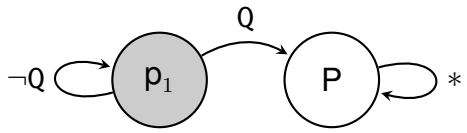


Figure 9: A c-automaton for ship-coordinates.

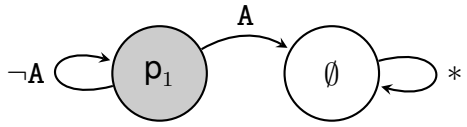


Figure 10: An i-automaton for ship-coordinates.

– Values derived from 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 9, 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. 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. So, before the game actually starts, there is a game operation whose reclassifier raises the integrity of all ship-coordinates ensuring that neither player can make changes. An i-automaton that expresses this policy is presented in Figure 10, where `A` is the reclassifier annotating the operation that accepts the initial coordinates.

The JRIF implementation of Battleship was obtained by making just a few modifications to the Jif implementation. We replaced Jif labels with RIF labels, and we replaced various Jif declassification or endorsement commands with JRIF reclassifications. Figures 11 and 12 illustrate some differences and similarities between programming in JRIF and in Jif. Methods in the Jif implementation that involved only label parameters and dynamic labels could be used without any modification in the JRIF implementation. We believe that any program written in

Jif can be easily ported to JRIF. The JRIF source for the Battleship implementation is found on JRIF's web page [21], along with the original Jif source (for comparison).

```
boolean{c[q0*:{_}];i[q1*:{}]} processQuery
(Coordinate[{i[q1*:{}]}]{i[q1*:{}]} query)
{
  Board[{c[q0*:{P},q1:{_},Q:q0→q1];i[q1*:{}]}]
   brd = this.board;
  List[{i[q1*:{}]}] oppQueries =
   this.opponentQueries;
  oppQueries.add(query);
  boolean result = brd.testPosition(query);
  return reclassify(result,Q);
}
```

Figure 11: Method `processQuery` from JRIF implementation. It checks the success of opponent's hit

```
boolean{P<-* meet O<-*} processQuery
 (Coordinate[{P<-* meet O<-*}]{P<-* meet O<-*} query)
{
  Board[{P->*; P<-* meet O<-*}] brd = this.board;
  List[{P<-* meet O<-*}] oppQueries =
   this.opponentQueries;
  oppQueries.add(query);
  boolean result = brd.testPosition(query);
  return declassify(result,{P->*;P<-* meet O<-*}
                    to {P<-* meet O<-*});
}
```

Figure 12: Method `processQuery` from JIF implementation. It checks the success of opponent's hit

The compilation and execution time of the JRIF version of Battleship is 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 (The Jif version is ~ 615 LOC, and JRIF version is ~ 628 LOC). To measure 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 in JRIF execution time is mostly due to the overhead of creating dynamic labels in JRIF, which currently requires more method calls than Jif's dynamic labels do.

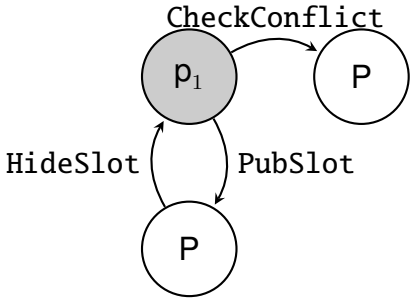
4.2 A Shared Calendar

To further explore the expressive power of RIF labels, we developed a shared calendar application from scratch in JRIF. This application allows users to create and share events in calendars. Each event consists of the fields: time, date, duration, and description. Declassification, classification, endorsement, and deprecation all appear 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 labels.

