View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by eCommons@Cornell

RIF: Reactive Information Flow Labels*

Elisavet Kozyri
Harvard University
ekozyri@seas.harvard.edu

Fred B. Schneider
Cornell University
fbs@cs.cornell.edu

December 10, 2019

Abstract

Restrictions that a *reactive information flow* (RIF) label imposes on a value are determined by the sequence of operations used to derive that value. This allows declassification, endorsement, and other forms of reclassification to be supported in a uniform way. *Piecewise noninterference* (PWNI) is introduced as a fitting security policy, because noninterference is not suitable. A type system is given for static enforcement of PWNI in programs that associate *checkable* classes of RIF labels with variables. Two checkable classes of RIF labels are described: *RIF automata* are general-purpose and based on finite-state automata; κ-*labels* concern confidentiality in programs that use cryptographic operations.

1 Introduction

Data is usually accompanied by restrictions about uses. Information flow control [50] has been employed to ensure those restrictions propagate from inputs to the outputs of operations. In Denning's initial work [21] and in much that has followed—both for confidentiality and integrity—the set of

*Supported in part by AFOSR grants F9550-16-0250, F9550-19-1-0264, and NSF grant 1642120. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

restrictions assigned to the output of an operation is the union of the restrictions associated with its inputs. But by ignoring the operator and the values of inputs, that approach can be too conservative.

The most general formulation of *flow-derived restrictions* would assign restrictions to the output of an operation $op(x_1, x_2, \ldots, x_n)$ according to operator op, its inputs $x_1, x_2, \ldots, x_n$, and the restrictions associated with those inputs. That output might warrant fewer restrictions, additional restrictions, or an incomparable set of restrictions than are associated with its inputs.

- With an operation that computes the winner of an election, the inputs are votes and the output is the majority. Each input is secret to the principal casting that vote, whereas the output ought to be readable by any principal. So the output should be associated with fewer restrictions than the inputs.

- A conference-management system matches papers to reviewers, where that matching is generated by a non-deterministic computation. The inputs—a list of reviewers and a list of submissions—can be read by the entire program committee, but conflicts of interest dictate that only a subset of the program committee learn which reviewers are assigned to any given paper. So, as with other aggregation problems [31], outputs are associated with more stringent restrictions than inputs.

Previous work on information flow control—declassification and erasure policies [16], information flow locks [11, 10], typed declassification [24], expressions for declassification (for confidentiality) and endorsement (for integrity) [44, 46], and capability-based mechanisms for downgrading security policies [35, 49, 56]—does not support arbitrary changes in restrictions linked with specific operations. Other approaches (e.g., [30, 40, 47, 48, 53]) allow such changes but only between two levels of labels (e.g., public and secret).

Reactive information flow (RIF) labels, which we introduce in §2, seek to address these limitations by allowing stronger, weaker, or incomparable restrictions to be associated with the output of an operation, as determined by the operator and the restrictions on inputs. Piecewise noninterference (PWNI) described in §4 then extends classical noninterference in a way that handles changes to restrictions that RIF labels support. Using terminology introduced by Sabelfeld and Sands [53], PWNI stipulates for programs that associate RIF labels with variables, *what* information undergoes changes of restrictions and *where* in the program code. A type system is given in §5

to support static enforcement of PWNI for certain classes of RIF labels. Examples of those classes include RIF automata in §3 and κ-labels in §6. In §7 we discuss how previous work relates to RIF labels and PWNI.

2 RIF Labels

Restrictions. Restrictions are assumed to be elements of a join semilattice $\langle R,\ \sqcup_R,\ \sqsubseteq_R\rangle$.[1] For confidentiality, an element of R might identify which principals are allowed to read some value, either by enumerating that set of principals or by giving the name (e.g., public, secret, etc.) for such a set; for integrity, it might identify the set of principals allowed to write that value. But other kinds of restrictions also could be specified using elements in R—*use-based privacy* [9] is an example.

Relation $r \sqsubseteq_R r'$ is satisfied if compliance with restrictions $r' \in R$ implies compliance with restrictions $r \in R$, so r' is *at least as strong* as r or, equivalently, r is *at least as weak* as r'. When elements of R denote sets of principals, then for confidentiality we would define $r \sqcup_R r'$ to be $r \cap r'$ and define $r \sqsubseteq_R r'$ to hold if and only if $r \supseteq r'$ holds—if r' allows a principal to read then so does r, but r might also allow other principals to read, too. And, for integrity, we would define $r \sqcup_R r'$ to be $r \cup r'$ and define $r \sqsubseteq_R r'$ to hold if and only if $r \subseteq r'$ holds.

Reclassifiers and RIF Labels. A *reclassifier* abstracts how an operation changes the restrictions for an argument.[2] To associate reclassifiers with operations, we extend a language that defines *ordinary expressions*: variables and terms $op(\mathcal{E}_1, \ldots, \mathcal{E}_n)$, where op is an operator and each $\mathcal{E}_i$ is an ordinary expression. For ordinary expressions $\mathcal{E}_1$, ..., $\mathcal{E}_n$,

$$[op(\mathcal{E}_1, \ldots, \mathcal{E}_n)]_{\mathsf{f_1},\mathsf{f_2},\ldots,\mathsf{f_n}} \tag{1}$$

defines a *reclassifying expression*. It specifies that reclassifier $\mathsf{f_i}$ identifies how the restrictions associated with the value of ordinary expression $\mathcal{E}_i$ should be changed for constructing the restrictions associated with the value produced by $op(\mathcal{E}_1, \ldots, \mathcal{E}_n)$. Notation $[op(\mathcal{E}_1, \ldots, \mathcal{E}_n)]_{\mathsf{f}}$ is used as a shorthand

[1] A partial order would suffice for the theory of RIF labels developed in this section. A join semilattice becomes useful for certain instantiations of RIF labels, as illustrated in §3 and §6.

[2] The term *reclassify* is taken from Denning's thesis [21] where it is used to describe an operation that changes the restrictions imposed on objects.

for $[op(\mathcal{E}_1, \ldots, \mathcal{E}_n)]_{\mathsf{f},\mathsf{f},\ldots,\mathsf{f}}$, and we sometimes abbreviate $[op(\mathcal{E}_1, \ldots, \mathcal{E}_n)]_{\mathsf{f}_1,\mathsf{f}_2,\ldots,\mathsf{f}_n}$ by $[op(\mathcal{E}_1, \ldots, \mathcal{E}_n)]_{\bar{\mathsf{f}}}$ or simply $[\mathcal{E}]_{\bar{\mathsf{f}}}$ if the elided specifics are irrelevant.

