JRIF: Reactive Information Flow Control for Java

Elisavet Kozyri[1(✉)], Owen Arden[2], Andrew C. Myers[1], and Fred B. Schneider[1]

[1] Cornell University, Ithaca, USA
{ekozyri,andru,fbs}@cs.cornell.edu
[2] University of California, Santa Cruz, USA
owen@soe.ucsc.edu

Abstract. A *reactive information flow* (RIF) automaton for a value v specifies (i) restrictions on uses for v and (ii) the RIF automaton for any value that might be derived from v. RIF automata thus specify how transforming a value alters restrictions for the result. As labels, RIF automata are both expressive and intuitive vehicles for describing allowed information flows. JRIF is a dialect of Java that uses RIF automata for specifying information flow control policies. The implementation of JRIF involved replacing the information flow type system of the Jif language by a RIF-based type system. JRIF demonstrates (i) the practicality and utility of RIF automata, and (ii) the ease with which an existing information flow control system can be modified to support the expressive power of RIF automata.

Keywords: Information flow control · Reclassification · Automata

1 Introduction

Many language-based enforcement mechanisms (e.g., [4,22,29]) work by enriching traditional type systems with information flow labels; the labels specify restrictions on confidentiality and sometimes integrity, too. Restrictions that programmers wish to impose on how the outputs of an operation may be used are likely to differ from any restrictions that were imposed on inputs to that operation. Consider, for example, a program that tallies votes for an election. Each vote would be considered confidential to a single voter, whereas the majority value is made public. So 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 public, but the identity of each paper's reviewers should be kept confidential. In this case, more restrictions are imposed on the output of an operation that matches papers to reviewers than the restrictions imposed on its input.

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.

Previous work in language-based enforcement for information flow control allows restrictions on outputs to be different from the restrictions on inputs. Explicit expressions for declassification (for confidentiality) and endorsement (for integrity) [21,22], as well as capability-based mechanisms for downgrading security policies [18,27,33] are unsatisfying, though, because output restrictions are not connected to input restrictions or to the operation performed on that input. Rather, with these mechanisms, restrictions can be replaced in arbitrary ways. For example, an explicit declassification expression [21,22] allows the label on 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. Other approaches deduce output restrictions based on input restrictions and run-time events (e.g., [4,5,7]) instead of considering the operations applied in computing these inputs. Some techniques do consider operations applied during computation (e.g., [13,20,25,26,31]), but only to specify transformation of secret information into public information.

This paper introduces information flow labels that specify arbitrary changes between classes of restrictions, with changes explicitly connected to the operations that transform the labeled data. *R̲eactive I̲nformation F̲low automata* (RIF automata) [16] are automata whose states define restrictions and whose transitions are triggered by operations on a labeled value. Thus, RIF automata specify how restrictions are transformed in step with transformations to the data they protect; the connection between information transformations and changes to restrictions is, consequently, explicit.

In this paper, we explore the practicality and utility of RIF automata. We developed JRIF, a new dialect of Java for supporting reactive information flow control, to study the practicality of RIF automata for statically enforcing information flow policies. JRIF is derived from the Jif [21,22] compiler and run time, and the modifications to Jif were straightforward. Jif's labels, which are based on the Decentralized Label Model [23], were replaced by RIF automata, and Jif's 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. To illustrate the utility of RIF automata, we programmed two JRIF applications that leverage the expressive power of RIF automata: a Battleship game and a shared calendar application. A public release of the source code for the JRIF compiler and run time, along with the example applications are available at the JRIF web page [17].

We proceed as follows. Section 2 defines RIF automata. In Sect. 3, we present JRIF. Section 4 illustrates the practicality of JRIF by describing two applications. The implementation of JRIF is outlined in Sect. 5. Section 6 compares JRIF to related work, including other language-based models for controlling declassification and endorsement, and Sect. 7 concludes.

2 RIF Automata

A RIF automaton specifies restrictions on a value and how those restrictions change according to the history of operations involved in deriving that value.

- For confidentiality, the restrictions identify principals allowed to read the value.
- For integrity, the restrictions identify principals that should be trusted for the value to be trusted.

So, in both cases, restrictions are given as sets of principals.

Operations of interest to a programmer are associated with identifiers. We call these identifiers *reclassifiers*. So sequences of reclassifiers are abstract descriptions for the series of operations that were applied to values as program execution proceeds. A sequence of reclassifiers thus provides a basis for determining how the confidentiality or integrity of the output of series of operations differs from that of its inputs.

A RIF automaton is a finite-state automaton whose states map to sets of principals and whose transitions are associated with reclassifiers. Formally, a RIF automaton λ is defined to be a 5-tuple $\langle Q, \Sigma, \delta, q_0, \mathit{Prins} \rangle$, where:

- Q is a finite set of automaton states,
- Σ is a finite set of reclassifiers,
- δ is a total, deterministic transition function $Q \times \Sigma \rightarrow Q$,
- q_0 is the initial automaton state $q_0 \in Q$, and
- Prins is a function from states to sets of principals.