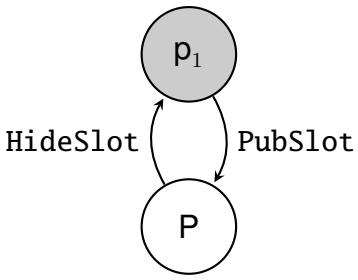
Operations supported by our shared calendar include:

- Create a *personal* event or a *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`
- Check and announce a conflict between personal events (not shared or canceled events) and an invitation for a new shared event.
  Reclassifier: `CheckConflict`
- Publish an event date and time (but not the event description). Reclassifier: `PubSlot`
- Hide an event date and time. Reclassifier: `HideSlot`

The reclassifiers that annotate these operations change the confidentiality and integrity of events. Once an event is accepted (`Accept` is applied), the resulting shared event is given the highest integrity, since all of the attendees endorse it. Having the highest integrity implies that no attendee is able to modify this shared event, thereafter. If an event is cancelled (`Cancel` is applied), then this event is



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



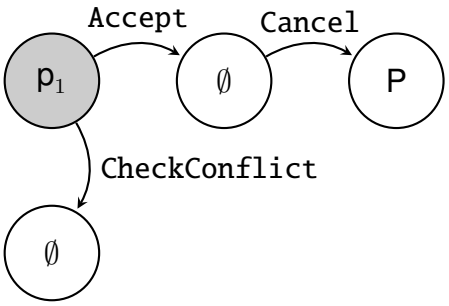
(b) A c-automaton that does not permit declassification for conflict-checking.

Figure 13: RIF automata for event confidentiality. Self-loops are omitted for clarity.

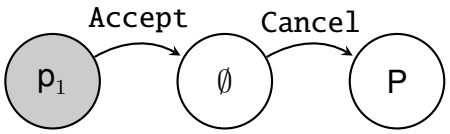
given the lowest integrity, as are all values that may be derived from it by applying supported operations. With lowest integrity, cancelled events and all values derived from them can be distinguished. If `CheckConflict` is applied to a personal event and an invitation for a new shared event, then the result gets the lowest confidentiality and the highest integrity. This is because the result is readable and trusted by all principals that learn about the conflict. If `PubSlot` is applied to an event, then the event's date and time can flow to all principals, until a `HideSlot` is subsequently applied to that event.

Figure 13 illustrates c-automata for events created by a principal p_1. The c-automaton in Figure 13a permits a full declassification triggered by reclassifier `CheckConflict`; the c-automaton in Figure 13b does not. Both c-automata specify a declassification under `PubSlot`, and a classification under `HideSlot`. Figure 14 gives corresponding i-automata for the events of p_1. The i-automaton in Figure 14a permits a full endorsement triggered by reclassifier `CheckConflict`; the i-automaton in Figure 14b does not. Both i-automata specify an endorsement under `Accept`, and a deprecation under `Cancel`. Notice that `CheckConflict` triggers transitions in both a c-automaton and an i-automaton, contrary to, say, `PubSlot`. Source code for this shared calendar implementation in JRIF can be found on JRIF's web page [21].

We use dynamic labels and label parameters extensively in the shared calendar application. The event class has label parameters that are typically instanti-



(a) An i-automaton that permits endorsement for conflict checking.



(b) An i-automaton that does not permit endorsement for conflict-checking.

Figure 14: RIF automata for event integrity. Self-loops are omitted; for instance, the result of applying *CheckConflict* to a canceled event has low integrity.

```
if (T(lEvt,C) ⊑ {c[q0*:{_}];i[q1*:{}]}
    && T(lCal,C) ⊑ {c[q0*:{_}];i[q1*:{}]}){
  if (reclassify(cal.hasConflict(e,lEvt,lCal),C)
    result = true;       // Conflict detected
  else result = false;   // No conflict
}
```

Figure 15: Checking if the conflict is allowed to be declassified and endorsed, where `C` corresponds to reclassifier `CheckConflict`. Here, `lEvt` is the dynamic label of the requested shared event `e`, `lCal` 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.

ated by dynamic labels. These RIF labels tag the components of events (time, date, duration, and description). Since events have label parameters, methods that manipulate events make use of label parameters and dynamic labels in their type declarations. Figure 15 excerpts from the conflict-checking method. Here, the label of the event is dynamically checked to see whether it permits the conflict check to be declassified and endorsed before performing the corresponding operation.

A user's events may be tagged with different dynamic labels. For example, a user might pick the c-automaton in Figure 13a for some events but pick the c-automaton in Figure 13b for others. Events can have different i-automata, too. An unshared event has one of the i-automata in Figure 14, but an accepted event can be treated with higher integrity and thus tagged with the i-automaton denoted by taking the `Accept` transition. In addition, 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 [43].