When reclassifying expressions are used to compute values, then sequences of reclassifiers offer abstract descriptions for the series of operations that have been applied to values as program execution proceeds. Such a sequence of reclassifiers then provides a basis for determining the set of restrictions associated with a computed value. For example, consider the following program.

$$w_1 := [div(x_1, x_2)]_{\mathsf{f}};\ y_2 := [mod(w_1, w_2)]_{\mathsf{f}'};\ z_3 := [add(y_1, y_2, y_3)]_{\mathsf{f}''} \tag{2}$$

Here, restrictions on the value stored in z_3 are derived from:

- the restrictions on the value stored in x_1 and in x_2, changed according to sequence $\mathsf{f}\,\mathsf{f}'\,\mathsf{f}''$ because each of these values flows to w_1 through f, then w_1 flows to y_2 through f', and finally y_2 flows to z_3 through f''.
- the restrictions on the value stored in w_2, changed according to sequence $\mathsf{f}'\mathsf{f}''$, and
- the restrictions on the values stored in y_1 and y_3, changed according to f''.

A RIF label λ specifies a set of restrictions for an associated value v as well as for values derived by executing operations on v. Formally, a RIF label maps sequences of reclassifiers to elements of underlying join semilattice $\langle R,\ \sqcup_R,\ \sqsubseteq_R\rangle$ of restrictions. This mapping is specified for a set Λ of RIF labels by giving a set $\mathcal{F}$ of reclassifiers along with two functions $\mathcal{R}$ and $\mathcal{T}$.

Definition ($\mathcal{R}$ Function). *$\mathcal{R}$ maps $\lambda \in \Lambda$ to the restriction $r \in R$ that λ currently imposes:*

$$\mathcal{R}\colon \Lambda \to R \tag{3}$$

Definition ($\mathcal{T}$ Function). *$\mathcal{T}$ maps $\lambda \in \Lambda$ and each reclassifier $\mathsf{f} \in \mathcal{F}$ to a RIF label that should be associated with the value produced by an operation f abstracts:*

$$\mathcal{T}\colon \Lambda \times \mathcal{F} \to \Lambda \tag{4}$$

As an example, the restrictions associated with the value of a reclassifying expression (1) will incorporate restrictions $\mathcal{R}(\mathcal{T}(\lambda_i, \mathsf{f}_i))$ derived from RIF label λ_i being associated with the value of expression $\mathcal{E}_i$.

$\mathcal{T}$ is extended to a finite sequence[3] $F \in \mathcal{F}^*$ of reclassifiers in the usual way, with empty sequence ϵ of reclassifiers considered an element of every set $\mathcal{F}$ of reclassifiers (where it serves as an identity reclassifier).

$$\mathcal{T}(\lambda, \epsilon) \triangleq \lambda \quad (5)$$

$$\mathcal{T}(\lambda, F\mathsf{f}) \triangleq \mathcal{T}(\mathcal{T}(\lambda, F), \mathsf{f}) \quad (6)$$

Classes of RIF Labels. A class of RIF labels can serve as the basis for a join semilattice $\langle \Lambda, \sqcup_\Lambda, \sqsubseteq_\Lambda \rangle$, where the cardinality of Λ can be infinite. The *join* operator $\sqcup_\Lambda$ is used for combining RIF labels; the *restrictiveness* relation $\sqsubseteq_\Lambda$ specifies whether one RIF label is *at least as restrictive as* another. We posit that Λ includes elements $\bot$ and $\top$ such that for all $\lambda \in \Lambda$: $\bot \sqsubseteq_\Lambda \lambda$ and $\lambda \sqsubseteq_\Lambda \top$ hold.

Since, by definition, $\lambda \sqsubseteq_\Lambda \lambda \sqcup_\Lambda \lambda'$ and $\lambda' \sqsubseteq_\Lambda \lambda \sqcup_\Lambda \lambda'$ hold in a join semilattice, a combination of RIF labels is at least as restrictive as any of its constituents. For the examples of RIF labels in this paper, $\sqcup_\Lambda$ and $\sqcup_R$ are related in a way that ensures restrictions imposed by $\lambda \sqcup_\Lambda \lambda'$ are the same as the combined restrictions imposed by individual RIF labels λ and λ'.[4]

$$\mathcal{R}(\lambda \sqcup_\Lambda \lambda') = \mathcal{R}(\lambda) \sqcup_R \mathcal{R}(\lambda') \quad (7)$$

The definition of restrictiveness relation $\sqsubseteq_\Lambda$ depends on partial order $\sqsubseteq_R$ for set R of restrictions, $\mathcal{R}$, and $\mathcal{T}$:

$$\lambda \sqsubseteq_\Lambda \lambda' \triangleq (\forall F \in \mathcal{F}^*\colon\ \mathcal{R}(\mathcal{T}(\lambda, F)) \sqsubseteq_R \mathcal{R}(\mathcal{T}(\lambda', F))) \quad (8)$$

This definition ensures that if $\lambda \sqsubseteq_\Lambda \lambda'$ holds, then

- current restrictions $\mathcal{R}(\lambda')$ specified by λ' are at least as strong as what λ imposes because, by definition, $\epsilon \in \mathcal{F}$ and thus $\epsilon \in \mathcal{F}^*$, $\mathcal{R}(\mathcal{T}(\lambda, \epsilon)) = \mathcal{R}(\lambda)$ and $\mathcal{R}(\mathcal{T}(\lambda', \epsilon)) = \mathcal{R}(\lambda')$ hold, so (8) implies that $\mathcal{R}(\lambda) \sqsubseteq_R \mathcal{R}(\lambda')$, and

- restrictions that λ' imposes for any derived value are at least as strong as what λ imposes—if v flows to w then there is a sequence $F \in \mathcal{F}^*$ that v flows through, and (8) requires that $\mathcal{R}(\mathcal{T}(\lambda, F)) \sqsubseteq_R \mathcal{R}(\mathcal{T}(\lambda', F))$ hold.

[3] As is conventional, $\mathcal{F}^*$ denotes the set of finite sequences of elements in $\mathcal{F}$.

[4] Reasonable examples of RIF labels do exist where (7) does not hold.

	$\mathcal{R}(\lambda) \subset \mathcal{R}(\mathcal{T}(\lambda, \mathsf{f}))$	$\mathcal{R}(\lambda) \supset \mathcal{R}(\mathcal{T}(\lambda, \mathsf{f}))$
Confidentiality	*declassification*	*classification*
Integrity	*deprecation*	*endorsement*

Figure 1: Terminology for reclassifications

The quantification over all sequences $F \in \mathcal{F}^*$ in (8) means that this definition for $\lambda \sqsubseteq_\Lambda \lambda'$ is conservative, since it imposes conditions for sequences F of reclassifiers that never arise in a program execution.

Definition (RIF Class)**.** *A* class *of RIF labels is formed as follows:*

$$\langle\langle R,\ \sqcup_R,\ \sqsubseteq_R\rangle, \langle\Lambda,\ \sqcup_\Lambda,\ \sqsubseteq_\Lambda\rangle, \mathcal{F}, \mathcal{R}, \mathcal{T}\rangle$$

The definition of a RIF class is silent about the existence of algorithms for computing $\sqcup_\Lambda$ or for deciding $\sqsubseteq_\Lambda$. We say that a RIF class is *checkable* if and only if:

(i) There exists an algorithm for computing $\lambda \sqcup_\Lambda \lambda'$ for all $\lambda, \lambda' \in \Lambda$.