RIF automata compactly represent certain mappings from sequences of reclassifiers to sets of principals. In theory, the number of states in a RIF automaton could be large; in practice, relatively small RIF automata suffice for representing many policies of practical interest. By requiring transition function δ to be total, any sequence of reclassifiers induces a sequence of transitions.[1] A RIF automaton for confidentiality is called a *c-automaton*; for integrity, an *i-automaton*.

Changes to the confidentiality or integrity restrictions associated with 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 relationships between starting and ending states.

Two simple examples illustrate how RIF automata express interesting information flow policies. Focusing on confidentiality, consider a system to support

[1] Reclassifiers that do not trigger a transition between states (i.e., self-transitions) need not be specified explicitly, thereby permitting compact representation of δ, as we illustrate in Sect. 3.

paper reviewing. For each submitted paper, three referees provide integer review scores. The system logs each referee's name with her review score for each given paper. This logging operation is identified by reclassifier *log*. The system accepts the paper if three reviews have been provided and the average review score is higher than some threshold. This comparison operation is identified by reclassifier *avg*.

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 (i.e., the result of *log*) may be read by the referee but not by the author (review scores are thus anonymous), (iii) the paper's final accept/reject result (i.e., the result of *avg*) can be read by everyone.[2] Notice that values covered by (i)–(iii) all derive at least partially from review scores. Figure 1 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.

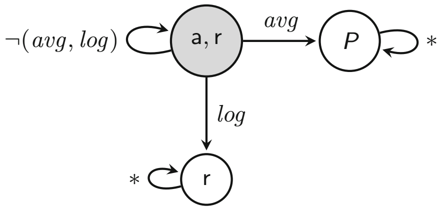

$$\begin{tikzcd}
\mathsf{a,r} \arrow[loop left, "\neg(avg,log)"] \arrow[r, "avg"] \arrow[d, "log"] & P \arrow[loop right, "*"] \\
\mathsf{r} \arrow[loop left, "*"] &
\end{tikzcd}$$

Fig. 1. 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.

A second example sketches RIF automata that enforce integrity policies for a document management system.[3] Given is an *original* document *doc* with high integrity. Empty set $\emptyset$ models high integrity (i.e., no principal must be trusted for the documents to be trusted). Operation $ext(doc, rules)$ derives a new document from document *doc* according to instructions (in *rules*) describing what text to excerpt. Because creative excerpting (e.g., omitting the text that gives conditions under which some agreement is made) can be used to generate a document that has different meaning from the original, *derived* documents have low integrity. We use set P to model low integrity (i.e., all principals must be trusted for

[2] This example thus addresses only a subset of information flow policies that a conference system needs to satisfy [15].

[3] This example is inspired by `TruDocs` [32].

the derived documents to be trusted). The i-automaton in Fig. 2 illustrates the desired integrity label for *doc*. Here, reclassifier *ext* triggers the deprecation.

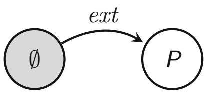


Fig. 2. An i-automaton of the document *doc*. When the excerpt operation (annotated with *ext*) is applied, the result is deprecated.

3 JRIF

JRIF (Java with Reactive Information Flow) extends Java's types to incorporate RIF automata. Programmers can tag fields, variables, and method signatures with RIF automata, and the JRIF compiler checks whether a program satisfies these RIF automata.

3.1 Syntax of JRIF

In JRIF, a *JRIF label* is a pair comprising a c-automaton λ_c and an i-automaton λ_i. The c-automaton specifies confidentiality restrictions and the i-automaton provides the independent specification for integrity restrictions. The JRIF syntax[4] of a JRIF label is given in Fig. 3. The set of all principals is represented by {_}, and the empty set is represented by {}. Reclassifications that are not given explicitly in a JRIF label are taken to be transitions whose starting and ending states are identical.

Figure 4 illustrates how the i-automaton in Fig. 2 is coded using JRIF syntax. The initial state `s` is distinguished by an asterisk `*` and maps to empty set {} of principals. State `t` maps to the set {_} of all principals. Reclassifier `ext` triggers a transition from `s` to `t`.

JRIF has the ordinary expressions of Java (e.g., constants, variables, etc.). In addition, an ordinary expression $\mathcal{E}$ can be annotated with a reclassifier `f` by writing

`reclassify(`$\mathcal{E}$`,f)`.

An annotated expression can appear wherever an ordinary Java expression can (e.g., in right-hand side expressions of assignments or in guard expressions of conditional commands). Expressions not explicitly annotated trigger no transition on JRIF labels.