5 Building a JRIF Compiler

We built a JRIF compiler by modifying the existing Jif compiler in relatively straightforward ways. Applying similar modifications to compilers for other information flow languages ought to be straightforward. This should not be so surprising: RIF labels expose the same interface to a type system as native information flow 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 stipulating that this relation is monotone with respect to transition relation `T`,

Item (3b) is essential for supporting our richer language of label comparisons and **where**-constraints. For example, if a programmer introduces **where**-constraint $\texttt{l2} \sqsubseteq \texttt{l1}$ (or equivalently, checks dynamically that this relation holds), then the type checker should be able to deduce that $\texttt{T}(\texttt{l2}, \texttt{F}) \sqsubseteq \texttt{T}(\texttt{l1}, \texttt{F})$ holds for every F.

We decided to build JRIF by extending the Jif compiler because Jif is a long-standing, widespread language for information flow control. JRIF adds 6k lines of code to Jif (which contains 230k LOC). Out of the 494 Java classes used in Jif, we modified only 31 and added 48 new classes for JRIF. 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. The source code for JRIF can be found on JRIF's web page [21].

Some features of Jif are orthogonal to enforcing RIF labels, and JRIF ignores 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 labels. As discussed below in §7, future versions of JRIF could extend the type system to support these constraints on downgrading as a complement to security guarantees offered by RIF labels.

6 What JRIF Label Checking Enforces

Label checking in information flow control systems usually enforces noninterference [16] or some variation. For confidentiality, noninterference stipulates that changes to values that principal p cannot read initially should not cause changes to values that p can read during program execution. Equivalently, if the initial states of two program executions agree on values that principal p can read, then each successive state during program execution should agree on values that p can read. For integrity, noninterference requires values that initially depend on trusting p do not cause changes to values whose trust does not depend on p. Moreover, these conditions must hold for each principal p.

Reclassifications complicate a definition of permissible 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 would be violated. To accommodate reclassification in defining permissible information flow, it suffices to partition execution into segments that each satisfy noninterference. Segments are delimited by reclassifications, relative to a single principal p.

> **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 any 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 any newly endorsed values, then these pieces agree on the values that get assigned to variables whose trust does not depend on p throughout execution.

We have proven that RIF automata enforce PWNI for a simple imperative language [22], giving us confidence in the formal guarantees enforced by the JRIF type system.[9] Many models [2, 3, 19, 24, 36] have been proposed for expressing and enforcing policies that permit changing the restrictions imposed on the use of values, but PWNI is the first to handle both classifications and deprecations.

7 RIF and Robustness

In JRIF, downgrading (e.g., declassification and endorsement) is an application's response to particular operations. A possible extension would be to also stipulate that only specific principals be allowed to change restrictions associated with these operations. This extension is essentially *robust downgrading* [13] applied

[9] Despite the similarities in choice of name, *reactive noninterference* [5] is unrelated to our reactive information flow specifications. Reactive noninterference is a definition of non-interference for reactive programs (i.e. programs that, during execution, may wait for user inputs, or produce outputs). The term "reactive" in our paper characterizes specifications, whereas in [5], it characterizes programs.

to RIF automata. Robust downgrading, used in Jif [29] and Fabric [26, 1], requires that downgrades occur only in high-integrity contexts, thereby 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 what information is declassified (in the case of declassification), JRIF guarantees that disclosed values are produced only by specified sequences of operations. Yet, RIF specifications are compatible with robust downgrading. And JRIF's type system could be extended with an enforcement mechanism to enforce RIF specifications and robustness simultaneously. There are two options.

One option is 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 enhanced expressiveness of the second approach should have interesting applications, but we leave further exploration to future work.

8 Related work

Expressive structures, like automata, have previously been used to represent information flow specifications. Program dependence graphs [17, 20], which represent data and flow dependencies between values specify allowable declassifications. And Rocha et al. [31, 32] employ *policy graphs* to specify sequences of functions that cause declassifications. However, this work does not handle arbitrary reclassifications; it only handles 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* [30], which refines this idea with policies that are owned [10] by principals who may differ in which code they trust. These systems enforce a form of *intransitive* flow policy [34] since direct flows that do not involve the declassifying operation are prohibited.