(ii) There exists a sound (no false positives) if not complete (false negatives possible) test for determining whether $\lambda \sqsubseteq_\Lambda \lambda'$ holds for all $\lambda, \lambda' \in \Lambda$.

The type system we give in §5 is decidable if its RIF labels are from a checkable class.

Reclassifications. A reclassifier f triggers a *reclassification* in a RIF label λ, if $\mathcal{T}(\lambda, \mathsf{f}) \neq \lambda$ holds. In the literature, reclassification is categorized based on the restrictions that new and old labels impose—specifically, whether $\mathcal{R}(\mathcal{T}(\lambda, \mathsf{f}))$ is weaker than $\mathcal{R}(\lambda)$. Figure 1 defines specialized terminology for this categorization, where $\mathcal{R}(\lambda)$ gives a set of principals that must be trusted not to divulge the value (for confidentiality) or not to have corrupted the value (for integrity). So, according to Figure 1, reclassifier f triggers a *declassification* if the principals trusted not to divulge the value by the new label (i.e., $\mathcal{T}(\lambda, \mathsf{f})$) are a superset of those trusted by the old label (i.e., λ); if they are a subset, then f triggers a *classification.* Also, reclassifier f triggers an *endorsement* when the principals trusted not to have corrupted the value by the new label (i.e., $\mathcal{T}(\lambda, \mathsf{f})$) are a subset of those trusted by the old label (i.e., λ); if they are a superset, then f triggers a deprecation.

The approach in this paper, however, involves a finer-grained categorization of reclassifications. That categorization is based on how a new RIF label $\mathcal{T}(\lambda, \mathsf{f})$ is related to the old RIF label λ: When $\mathcal{T}(\lambda, \mathsf{f}) \sqsubset_\Lambda \lambda$ holds then we say that f triggers a *downgrade*; when $\mathcal{T}(\lambda, \mathsf{f}) \not\sqsubseteq_\Lambda \lambda$ holds then we say that f triggers an *upgrade*. And the reclassifying expressions that appear in a program in conjunction with the RIF labels that tag variables in these expressions, specify *what* information is being reclassified and *where* in the program. In particular, given a reclassifying expression $[\mathcal{E}]_{\bar{\mathsf{f}}}$ at a particular program point, if the RIF label of $\mathcal{E}$ is different from the RIF label of $[\mathcal{E}]_{\bar{\mathsf{f}}}$, then $\bar{\mathsf{f}}$ triggers a reclassification: The *what* dimension of this reclassification is the current value of $\mathcal{E}$, and the *where* dimension is that particular program point (i.e., code locality is employed).

The RIF label for ordinary and reclassifying expressions are deduced as usual from the RIF labels of the variables that appear in those expressions. Starting from a fixed mapping Γ that associates a RIF label $\Gamma(x) \in \Lambda$ with each variable x, define:

$$\Gamma(op(\mathcal{E}_1, \ldots, \mathcal{E}_n)) \triangleq \Gamma(\mathcal{E}_1) \sqcup_\Lambda \ldots \sqcup_\Lambda \Gamma(\mathcal{E}_n) \tag{9}$$

$$\Gamma([op(\mathcal{E}_1, \ldots, \mathcal{E}_n)]_{\mathsf{f}_1, \ldots, \mathsf{f}_n}) \triangleq \mathcal{T}(\Gamma(\mathcal{E}_1), \mathsf{f}_1) \sqcup_\Lambda \ldots \sqcup_\Lambda \mathcal{T}(\Gamma(\mathcal{E}_n), \mathsf{f}_n) \tag{10}$$

Thus, the RIF label associated with a reclassifying expression combines RIF labels obtained after the indicated transitions have been performed.

We give a concrete example of reclassification using program (2) and assuming that the following restrictiveness relation holds:

$$\Gamma([mod(w_1, w_2)]_{\mathsf{f}'}) \sqsubset_\Lambda \Gamma(mod(w_1, w_2))$$

So reclassifier f' triggers a downgrade. For the *what* dimension, the value to be downgraded is the result of applying *mod* to the values stored in w_1 and w_2 at program point "$y_2 := [mod(w_1, w_2)]_{\mathsf{f}'};\ z_3 := [add(y_1, y_2, y_3)]_{\mathsf{f}''}$"—as opposed to the values stored in w_1 and w_2 at initialization. For the *where* dimension: the value of $mod(w_1, w_2)$ is downgraded at that program point—not earlier. A formal characterization for the security policy satisfied by a program that employs RIF labels and reclassifying expressions is given in §4.

3 RIF Automata

Finite state automata provide the basis for a checkable class of RIF labels, called *RIF automata*, that have broad practical utility. This class of RIF

labels is supported in the JRIF[5] programming language [34]. We built JRIF to gain practical experience with using RIF automata for specifying security policies and to understand the compiler modifications needed when a programming language that supports ordinary security label types is converted to use RIF automata. The *privacy automata* used in the Avanance language [9] for specifying use-based privacy are also instances of RIF automata.

Formalization of RIF Automata. A finite state automaton can serve as a RIF label λ_α by: (i) having the set of reclassifiers be the automaton's input alphabet, and (ii) associating restrictions with each automaton state. Restrictions imposed by λ_α are those associated with the automaton's current state. Reclassifiers change the automaton's current state. Thus they cause a (potentially) different set of restrictions to be imposed.

A set Λ_{RA} comprising RIF automata is defined relative to some join semilattice of restrictions $\langle R, \sqcup_R, \sqsubseteq_R \rangle$. Each RIF automaton $\lambda_\alpha \in \Lambda_{\mathrm{RA}}$ is described by a 5-tuple

$$\lambda_\alpha \triangleq \langle Q_\alpha, \mathcal{F}, \delta_\alpha, q_\alpha, \rho_\alpha \rangle \tag{11}$$

where:

Q_α is a finite set of automaton states

$\mathcal{F}$ is the finite set of reclassifiers

$\delta_\alpha \colon Q_\alpha \times \mathcal{F} \to Q_\alpha$ is a (deterministic) next-state transition function[6]

$q_\alpha \in Q_\alpha$ is the current state of the RIF automaton

$\rho_\alpha \colon Q_\alpha \to R$ gives the restrictions associated with each automaton state

We require transition function δ_α to be total, so any sequence of reclassifiers from $\mathcal{F}$ causes a sequence of transitions. And for $F \in \mathcal{F}^*$ we write $\lambda_\alpha(F)$ to denote the automaton that results when λ_α performs the transitions indicated by a sequence F of reclassifiers. Automaton $\lambda_\alpha(F)$ thus replaces current state q_α of λ_α with $\delta^*_\alpha(q_\alpha, F)$; we define $\lambda_\alpha(F)$ formally by:

$$\lambda_\alpha(F) \triangleq \lambda_\alpha[q_\alpha \mapsto \delta^*_\alpha(q_\alpha, F)]$$

[5] JRIF is a variant of the Jif security-typed programming language, which extends Java with support for information flow control.