A simple method for PIN (personal identification number) checking written in JRIF is shown in Fig. 5. Here, method `check` takes as arguments an integer input `in` and an integer `PIN`; it checks if these two arguments are equal. The arguments

[4] For clarity, the syntax presented in this section simplifies the syntax actually used in our JRIF implementation.

$$
\begin{array}{ll}
\lambda & ::= \{\lambda_c; \lambda_i\} \\
\lambda_c & ::= \mathtt{c}\ [\mathit{ListOfTerms}] \\
\lambda_i & ::= \mathtt{i}\ [\mathit{ListOfTerms}] \\
\mathit{ListOfTerms} & ::= T \mid T, \mathit{ListOfTerms} \\
T & ::= \mathit{State} \mid \mathit{InitialState} \mid \mathit{Transition} \\
\mathit{State} & ::= \mathit{ID} : \{\mathit{ListOfPrincipals}\} \\
\mathit{InitialState} & ::= \mathit{ID}{*} : \{\mathit{ListOfPrincipals}\} \\
\mathit{Transition} & ::= \mathit{ID} : \mathit{ID} \rightarrow \mathit{ID}
\end{array}
$$

Fig. 3. Syntax for JRIF labels, where *ID* represents an alphanumeric string.

```
i[s*:{}, t:{_}, ext:s→t]
```

Fig. 4. Syntactic representation of an *i*-automaton

are tagged with different *c*-automata.[5] Input `in` is, for simplicity, considered public (all principals can read it). `PIN` can initially be read only by principal `p` (the principal that picked this `PIN`), but the result of applying the equality check (annotated with reclassifier `C`) on `PIN` 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.

Label Checking. Label checking in JRIF is performed by a procedure that decides whether the restrictions imposed by one JRIF label are not weaker than the restrictions imposed by another JRIF label. This is a *restrictiveness relation* between JRIF labels, and it is analogous to the subtyping relation in ordinary type systems. Whenever a value will be stored into a variable, the JRIF label that tags this variable must be at least as restrictive as the JRIF label on the value because, otherwise, the restrictions imposed by the value's JRIF label might be violated (e.g., more principals may read values than those allowed by that JRIF label) as execution proceeds.

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

Fig. 5. PIN check

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

We formalize the restrictiveness relation first for RIF automata and then for JRIF labels. Let $\mathcal{R}$ map each RIF automaton to the set of principals mapped by its initial state[6], and let $\mathcal{T}$ map a RIF automaton and a sequence F of reclassifiers to the RIF automaton obtained by taking the corresponding sequence of transitions.[7]

For c-automata, define λ'_c to be at least as restrictive as λ_c, denoted $\lambda_c \sqsubseteq_c \lambda'_c$, if for each possible sequence F of reclassifiers, principals allowed to read the resulting value according to λ'_c are also allowed by λ_c. Specifically, the set of principals in the state reached by taking F transitions on λ'_c is a subset of the principals in the corresponding state of λ_c. Relation $\sqsubseteq_c$ is formally defined as follows:

$$\lambda_c \sqsubseteq_c \lambda'_c \triangleq (\forall F\colon \mathcal{R}(\mathcal{T}(\lambda_c, F)) \supseteq \mathcal{R}(\mathcal{T}(\lambda'_c, F))). \tag{1}$$

For i-automata, λ'_i is at least as restrictive as λ_i, denoted $\lambda_i \sqsubseteq_i \lambda'_i$, if for all possible sequences of reclassifiers, principals that must be trusted according to λ'_i include those that must be trusted according to λ_i. So, relation $\sqsubseteq_i$ is defined as follows:

$$\lambda_i \sqsubseteq_i \lambda'_i \triangleq (\forall F\colon \mathcal{R}(\mathcal{T}(\lambda_i, F)) \subseteq \mathcal{R}(\mathcal{T}(\lambda'_i, F))). \tag{2}$$

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

$$\{\lambda_c; \lambda_i\} \sqsubseteq \{\lambda'_c; \lambda'_i\} \triangleq (\lambda_c \sqsubseteq_c \lambda'_c) \wedge (\lambda_i \sqsubseteq_i \lambda'_i).$$

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

JRIF labels inferred by the JRIF compiler for an expression are at least as restrictive as the JRIF 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 comprising that expression. JRIF constructs such a c-automaton by taking the product of all c-automata of the referenced 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

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

[7] $\mathcal{T}(\langle Q, \Sigma, \delta, q_0, Prins\rangle, F) \triangleq \langle Q, \Sigma, \delta, \delta^*(q_0, F), Prins\rangle$. Here δ^* is the transitive closure of δ: $\delta^*(q_0, F'\mathsf{f}) \triangleq \delta(\delta^*(q_0, F'), \mathsf{f})$ and $\delta^*(q_0, \epsilon) \triangleq q_0$, where ϵ denotes the empty sequence.