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.[10] RIF automata can be extended to express such specifications by associating reclassifiers with conditions on states formulated using 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.

Li and Zdancewic [24] formalize downgrading (i.e. declassification and endorsement) policies by using simply-typed lambda terms. Here, information flow labels are sets of lambda terms (i.e. functions); when one of these lambda terms is applied to the corresponding value, the result is downgraded. RIF automata with no cycles can be modified to express such information flow labels, by associating reclassifiers with particular sets of lambda terms, by mapping the last reachable states to a *low* label (e.g. all sets of principals for confidentiality, empty set for integrity), and by mapping all other states to a *high* label (e.g. empty set for confidentiality, all sets of principals for integrity).

Sabelfeld et al. [38] introduce a four-dimension categorization (what, where, who, when) of declassification. In JRIF, a reclassifier that causes a declassification

[10] Chong and Myers [12] also propose extension of their theory to operations other than declassification and erasure.

indicates "what" will be declassified and "where" in the program.

Broberg et al. [9] 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, a function that makes a RIF automaton take a transition based on a reclassifier is a Level 1 control (a *determining function*), a function that returns the principals in the initial state of a RIF automaton is a Level 2 control (a *meta policy*). However, whereas Broberg et al. [9] consider these controls to be the same for all values in a program, in JRIF this hierarchy is different for each RIF automaton.

8.1 Capability-based systems

Many recent systems for information flow control are based on capabilities, including Flume [23], HiStar [42], Asbestos [15], Aeolus [11], Laminar [33], and LIO [41]. We focus our discussion on Flume, but similar arguments 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, then 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 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 scheme gives the password manager more power than Alice might 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 mapped to 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.

8.2 Paragon and Paralocks

The Paragon [8] programming language has information flow control based on Paralocks [7, 6]. 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.

9 Semantics for reclassifiers

Annotating expressions with reclassifiers is the responsibility of JRIF programmers. While it might seem attractive to enforce a formal connection between reclassifiers and the operations they represent, doing so seems challenging.

Mechanically checking that expressions are annotated with the correct reclassifiers is not trivial. An initial requirement would be having a specification language for the semantics associated with each reclassifier. Such a semantics would, for example, have to express information-theoretic properties of a transformation associated with a reclassifier. Expressions in the programming language would also need to be analyzable in terms of this semantics, in order to verify the correctness of an annotation.

Even given such a specification language and program analyzer, it is still the author of a policy who specifies how reclassifiers transform restrictions on values. For example, knowing that the application of a given reclassifier only reveals 1 bit of secret input information does not necessarily imply that the output information can be considered public. For some values, it may be that a particular bit is security-critical, so the output information would still be deemed secret. Thus, the expressive power of RIF automata is still needed, to enable policy writers to specify how restrictions on values are transformed given the semantics of the applied reclassifiers.

10 Conclusion

JRIF is a extension of Java for supporting Reactive Information Flow Control based on RIF automata. RIF labels specify allowed uses of the values they are associated with, along with the RIF label to associate with derived values. The JRIF compiler was implemented as a straightforward extension of the Jif compiler and runtime, demonstrating that RIF specifications are easily incorporated into 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 invariably makes 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 the 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] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 79–90, New York, NY, USA, 2009. ACM.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] S. Chong and A. C. Myers. Decentralized robustness. In *19th IEEE Computer Security Foundations Workshop (CSFW)*, 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 (CSF)*, 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] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic updating of information-flow policies. In *Foundations of Computer Security Workshop*, 2005.

[20] 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.

[21] E. Kozyri, O. Arden, A. C. Myers, and F. B. Schneider. JRIF: Java with Reactive Information Flow. Software release, at `http://www.cs.cornell.edu/jrif`, Feb. 2016.

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

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[36] 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.

[37] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop (CSFW)*, pages 255–269, June 2005.

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

[39] 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.

[40] V. Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003.

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

[42] 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.

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