[6] Closure $\delta^*_\alpha \colon Q_\alpha \times \mathcal{F}^* \to Q_\alpha$ is obtained from δ_α in the usual way: $\delta^*_\alpha(q, \epsilon) \triangleq q$ and $\delta^*_\alpha(q, \mathsf{Ff}) \triangleq \delta_\alpha(\delta^*_\alpha(q, \mathsf{F}), \mathsf{f})$

$$\text{where for } q_i \in Q_\alpha\text{:}\quad \lambda_\alpha[q_\alpha \mapsto q_i] \;\triangleq\; \langle Q_\alpha, \mathcal{F}, \delta_\alpha, q_i, \rho_\alpha\rangle$$

$\mathcal{R}_{\mathrm{RA}}$ and $\mathcal{T}_{\mathrm{RA}}$ are defined as expected.

$$\mathcal{R}_{\mathrm{RA}}(\lambda_\alpha) \;\triangleq\; \rho_\alpha(q_\alpha) \qquad\qquad \mathcal{T}_{\mathrm{RA}}(\lambda_\alpha, \mathsf{f}) \;\triangleq\; \lambda_\alpha(\mathsf{f})$$

Instantiating definition (8) of $\sqsubseteq_\Lambda$ using these definitions gets:

$$\lambda_\alpha \sqsubseteq_{\mathrm{RA}} \lambda_{\alpha'} \;\triangleq\; (\forall F \in \mathcal{F}^*\text{:}\;\; \rho_\alpha(\delta^*_\alpha(q_\alpha, F)) \sqsubseteq_R \rho_{\alpha'}(\delta^*_{\alpha'}(q_{\alpha'}, F))) \tag{12}$$

A computable test for deciding $\lambda_\alpha \sqsubseteq_{\mathrm{RA}} \lambda_{\alpha'}$ is now given. It is based on constructing a *restrictiveness product* automaton $\lambda_\alpha \otimes \lambda_{\alpha'}$ that is the product of RIF automata λ_α and $\lambda_{\alpha'}$ but with unreachable automaton states eliminated.

$$\begin{aligned}
\lambda_\alpha \otimes \lambda_{\alpha'} \;\triangleq\;& \langle Q_{\alpha\otimes\alpha'},\ \mathcal{F},\ \delta_{\alpha\otimes\alpha'},\ \langle q_\alpha, q_{\alpha'}\rangle,\ \rho_{\alpha\otimes\alpha'}\rangle \\
\text{where}\quad & Q_{\alpha\otimes\alpha'} \subseteq Q_\alpha \times Q_{\alpha'} \\
& \delta_{\alpha\otimes\alpha'}(\,\langle q, q'\rangle, \mathsf{f}) \;\triangleq\; \langle \delta_\alpha(q, \mathsf{f}),\ \delta_{\alpha'}(q', \mathsf{f})\rangle \\
& \rho_{\alpha\otimes\alpha'}(\,\langle q, q'\rangle\,) \;\triangleq\; \langle \rho_\alpha(q),\ \rho_{\alpha'}(q')\rangle \\
& (\forall q \in Q_{\alpha\otimes\alpha'} : (\exists F_q \in \mathcal{F}^* : \delta^*_{\alpha\otimes\alpha'}(\langle q_\alpha, q_{\alpha'}\rangle, F_q) = q))
\end{aligned} \tag{13}$$

Restrictiveness product automata are not RIF automata because the signature for $\rho_{\alpha\otimes\alpha'}$ in $\lambda_\alpha \otimes \lambda_{\alpha'}$ is

$$\rho_{\alpha\otimes\alpha'}\text{:}\;\; Q_{\alpha\otimes\alpha'} \to R \times R,$$

whereas a RIF automaton would have to associate a single element of R (rather than a pair of elements) with each automaton state.

Condition (13), which stipulates that all states in $Q_{\alpha\otimes\alpha'}$ are reachable, is straightforward to check with a linear-time depth-first search of the state-transition graph for $\delta_{\alpha\otimes\alpha'}$. And condition (13) implies that for any predicate $P(\langle q, q'\rangle)$ on states $\langle q, q'\rangle \in Q_{\alpha\otimes\alpha'}$

$$(\forall \langle q, q'\rangle \in Q_{\alpha\otimes\alpha'}\text{:}\;\; P(\langle q, q'\rangle)) \;=\; (\forall F \in \mathcal{F}^*\text{:}\;\; P(\delta^*_{\alpha\otimes\alpha'}(\langle q_\alpha, q_{\alpha'}\rangle,\ F))) \tag{14}$$

holds. Therefore a decision procedure for $\lambda_\alpha \sqsubseteq_{\mathrm{RA}} \lambda_{\alpha'}$ need only check a predicate on the pair of restrictions associated with each state $\langle q,\, q'\rangle \in Q_{\alpha\otimes\alpha'}$ — namely that $r \sqsubseteq_R r'$ holds if $\rho_{\alpha\otimes\alpha'}(\langle q,\, q'\rangle) = \langle r,\, r'\rangle$.

$$\lambda_\alpha \sqsubseteq_{\mathrm{RA}} \lambda_{\alpha'} \;=\; (\forall \langle q, q'\rangle \in Q_{\alpha\otimes\alpha'}\text{:}\;\; \rho_\alpha(q) \sqsubseteq_R \rho_{\alpha'}(q')) \tag{15}$$

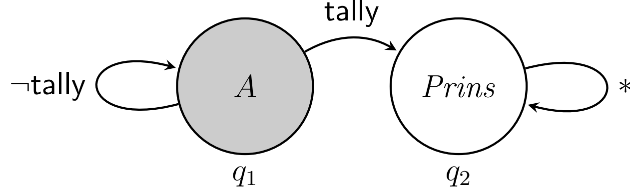

Figure 2: RIF automaton $\lambda_{voter(A)}$ for secret ballots

To prove that (15) is equivalent to definition (12) of $\lambda_\alpha \sqsubseteq_{\text{RA}} \lambda_{\alpha'}$, observe that

$$\begin{array}{ll} & (\forall \langle q, q' \rangle \in Q_{\alpha \otimes \alpha'}\colon\ \rho_\alpha(q) \sqsubseteq_R \rho_{\alpha'}(q')) \\ = & \text{due to (14), with } P(\langle q, q' \rangle) \triangleq \rho_\alpha(q) \sqsubseteq_R \rho_{\alpha'}(q') \\ & (\forall F \in \mathcal{F}^*\colon\ \rho_\alpha(\delta^*_\alpha(q_\alpha, F)) \sqsubseteq_R \rho_{\alpha'}(\delta^*_{\alpha'}(q_{\alpha'}, F))) \end{array}$$

The properties of $\sqcup_{\text{RA}}$ for RIF automata are satisfied by using a form of product construction that does yield a RIF automaton.