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.[8]

The JRIF label of an annotated expression $\texttt{reclassify(}\mathcal{E}\texttt{,f)}$ is the JRIF label of expression $\mathcal{E}$ after performing an f transition. Specifically, if $\lambda = \{\lambda_c; \lambda_i\}$ is the JRIF label of $\mathcal{E}$, then $\mathcal{T}(\lambda, \mathsf{f}) \triangleq \{\mathcal{T}(\lambda_c, \mathsf{f}); \mathcal{T}(\lambda_i, \mathsf{f})\}$ is the JRIF label of $\texttt{reclassify(}\mathcal{E}\texttt{,f)}$. This rather simple rule is what gives JRIF labels their expressive power.

Information flows can be explicit or implicit [8,34]. An *explicit flow* occurs when information flows from one variable to another due to an assignment

$$\texttt{x} = \texttt{reclassify(}\mathcal{E}\texttt{,f)}. \tag{3}$$

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

$$\texttt{if}\ (\texttt{reclassify(}\mathcal{E}\texttt{,f)})\ \{\texttt{x} = 1\}\ \texttt{else}\ \{\texttt{x} = 2\}. \tag{4}$$

Knowing the value of `x` after statement (4) completes reveals whether $\mathcal{E}$ 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. Assignment (3) is secure if the JRIF label of `x` is at least as restrictive as both the *pc* label and the JRIF label of $\texttt{reclassify(}\mathcal{E}\texttt{,f)}$. 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 JRIF label of $\texttt{reclassify(}\mathcal{E}\texttt{,f)}$. This increase ensures that assignments in either branch are constrained to variables with JRIF labels at least as restrictive as the JRIF label of $\texttt{reclassify(}\mathcal{E}\texttt{,f)}$.

JRIF employs label checking rules for all basic Java features, including method overloading, class inheritance, and exceptions.[9] The formal description for all rules employed by JRIF is out of scope for this paper. However, the ideas that underlie these rules are based on the rules just explained for explicit and implicit flows.

We illustrate label checking by returning to method `check` from Fig. 5. This method compiles successfully in JRIF, because:

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

[8] The product of two c-automata $\lambda_c = \langle Q, \Sigma, \delta, q_0, \mathit{Prins}\rangle$ and $\lambda'_c = \langle Q', \Sigma', \delta', q'_0, \mathit{Prins}'\rangle$ is defined to be $\lambda_c \sqcup_c \lambda'_c \triangleq \langle Q \times Q', \Sigma \cup \Sigma', \delta_\times, \langle q_0, q'_0\rangle, \mathit{Prins}_\times\rangle$ where $\delta_\times$ and $\mathit{Prins}_\times$ are defined for $\langle q, q'\rangle \in Q \times Q'$ as: $\delta_\times(\langle q, q'\rangle, \mathsf{f}) \triangleq \langle \delta(q, \mathsf{f}), \delta'(q', \mathsf{f})\rangle$, and $\mathit{Prins}_\times(\langle q, q'\rangle) \triangleq \mathit{Prins}(q) \cap \mathit{Prins}'(q')$. The definition for the product of two i-automata is the same with the only difference being the definition of $\mathit{Prins}_\times$, where instead of intersection we take the union of sets. RIF automata form a lattice.

[9] Label checking rules for Java features already exist in Jif. Their core component is a call to a decision algorithm for the restrictiveness relation. So, we were able to support JRIF labels simply by substituting Jif's decision algorithm with JRIF's decision algorithm for JRIF label restrictiveness.

– 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 [17].

Dynamic Labels. Sometimes an information flow label only becomes known at run time. To accommodate this, JRIF adopts Jif's *dynamic labels.* So JRIF dynamic labels may be instantiated as run-time values, stored in variables, and compared dynamically.

Since the actual JRIF label that a dynamic label denotes is not known at compile time, the JRIF label checker requires the programmer to provide code that checks for unsafe flows at run time. For example, consider

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

where `x` has been declared to have a dynamic label `L1`, and `y` a dynamic label `L2`. This assignment statement is secure only when $\mathcal{T}(\texttt{L1}, \texttt{f}) \sqsubseteq \texttt{L2}$ holds. In JRIF, programmers can write `T(L1,f)` to represent a dynamic label whose value is $\mathcal{T}(\texttt{L1}, \texttt{f})$. So, to ensure that $\mathcal{T}(\texttt{L1}, \texttt{f}) \sqsubseteq \texttt{L2}$ holds when (5) executes, the JRIF programmer must insert a conditional test to guard (5):

$$\texttt{if (T(L1,f)} \sqsubseteq \texttt{L2)}\ \texttt{y} = \texttt{reclassify(x mod 4, f)} \tag{6}$$

– At compile time, constraint `T(L1,f)` $\sqsubseteq$ `L2` informs the type system about the necessary relationship between `L1` and `L2`, because the type system may assume `T(L1,f)` $\sqsubseteq$ `L2` holds when the "then" clause starts executing.
– At run time, the system constructs the JRIF label that results from an `f` transition on `L1` and checks whether `L2` is at least as restrictive.

This example also illustrates an interesting property of JRIF labels: the same reclassifier may have different effects on different labels. For some instantiations of `L1`, transitioning according to `f` may satisfy relation `T(L1,f)` $\sqsubseteq$ `L2`, and for other instantiations of `L1`, that transition may not satisfy this relation.

Programming with RIF Versus Classic Labels. We use the term *classic label* to refer to an information flow label that specifies the same restrictions on all values derived from the value with which this label is associated (e.g., [8]).[10] For example, if a user's `PIN` is associated with a classic label specifying that only this user is allowed to read `PIN`, then even the result of a PIN check involving `PIN` is allowed to be read only by that user, instead of by everyone. Classic labels often impose more restrictions than needed.

Information flow control systems employing classic labels (e.g., [21,22]) are forced to use explicit declassification (for confidentiality) and endorsement (for integrity) commands to attach appropriate labels to derived values (i.e., labels

[10] Classic labels can be simulated by one-state RIF automata.

that impose weaker restrictions). Reclassifications in JRIF have a concise description in terms of an identifier (i.e., the reclassifier); declassifications and endorsements for classic labels are more verbose, since they glue a target label (i.e., the label that will be attached to the output) and, sometimes, must include the source label (i.e., the label attached to the input) as well.

JRIF labels are more verbose than classic labels, but there is a pay-off—changes to confidentiality and integrity specified in JRIF labels are not expressible by classic labels. Systems using classic labels need additional program code to emulate JRIF labels. This additional code is not automatically checkable for security, and thus, the programmer bears the full responsibility to implement the intended policy correctly.

Compared to languages using classic labels, JRIF better separates program logic from information flow policies. This makes JRIF programs easier to write and easier to maintain. Suppose, for example, that a programmer decides that some input value—a game player's name—should not be declassified when formerly it was.

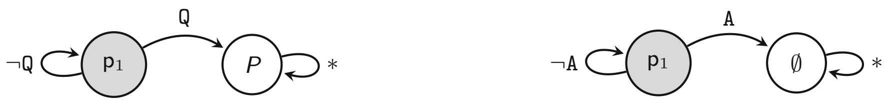
– In JRIF, this change to the program involves modifying the JRIF label declaration on any field storing the player's name. The *c*-automaton of the label would be inspected and edited so that it contains no transitions to automaton states that map to additional principals.
– To accommodate this change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

4 Example Applications Using JRIF

4.1 Battleship

The Battleship game 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 adversary's ship coordinates correctly. Also, players must be restricted from changing the position of their ships after initial placements.

A simple *c*-automaton suffices to specify the confidentiality policy for the ship-coordinates of each player. 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 Fig. 6a, where `Q` is the reclassifier for the



(a) A *c*-automaton for ship-coordinates.

(b) An *i*-automaton for ship-coordinates.

Fig. 6. RIF automata for ship-coordinates

```
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);
}
```

Fig. 7. Method `processQuery` from JRIF implementation. It checks the success of opponent's hit.

operation that checks whether an opponent's attack succeeded, and P is the set of all principals.

The integrity policy of ship-coordinates can be expressed using an *i*-automaton. Once p_1 selects the coordinates of her ships, they are as trusted as p_1. After ship-coordinates are chosen, they should not be changed during the game. So, before the game starts, there is a game operation whose reclassifier raises the integrity of all ship-coordinates, thereby ensuring that neither player can make changes. An *i*-automaton that expresses this policy is presented in Fig. 6b, where `A` is the reclassifier annotating the operation that accepts the initial coordinates.

We borrowed Jif's implementation of Battleship [21] to show that Jif programs are easily ported to JRIF. To obtain that JRIF port, we replaced Jif labels with JRIF labels, and we replaced various Jif declassification or endorsement commands with JRIF reclassifications. Methods in the Jif implementation that involved only label parameters and dynamic labels could be used without any modification in the JRIF implementation. Figure 7 contains a method of the Battleship implementation in JRIF. This method demonstrates the use of the *c*-automaton in Fig. 6a and the application of reclassifier `Q`. The full JRIF source for the Battleship implementation is found on JRIF's web page [17], along with the original Jif source (for comparison).

4.2 A Shared Calendar

To explore the expressive power of JRIF labels, we developed a shared calendar application from scratch.[11] The application allows users to create and share events in calendars. Each event consists of fields: time, date, duration, and description. Declassification, classification, endorsement, and deprecation all are employed in this application. Also, users may choose dynamic JRIF labels to associate with values.