$$\begin{array}{lll} \lambda_\alpha \sqcup_{\text{RA}} \lambda_{\alpha'} & \triangleq & \langle Q_{\alpha \sqcup \alpha'},\ \mathcal{F},\ \delta_{\alpha \sqcup \alpha'},\ \langle q_\alpha, q_{\alpha'} \rangle,\ \rho_{\alpha \sqcup \alpha'} \rangle \\ \quad \text{where} & & Q_{\alpha \sqcup \alpha'} = Q_\alpha \times Q_{\alpha'} \\ & & \delta_{\alpha \sqcup \alpha'}(\langle q, q' \rangle, \mathsf{f}) \ \triangleq\ \langle \delta_\alpha(q, \mathsf{f}),\ \delta_{\alpha'}(q', \mathsf{f}) \rangle \\ & & \rho_{\alpha \sqcup \alpha'}(\langle q, q' \rangle) \ \triangleq\ \rho_\alpha(q) \sqcup_{\text{R}} \rho_{\alpha'}(q') \end{array}$$

Note that (7) relating $\sqcup_{\text{R}}$ and $\sqcup_{\text{RA}}$ is satisfied by this definition.

Finally, we prove in the Appendix A that $\langle \Lambda_{\text{RA},},\ \sqsubseteq_{\text{RA}},\ \sqcup_{\text{RA}} \rangle$ is a join semilattice.

Examples of RIF Automata. A security policy that we might associate with the ballot that each participant A casts in an election is: (i) only A may read the ballot's value and (ii) anyone may read the majority value derived from all of the ballots cast. We formalize this security policy as a RIF automaton $\lambda_{voter(A)}$, where *Prins* is the set of principals eligible to learn the election outcome, the join semilattice of underlying restrictions is $\langle 2^{Prins},\ \cap,\ \supseteq \rangle$, and set $\mathcal{F}$ of reclassifiers includes tally, which will be associated with calculating the election outcome.

Figure 2 gives a graphic depiction[7] of $\lambda_{voter(A)}$; the formal definition is:

$$\lambda_{voter(A)} \triangleq \langle \{q_1, q_2\}, \mathcal{F}, \delta_{voter}, q_1, \rho_{voter(A)} \rangle$$

where

$$\delta_{voter}(q, \mathsf{f}) \triangleq \begin{cases} q_2 & \text{if } q = q_1 \wedge \mathsf{f} = \mathsf{tally} \\ q & \text{otherwise} \end{cases}$$

$$\rho_{voter(A)}(q) \triangleq \begin{cases} \{A\} & \text{if } q = q_1 \\ Prins & \text{if } q = q_2 \end{cases}$$

Restriction $\rho_{voter(A)}$ defines which principals can read the value being labeled by $\lambda_{voter(A)}$:

- the value is secret to A if $\lambda_{voter(A)}$ is in state q_1, because $\rho_{voter(A)}(q_1) = \{A\}$, and
- any value derived by using a tally operation becomes public because the current state transitions to q_2 and $\rho_{voter(A)}(q_2) = Prins$ hold.

Thus, according to the terminology of Figure 1, tally causes declassification.

A programmer would use reclassifying expression $[maj(v_A, v_B, \ldots, v_Z)]_{\mathsf{tally}}$ to assert that computing the majority of votes in v_A, v_B, ..., v_Z implements the intended effect of a tally operation. That derived value can be stored in a variable whose RIF label imposes no restriction on readers. Assignment

$$winner := [maj(v_A, v_B, \ldots, v_Z)]_{\mathsf{tally}} \tag{16}$$

has exactly that effect if the RIF automaton associated with each variable v_A is $\lambda_{voter(A)}$ in state q_1, and the RIF automaton associated with variable $winner$ is at least as restrictive as

$$\lambda_{voter(A)}(\mathsf{tally}) \sqcup_{\text{RA}} \lambda_{voter(B)}(\mathsf{tally}) \sqcup_{\text{RA}} \cdots \sqcup_{\text{RA}} \lambda_{voter(Z)}(\mathsf{tally})$$

which happens to be equivalent to $\lambda_{voter(x)}[q_1 \mapsto q_2]$ for any $x \in \{A, \ldots, Z\}$.

[7]A conventional graphical representation for finite-state automata is used. Circles denote states of the automaton. Arrows between states are labeled with lists of reclassifiers, any of which can trigger the allowed transitions. We will write * to abbreviate the list of all reclassifiers and $\neg\ell$ to denote a list of all reclassifiers not contained in list ℓ. The label inside each state q indicates associated restrictions $\rho_\alpha(q)$, and the grey-filled state indicates the current automaton state.

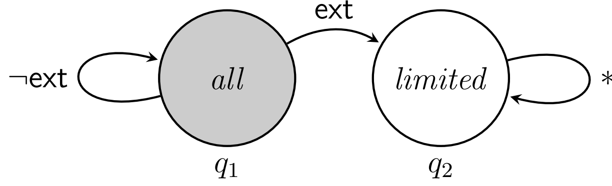

Figure 3: RIF automaton of document excerpting.

A second example sketches RIF automata that enforce integrity policies for a document management system.[8] Given is a set of *original* documents; these are trusted by all principals for all purposes. Operation $ext(D, parms)$ derives a new document by excerpting from document D according to $parms$. Because "creative" excerpting can be used to generate a document that has different meaning from the original, cautious principals should hesitate to use such *derived* documents for certain purposes. Using the terminology of Figure 1, excerpting causes deprecation.

One RIF automaton λ_D for supporting such a policy might employ a join semilattice of underlying restrictions $\langle\{all, limited\},\ \sqcup_R,\ \sqsubseteq_R\rangle$, where $limited \sqsubseteq_R all$ holds; these restrictions indicate whether a document is trusted for *all* purposes or for *limited* purposes. λ_D would employ a set $\mathcal{F}_{\text{Docs}}$ of reclassifiers that includes ext, which will correspond to excerpting operations. And λ_D would have two automaton states, where:

$$\rho_{\text{Docs}}(q) \triangleq \begin{cases} all & \text{if } q = q_1 \\ limited & \text{if } q = q_2 \end{cases}$$

Figure 3 gives a graphic depiction for a RIF automaton λ_D associated with an original document D; the formal definition of λ_D is:

$$\lambda_D \triangleq \langle\{q_1, q_2\}, \mathcal{F}_{\text{Docs}}, \delta_{\text{Docs}}, q_1, \rho_{\text{Docs}}\rangle$$

where

$$\delta_{\text{Docs}}(q, \mathsf{f}) \triangleq \begin{cases} q_2 & \text{if } q = q_1 \wedge \mathsf{f} = \mathsf{ext} \\ q & \text{otherwise} \end{cases}$$

RIF automaton $\lambda_D[q_1 \mapsto q_2]$ would be associated with any document produced by excerpting from D.

[8] This example is inspired by `TruDocs` [54].