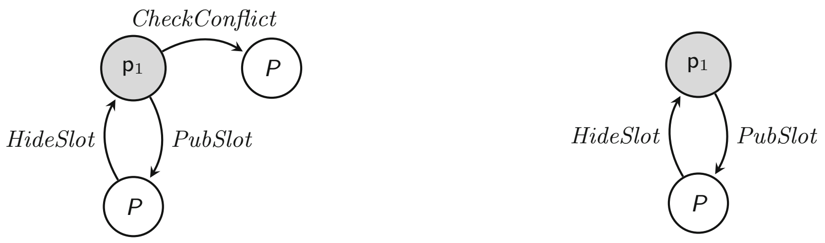
Operations supported by our shared calendar include:

- 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 given the lowest integrity, as are all values that subsequently 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 8 illustrates c-automata for events created by a principal p_1. The c-automaton in Fig. 8a permits a full declassification triggered by reclassifier *CheckConflict*; the c-automaton in Fig. 8b does not. Both c-automata specify a declassification under *PubSlot*, and a classification under *HideSlot*. Figure 9 gives corresponding i-automata for the events of p_1. The i-automaton in Fig. 9a permits a full endorsement triggered by reclassifier *CheckConflict*; the i-automaton in Fig. 9b 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*.

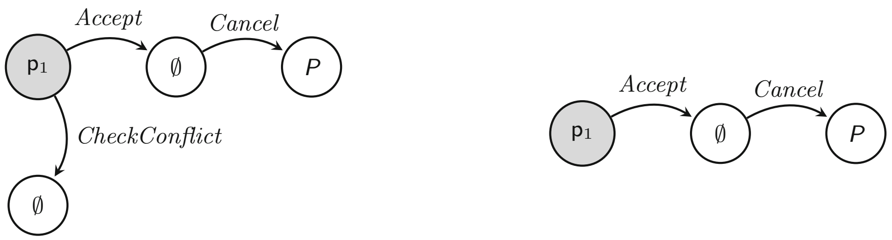
[11] Source code for this shared calendar implementation in JRIF can be found on JRIF's web page [17].



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

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

Fig. 8. RIF automata for event confidentiality. Self-loops are omitted for clarity.



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

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

Fig. 9. RIF automata for event integrity. Self-loops are omitted; for instance, the result of applying *CheckConflict* to a canceled event has low integrity.

Dynamic labels are used extensively in the shared calendar application. Figure 10 excerpts from the conflict-checking method. Here, the label of the event is checked dynamically to see whether it permits the result from the conflict check to be declassified and endorsed before performing the corresponding operation. In Fig. 10, `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. If `lEvt` and `lCal` after having taken a `C` transition impose no restrictions to the resulting value, and if `hasConflict` is `true`, then a conflict will be announced.

Different users may tag events with different dynamic labels. For example, a user might pick the *c*-automaton in Fig. 8a for some events but pick the *c*-automaton in Fig. 8b for others. Events can have different *i*-automata, too. An unshared event has one of the *i*-automata in Fig. 9, but an accepted event can be treated with higher integrity and thus be tagged with the *i*-automaton denoted by taking the *Accept* transition. In addition, the time slot of some events could be

```
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;       // A conflict will be announced
    else
        result = false;      // No conflict will be announced
}
```

Fig. 10. Checking if the conflict is allowed to be declassified and endorsed, where `C` corresponds to reclassifier *CheckConflict*.

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: an event and 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 [36].

5 Building the JRIF Compiler

We built a JRIF compiler by modifying the existing Jif compiler in relatively straightforward ways.[12] Extending compilers for other information flow languages ought to be similar. This should not be so surprising: JRIF labels expose the same interface to a type system as native information flow labels.

Our strategy for building JRIF involved three steps:

1. Add syntax for JRIF labels and for annotating expressions with reclassifiers.
2. Add typing rules for annotated expressions (according to Sect. 3.1).
3. Modify the type checker to handle this more expressive class of labels:
   (a) implement the restrictiveness relation on JRIF labels,
   (b) add an axiom stipulating that this relation is monotone with respect to transition function $\mathcal{T}$.

Item (3b) is essential for supporting our richer language of label comparisons. For example, if relation $\texttt{l2} \sqsubseteq \texttt{l1}$ holds for two dynamic JRIF labels, then the type checker must be able to deduce that $\mathcal{T}(\texttt{l2}, \mathsf{f}) \sqsubseteq \mathcal{T}(\texttt{l1}, \mathsf{f})$ holds for every f.

We decided to build JRIF by extending the Jif compiler because Jif is a widely studied language for information flow control and the Jif compiler is readily available. JRIF adds 6k lines of code to Jif (which contains 230k LOC). Out of the 494 Java classes comprising 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. Moreover, extending Jif enabled us to harness Jif features, such as dynamic labels,

[12] The source code for JRIF can be found on JRIF's web page [17].

label parameters, and label inference, which reduce the annotation burden on the programmer.

Some features of Jif are orthogonal to enforcing JRIF 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 JRIF labels are concerned with what operation is applied to what value, authority and ownership are ignored for the enforcement of JRIF labels.

What the JRIF Compiler Enforces
Label checking in information flow control systems usually enforces noninterference [11] 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 these executions should agree on computed 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. Instead, *piesewise noninterference* (PWNI) [16] can handle arbitrary reclassifications caused by applied operations. PWNI allows flows through specified declassifications, but also it prevents classified information from being leaked. We have proven that RIF automata enforce PWNI for a simple imperative language [16], giving us confidence in the formal guarantees enforced by the JRIF type system. Many models (e.g., [1,2,7,19,30]) have been proposed for expressing and enforcing policies that permit changing the restrictions imposed on values, but PWNI is the first to handle arbitrary reclassifications caused by applied operations.

6 Related Work

Expressive structures, like automata, have previously been used to represent information flow specifications. Program dependence graphs [12,14], which represent data and flow dependencies between values, specify allowable declassifications. And Rocha et al. [25,26] employ *policy graphs* to specify sequences of functions that cause declassifications. However, this work does not handle arbitrary reclassifications; it only handles declassifications.

Several programming language approaches that control declassification (e.g., [13]) employ some notion of *trusted processes* or *code* [3]. Early versions of Jif used *selective downgrading* [24] to refine this idea by allowing different principals to trust different pieces of code. These systems enforce a form of *intransitive* flow policy [28], since direct flows that do not involve the declassifying operation are prohibited.

In capability-based systems, such as Flume [18], HiStar [35], Asbestos [9], Aeolus [6], Laminar [27], and LIO [33], declassifications are allowed to be performed by procedures possessing specific capabilities. Implementing fine-grained information flow policies in these systems would require a large number of different kinds of capabilities, possibly reaching the number of objects that should be protected in a system.

Chong and Myers [7] introduce information flow specifications that use conditions on program state as a basis for deciding when a value may be reclassified. Paragon [4] and SHAI [10] can also support policies that specify reclassifications based on program state predicates. These policies cannot always be translated into RIF automata, because the former specify reclassification based on a richer set of conditions. RIF automata could be compiled to such policies. The translation requires extending the JRIF program state: each JRIF variable is implemented using a value and the sequences of operations involved in deriving this value. It is a cumbersome translation, and we believe that it will generally be difficult to express a RIF automaton as a policy that specifies reclassifications based on program state predicates.

Li and Zdancewic [19] 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. [31] 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.

7 Conclusion

JRIF is an extension of Java for supporting Reactive Information Flow Control based on RIF automata. RIF automata specify restrictions on the values they are associated with, along with the RIF automaton to associate with derived values. The JRIF compiler was implemented by extending the Jif compiler and run time, thereby demonstrating that RIF automata are easily incorporated into languages that already support information flow types.

JRIF's type system is more expressive than classic information flow type systems. For instance, JRIF allows programmers to specify rich policies based on the sequence of operations used to derive a value. Existing programming languages allow such policies to be emulated in the state and control flow of a program, but doing so invariably makes code more complex and provides few security guarantees.

We illustrated JRIF programs with 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. Askarov, A., Sabelfeld, A.: Gradual release: unifying declassification, encryption and key release policies. In: IEEE Symposium on Security and Privacy, pp. 207–221 (2007). https://doi.org/10.1109/SP.2007.22
2. Banerjee, A., Naumann, D., Rosenberg, S.: Expressive declassification policies and modular static enforcement. In: IEEE Symposium on Security and Privacy, pp. 339–353 (2008). https://doi.org/10.1109/SP.2008.20
3. Bell, E.D., LaPadula, J.L.: Secure computer systems: mathematical foundations (1973)
4. Broberg, N., van Delft, B., Sands, D.: Paragon for practical programming with information-flow control. In: Shan, C. (ed.) APLAS 2013. LNCS, vol. 8301, pp. 217–232. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03542-0_16
5. Broberg, N., Sands, D.: Flow locks: towards a core calculus for dynamic flow policies. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 180–196. Springer, Heidelberg (2006). https://doi.org/10.1007/11693024_13
6. Cheng, W., et al.: Abstractions for usable information flow control in Aeolus. In: Proceedings of the 2012 USENIX Conference on Annual Technical Conference, USENIX ATC 2012, p. 12. USENIX Association, Berkeley (2012). http://dl.acm.org/citation.cfm?id=2342821.2342833
7. Chong, S., Myers, A.: End-to-end enforcement of erasure and declassification. In: 2008 IEEE 21st Computer Security Foundations Symposium, CSF 2008, pp. 98–111 (2008). https://doi.org/10.1109/CSF.2008.12
8. Denning, D.E.R.: Secure information flow in computer systems. Ph.D. thesis, West Lafayette, IN, USA (1975)
9. Efstathopoulos, P., et al.: Labels and event processes in the Asbestos operating system. In: Proceedings of the Twentieth ACM Symposium on Operating Systems Principles, SOSP 2005, pp. 17–30. ACM, New York (2005). https://doi.org/10.1145/1095810.1095813
10. Elnikety, E., Garg, D., Druschel, P.: SHAI: enforcing data-specific policies with near-zero runtime overhead. Technical report, Max Planck Institute for Software Systems, Saarland Informatics Campus, Germany, January 2018
11. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy, pp. 11–20 (1982)
12. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Int. J. Inf. Secur. **8**(6), 399–422 (2009). https://doi.org/10.1007/s10207-009-0086-1
13. Hicks, B., King, D., McDaniel, P., Hicks, M.: Trusted declassification: high-level policy for a security-typed language. In: Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security, PLAS 2006, pp. 65–74. ACM, New York (2006). https://doi.org/10.1145/1134744.1134757
14. Johnson, A., Waye, L., Moore, S., Chong, S.: Exploring and enforcing security guarantees via program dependence graphs. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pp. 291–302. ACM, New York (2015). https://doi.org/10.1145/2737924.2737957