Some applications might require a more refined basis for deciding whether a document should be trusted for a specific purpose. One obvious basis for making such trust assessments is the set of principals participating in the document's derivation. A RIF automaton can specify such policies. There would be an automaton state q_S for each set S of principals corresponding to a subset of *Prins*. And restrictions being associated with an automaton state q_S would depend on members of S. So the join semilattice of underlying restrictions is $\langle 2^{Prins}, \cup, \subseteq \rangle$. Transitions are facilitated by having a set $\mathcal{F}_{\text{Doc}}$ of reclassifiers contain an element ext_p for each principal p that invokes an excerpting operation. Here is the formal definition of an automaton λ_D that is associated with a document D that some principal $W \in Prins$ has written:

$$\lambda_D \triangleq \langle Q_{Prins}, \mathcal{F}_{\text{Doc}}, \delta_{\text{Doc}}, q_{\{W\}}, \rho_{\text{Doc}} \rangle$$

where

$$Q_{Prins} \triangleq \{q_S \mid S \in 2^{Prins}\}$$
$$\delta_{\text{Doc}}(q_S, \mathsf{exc}_p) \triangleq q_{S \cup \{p\}}$$
$$\rho_{\text{Doc}}(q_S) \triangleq S$$

4 A Security Policy for RIF Labels

Our security policy for programs that use RIF labels is obtained by extending termination-insensitive noninterference (TINI) [52]. TINI applies to programs comprising deterministic commands, so execution of a command $\mathcal{C}_1$ that is started in a memory $\mathcal{M}_1$ and terminates in a memory $\mathcal{M}_N$ can be described by giving a finite *trace*

$$\langle \mathcal{C}_1,\ \mathcal{M}_1 \rangle \to \langle \mathcal{C}_2,\ \mathcal{M}_2 \rangle \to \cdots \to \langle \bullet,\ \mathcal{M}_N \rangle \quad (17)$$

where each *state* $\langle \mathcal{C}_i,\ \mathcal{M}_i \rangle$ gives a command $\mathcal{C}_i$ and a memory $\mathcal{M}_i$. We write $\bullet$ in the final state of a trace to signify further execution is not possible, but $\bullet$ is not considered a command. Thus the trace for a terminating command $\mathcal{C}$ includes at least two states: one state having command $\mathcal{C}$ followed by one having $\bullet$.

Memories are represented by functions that map variables x appearing in any command to values $\mathcal{M}(x)$. These functions are then extended as usual for mapping ordinary expressions and reclassifying expressions.

$$\mathcal{M}(op(\mathcal{E}_1, \ldots, \mathcal{E}_n)) \triangleq op(\mathcal{M}(\mathcal{E}_1), \ldots, \mathcal{M}(\mathcal{E}_n))$$

$$\mathcal{M}([\mathcal{E}]_{\mathsf{f_1,f_2,\ldots,f_n}}) \ \triangleq \ \mathcal{M}(\mathcal{E})$$

An operational semantics for command execution can be given by a partial function Υ, where $\Upsilon(\mathcal{C}, \mathcal{M})$ equals the finite trace corresponding to an execution of $\mathcal{C}$ that terminates when started in memory $\mathcal{M}$, and $\Upsilon(\mathcal{C}, \mathcal{M})$ is undefined if that execution does not terminate.

4.1 Observations as a Threat Model

A mapping Γ, which associates variables with RIF labels, induces equivalence classes of memories that agree on the values in variables allowed to flow to $\lambda \in \Lambda$:

$$\begin{aligned}\mathcal{M} =_\lambda \mathcal{M}' \ \triangleq \ & dom(\mathcal{M}) = dom(\mathcal{M}') \\ & \wedge\ (\forall x \in dom(\mathcal{M})\colon\ \Gamma(x) \sqsubseteq_\Lambda \lambda \Rightarrow \mathcal{M}(x) = \mathcal{M}'(x))\end{aligned}$$

where $dom(\mathcal{M})$ is the domain of a memory $\mathcal{M}$.

Our *threat model* is intended for analyzing systems in which principals are co-resident and able to detect changes (subject to restrictions defined by RIF labels) being made to shared memory or other resources. We formalize this threat model relative to λ by constructing an *observation*

$$\mathcal{M}_{i+1} \ominus_\lambda \mathcal{M}_i \ \triangleq \ \{\langle x,\ \mathcal{M}_{i+1}(x)\rangle \mid \mathcal{M}_i(x) \neq \mathcal{M}_{i+1}(x) \wedge \Gamma(x) \sqsubseteq_\Lambda \lambda\}$$

for each *transition* $\langle \mathcal{C}_i,\ \mathcal{M}_i\rangle \rightarrow \langle \mathcal{C}_{i+1},\ \mathcal{M}_{i+1}\rangle$ that occurs. So observation $\mathcal{M}_{i+1} \ominus_\lambda \mathcal{M}_i$ gives the new value for each variable that was (i) changed by the transition and (ii) has a RIF label at most λ. It is easy to show:

$$\begin{aligned}\mathcal{M} =_\lambda \mathcal{M}' \ &\Rightarrow\ (\mathcal{M}' \ominus_\lambda \mathcal{M} \ = \ \emptyset) \\ \lambda \sqsubseteq_\Lambda \lambda' &\Rightarrow\ (\mathcal{M}' \ominus_\lambda \mathcal{M}) \subseteq (\mathcal{M}' \ominus_{\lambda'} \mathcal{M})\end{aligned}$$

For a RIF label λ, program execution described by a trace τ results in a sequence[9] $\tau|_\lambda$ of non-empty observations that are derived from the successive transitions in τ, and it induces an equivalence relation on traces

$$\tau =_\lambda \tau' \ \triangleq \ \tau|_\lambda = \tau'|_\lambda.$$

Sequences $\tau|_\lambda$ of observations are the basis for our threat model. Each principal p is assigned a set $\mathcal{R}(p)$ of restrictions. The threat model stipulates that a principal p sees changes to any variable x satisfying $\mathcal{R}(\Gamma(x)) \sqsubseteq_R \mathcal{R}(p)$. Thus, principal p sees those sequences $\tau|_\lambda$ of observations where $\mathcal{R}(\lambda) \sqsubseteq_R \mathcal{R}(p)$ holds.

[9] A formal definition of $\tau|_\lambda$ is thus given by the following, where $\rightharpoondown$ is used to delimit

4.2 Piecewise Noninterference

In the absence of reclassifications, an *illicit* λ-*flow* is witnessed when executing a command in states having different values of a variable with RIF label λ' satisfying $\lambda' \not\sqsubseteq_\Lambda \lambda$ results in differences in updates to any variable with RIF label λ or less restrictive. TINI prohibits illicit λ-flows for any join semilattice. To formalize an extension to TINI for accommodating reclassification under our threat model, we require some notation. For τ a non-empty finite trace or subtrace,

$$\langle \mathcal{C}_1,\ \mathcal{M}_1\rangle \to \langle \mathcal{C}_2,\ \mathcal{M}_2\rangle \to \cdots \to \langle \mathcal{C}_N,\ \mathcal{M}_N\rangle$$

and indices i and j satisfying $1 \leq i \leq j \leq N$, define

$$\begin{array}{rclrcl} \tau[i..] & \triangleq & \langle \mathcal{C}_i,\ \mathcal{M}_i\rangle \to \cdots \to \langle \mathcal{C}_N,\ \mathcal{M}_N\rangle & \tau[i].\mathcal{C} & \triangleq & \mathcal{C}_i \\ \tau[i..j] & \triangleq & \langle \mathcal{C}_i,\ \mathcal{M}_i\rangle \to \cdots \to \langle \mathcal{C}_j,\ \mathcal{M}_j\rangle & \tau[i].\mathcal{M} & \triangleq & \mathcal{M}_i \\ \tau[i] & \triangleq & \langle \mathcal{C}_i,\ \mathcal{M}_i\rangle & |\tau| & \triangleq & N \end{array}$$

In addition, we write $\tau.\mathcal{C}$ as an abbreviation for $\tau[1].\mathcal{C}$ and write $\tau.\mathcal{M}$ for $\tau[1].\mathcal{M}$. It is convenient to define $\tau[i..]$, $\tau[i..j]$, $\tau[i]$, $\tau[i].\mathcal{C}$, and $\tau[i].\mathcal{M}$ as equivalent to ϵ when $1 \leq i \leq j \leq N$ does not hold.

We start by formalizing prohibition of illicit λ-flows for commands $\widehat{\mathcal{C}}$ that do not contain reclassifying expressions. By definition, an illicit λ-flow is not present if executing $\widehat{\mathcal{C}}$ in memories $\mathcal{M}$ and $\mathcal{M}'$ satisfying $\mathcal{M} =_\lambda \mathcal{M}'$ does not result in different updates to a variable that has a RIF label λ or less restrictive.

$$(\forall \lambda, \mathcal{M}, \mathcal{M}', \tau, \tau'\colon\ \ \tau = \Upsilon(\widehat{\mathcal{C}}, \mathcal{M}) \ \wedge\ \tau' = \Upsilon(\widehat{\mathcal{C}}, \mathcal{M}') \ \ \Rightarrow\ \ NI(\lambda, \tau, \tau')) \quad (18)$$

where

$$NI(\lambda, \tau, \tau') \quad \triangleq \quad \tau.\mathcal{M} =_\lambda \tau'.\mathcal{M} \ \ \Rightarrow\ \ \tau =_\lambda \tau'.$$

---

observations in the resulting sequence.

$$\tau|_\lambda \ \triangleq\ \begin{cases} \epsilon \quad \text{if } \tau = \epsilon \text{ or } \tau = \langle \mathcal{C},\ \mathcal{M}\rangle \\ \epsilon \quad \text{if } \tau = \langle \mathcal{C},\ \mathcal{M}\rangle \to \langle \mathcal{C}',\ \mathcal{M}'\rangle \ \wedge\ \mathcal{M}' \ominus_\lambda \mathcal{M} = \emptyset \\ \mathcal{M}' \ominus_\lambda \mathcal{M} \qquad \text{if } \tau = \langle \mathcal{C},\ \mathcal{M}\rangle \to \langle \mathcal{C}',\ \mathcal{M}'\rangle \ \wedge\ \mathcal{M}' \ominus_\lambda \mathcal{M} \neq \emptyset \\ (\langle \mathcal{C}',\ \mathcal{M}'\rangle \to \tau')|_\lambda \qquad \text{if } \tau = \langle \mathcal{C},\ \mathcal{M}\rangle \to \langle \mathcal{C}',\ \mathcal{M}'\rangle \to \tau' \ \wedge\ \mathcal{M}' \ominus_\lambda \mathcal{M} = \emptyset \\ \mathcal{M}' \ominus_\lambda \mathcal{M} \ \rightharpoondown\ (\langle \mathcal{C}',\ \mathcal{M}'\rangle \to \tau')|_\lambda \\ \qquad\qquad \text{if } \tau = \langle \mathcal{C},\ \mathcal{M}\rangle \to \langle \mathcal{C}',\ \mathcal{M}'\rangle \to \tau' \ \wedge\ \mathcal{M}' \ominus_\lambda \mathcal{M} \neq \emptyset \end{cases}$$

When $NI(\lambda, \tau, \tau')$ is *false* then τ and τ' exhibit the illicit λ-flow.

Reclassification. Each transition $\langle \mathcal{C}_i,\ \mathcal{M}_i \rangle \rightarrow \langle \mathcal{C}_{i+1},\ \mathcal{M}_{i+1} \rangle$ in a trace involves evaluating some (possibly empty) set of expressions. The operational semantics of commands determines what those expressions are and how their values are computed. We posit that the operational semantics for each command $\mathcal{C}_i$ also includes a function $\Delta(\mathcal{C}_i)$ that gives the set of reclassifying expressions that command $\mathcal{C}_i$ evaluates as part of its first transition.

For example, with a simple imperative programming language and ordinary expressions $\mathcal{E}$ and $\mathcal{E}'$ we might expect to have

$$\Delta(x := \mathcal{E}) \;=\; \emptyset \tag{19}$$

$$\Delta(x := [\mathcal{E}]_{\bar{\mathsf{f}}}) \;=\; \{[\mathcal{E}]_{\bar{\mathsf{f}}}\} \tag{20}$$

$$\Delta(\mathbf{if}\ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \mathbf{then}\ x := [\mathcal{E}']_{\bar{\mathsf{f}}}) = \;\{[\mathcal{E}]_{\bar{\mathsf{f}}}\} \tag{21}$$

if we are assuming that assignments are executed as a single transition and that an **if** statement involves a first transition to evaluate its Boolean guard followed by other transitions for the **then** (or **else**) parts.

Handling Downgrades.

A reclassifying expression $[\mathcal{E}]_{\bar{\mathsf{f}}}$ is considered to perform a λ-*downgrade* if $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \sqsubseteq_\Lambda \lambda$ and $\Gamma(\mathcal{E}) \not\sqsubseteq_\Lambda \lambda$ hold. The set of expressions that satisfy the definition of a λ-downgrade and are evaluated by $\mathcal{C}_i$ to make its transition is given by:

$$\{\mathcal{E} \mid [\mathcal{E}]_{\bar{\mathsf{f}}} \in \Delta(\mathcal{C}_i) \;\wedge\; \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \sqsubseteq_\Lambda \lambda \;\wedge\; \Gamma(\mathcal{E}) \not\sqsubseteq_\Lambda \lambda\}$$

To define our security policy for such programs $\widehat{\mathcal{C}}$ independent of specific programming constructs, we posit that the language semantics includes a function[10] $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ that satisfies:

$$\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i) \;\subseteq\; \{\mathcal{E} \mid [\mathcal{E}]_{\bar{\mathsf{f}}} \in \Delta(\mathcal{C}_i) \;\wedge\; \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \sqsubseteq_\Lambda \lambda \;\wedge\; \Gamma(\mathcal{E}) \not\sqsubseteq_\Lambda \lambda\} \tag{22}$$

If $\mathcal{E} \in \Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ holds, then $\mathcal{C}_i$ is the *reference point*[11] for the corresponding λ-downgrade $[\mathcal{E}]_{\bar{\mathsf{f}}}$. The reference point signifies *where* in program $\widehat{\mathcal{C}}$ this λ-downgrade is performed and *what* information is being λ-downgraded—the evaluation of $\mathcal{E}$ at that program point $\mathcal{C}_i$.