15. Kanav, S., Lammich, P., Popescu, A.: A conference management system with verified document confidentiality. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 167–183. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_11
16. Kozyri, E.: Enhancing expressiveness of information flow labels: reclassification and permissiveness. Ph.D. thesis, Ithaca, NY, USA (2018)
17. Kozyri, E., Arden, O., Myers, A.C., Schneider, F.B.: JRIF: Java with Reactive Information Flow, February 2016. Software release http://www.cs.cornell.edu/jrif/
18. Krohn, M., et al.: Information flow control for standard OS abstractions. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, SOSP 2007, pp. 321–334. ACM, New York (2007). https://doi.org/10.1145/1294261.1294293
19. Li, P., Zdancewic, S.: Downgrading policies and relaxed noninterference. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, pp. 158–170. ACM, New York (2005). https://doi.org/10.1145/1040305.1040319
20. Li, P., Zdancewic, S.: Practical information-flow control in web-based information systems. In: Proceedings of the 18th IEEE Workshop on Computer Security Foundations, CSFW 2005, pp. 2–15. IEEE Computer Society, Washington, DC (2005). https://doi.org/10.1109/CSFW.2005.23
21. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: JIF 3.0: Java Information Flow. Software release http://www.cs.cornell.edu/jif, July 2006
22. Myers, A.C.: JFlow: Practical mostly-static information flow control. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1999, pp. 228–241. ACM, New York (1999). https://doi.org/10.1145/292540.292561
23. Myers, A.C., Liskov, B.: A decentralized model for information flow control. In: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, SOSP 1997, pp. 129–142. ACM, New York (1997). https://doi.org/10.1145/268998.266669
24. Pottier, F., Conchon, S.: Information flow inference for free. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP 2000, pp. 46–57. ACM, New York (2000). https://doi.org/10.1145/351240.351245
25. Rocha, B., Bandhakavi, S., den Hartog, J., Winsborough, W., Etalle, S.: Towards static flow-based declassification for legacy and untrusted programs. In: IEEE Symposium on Security and Privacy, pp. 93–108 (2010). https://doi.org/10.1109/SP.2010.14
26. Rocha, B., Conti, M., Etalle, S., Crispo, B.: Hybrid static-runtime information flow and declassification enforcement. IEEE Trans. Inf. Forensics Secur. **8**(8), 1294–1305 (2013). https://doi.org/10.1109/TIFS.2013.2267798
27. Roy, I., Porter, D.E., Bond, M.D., McKinley, K.S., Witchel, E.: Laminar: practical fine-grained decentralized information flow control. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, pp. 63–74. ACM, New York (2009). https://doi.org/10.1145/1542476.1542484
28. Rushby, J.: Noninterference, transitivity and channel-control security policies. Technical report (1992)
29. Sabelfeld, A., Myers, A.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003). https://doi.org/10.1109/JSAC.2002.806121

30. Sabelfeld, A., Myers, A.C.: A model for delimited information release. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) ISSS 2003. LNCS, vol. 3233, pp. 174–191. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-37621-7_9
31. Sabelfeld, A., Sands, D.: Declassification: dimensions and principles. J. Comput. Secur. **17**(5), 517–548 (2009). http://dl.acm.org/citation.cfm?id=1662658.1662659
32. Schneider, F.B., Walsh, K., Sirer, E.G.: Nexus Authorization Logic (NAL): design rationale and applications. ACM Trans. Inf. Syst. Secur. **14**(1), 8:1–8:28 (2011). https://doi.org/10.1145/1952982.1952990
33. Stefan, D., Russo, A., Mitchell, J.C., Mazières, D.: Flexible dynamic information flow control in Haskell. In: Proceedings of the 4th ACM Symposium on Haskell, Haskell 2011, pp. 95–106. ACM, New York (2011). https://doi.org/10.1145/2034675.2034688
34. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2–3), 167–187 (1996). http://dl.acm.org/citation.cfm?id=353629.353648
35. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making information flow explicit in HiStar. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation-Volume 7, OSDI 2006, p. 19. USENIX Association, Berkeley (2006). http://dl.acm.org/citation.cfm?id=1267308.1267327
36. Zheng, L., Myers, A.C.: Dynamic security labels and static information flow control. Int. J. Inf. Secur. **6**(2), 67–84 (2007). https://doi.org/10.1007/s10207-007-0019-9