---

[10]The $\widehat{\mathcal{C}}$ subscript in $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ enables this function to depend on an enclosing program $\widehat{\mathcal{C}}$ and/or the position of $\mathcal{C}_i$ within $\widehat{\mathcal{C}}$. For example, the language semantics might omit $\mathcal{E}$ from $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, x := [\mathcal{E}]_{\bar{\mathsf{f}}})$ if that assignment appears in the scope of an **if** having a Boolean guard $\mathcal{E}'$, where $\Gamma(\mathcal{E}') \not\sqsubseteq_\Lambda \Gamma(x)$ holds.

[11]See [41] for a thorough study of reference points for declassification.

Notice that, by definition, assigning the value of an expression in $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ to a variable having RIF label λ does not constitute an illicit λ-flow. So $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ characterizes illicit λ-flows that have been eliminated by fiat. Trade-offs associated with selecting the content of $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ are discussed at the end of this section.

A generalization of (18) suffices to accommodate λ-downgrades that occur in the first transition of a trace that results from executing command $\widehat{\mathcal{C}}$. $NI(\lambda, \tau, \tau')$ was defined above in terms of updates to any variable x satisfying $\Gamma(x) \sqsubseteq_\Lambda \lambda$. But differences in updates to x that arise when expressions in $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \widehat{\mathcal{C}})$ have different values are, by definition, not considered illicit λ-flows. Therefore, initial memory pairs $\mathcal{M}$ and $\mathcal{M}'$ that cause expressions in $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \widehat{\mathcal{C}})$ to have different values should be ignored in checking for indistinguishable observations [51]:

$$(\forall \lambda, \mathcal{M}, \mathcal{M}', \tau, \tau': \quad \tau = \Upsilon(\widehat{\mathcal{C}}, \mathcal{M}) \wedge \tau' = \Upsilon(\widehat{\mathcal{C}}, \mathcal{M}') \quad \Rightarrow \quad dNI(\lambda, \tau, \tau')) \tag{23}$$

where

$$dNI(\lambda, \tau, \tau') \quad \triangleq \quad (\forall \mathcal{E} \in \Delta^-_{\widehat{\mathcal{C}}}(\lambda, \tau.\mathcal{C}) \colon\ \tau.\mathcal{M}(\mathcal{E}) = \tau'.\mathcal{M}(\mathcal{E})) \ \Rightarrow\ NI(\lambda, \tau, \tau')$$

By construction in (23), $\tau.\mathcal{C} = \widehat{\mathcal{C}}$ and $\tau'.\mathcal{C} = \widehat{\mathcal{C}}$ hold. Therefore $\tau.\mathcal{C} = \tau'.\mathcal{C}$ and

$$\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \tau.\mathcal{C}) = \Delta^-_{\widehat{\mathcal{C}}}(\lambda, \tau'.\mathcal{C}) \tag{24}$$

hold too. Thus, ignoring downgraded expressions in the first transition of trace τ (i.e., elements of $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \tau.\mathcal{C})$), $dNI(\lambda, \tau, \tau')$ is also ignoring downgraded expressions in the first transition of trace τ' (i.e., elements of $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \tau'.\mathcal{C})$).

To handle traces that perform λ-downgrades after the first transition in $\widehat{\mathcal{C}}$, we partition traces into consecutive subtraces starting with transitions that are λ-downgrades. These subtraces are called λ-*pieces*. To formalize, we introduce operators $\overset{\rightarrow\lambda}{\tau}$ and $\overset{\lambda\rightarrow}{\tau}$. Subtrace $\overset{\rightarrow\lambda}{\tau}$ is the initial λ-piece of trace τ and subtrace $\overset{\lambda\rightarrow}{\tau}$ is the rest. Thus, $\overset{\rightarrow\lambda}{\tau}$ and $\overset{\lambda\rightarrow}{\tau}$ satisfy the following two properties, for traces and subtraces τ satisfying $2 \leq |\tau|$.

- Trace τ splits at a λ-downgrade into $\overset{\rightarrow\lambda}{\tau}$ and $\overset{\lambda\rightarrow}{\tau}$:

$$\begin{array}{l} (\overset{\rightarrow\lambda}{\tau} = \tau \ \wedge\ \overset{\lambda\rightarrow}{\tau} = \epsilon) \\ \vee\ \ (\exists 1 < i < |\tau| \colon\ \overset{\rightarrow\lambda}{\tau} = \tau[1..i] \ \wedge\ \overset{\lambda\rightarrow}{\tau} = \tau[i..] \ \wedge\ \Delta^-_{\widehat{\mathcal{C}}}(\lambda, \tau[i].\mathcal{C}) \neq \emptyset) \end{array} \tag{25}$$

– There is no λ-downgrade after the first transition within λ-piece $\overset{\rightarrow\lambda}{\tau}$:

$$(\forall 1 < i < |\overset{\rightarrow\lambda}{\tau}|:\ \Delta^{-}_{\widehat{\mathcal{C}}}(\lambda,\ \overset{\rightarrow\lambda}{\tau}[i].\mathcal{C}) = \emptyset) \tag{26}$$

Here is a schematic representation of $\overset{\rightarrow\lambda}{\tau}$ and $\overset{\lambda\rightarrow}{\tau}$:

$$\tau = \underbrace{\langle \widehat{\mathcal{C}},\ \mathcal{M}\rangle \xrightarrow{*} \overbrace{\langle C_i,\ \mathcal{M}_i\rangle}^{}}_{\overset{\rightarrow\lambda}{\tau}} \overbrace{\xrightarrow{*} \langle C_n,\ \mathcal{M}_n\rangle}^{\overset{\lambda\rightarrow}{\tau}} \quad \text{where} \quad \Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i) \neq \emptyset$$

Notice that by repeatedly isolating the first λ-piece of the remainder, a trace τ is partitioned into consecutive λ-pieces.

Two λ-pieces $\overset{\rightarrow\lambda}{\tau}$ and $\overset{\rightarrow\lambda}{\tau'}$ are witnesses of an illicit λ-flow if executing the same command in memories $\mathcal{M}$ and $\mathcal{M}'$ that satisfy $\mathcal{M} =_\lambda \mathcal{M}'$ and agree on values of expressions that have already been λ-downgraded, leads to having (i) different commands in their last states, and/or (ii) different updates to a variable x where $\Gamma(x) \sqsubseteq_\Lambda \lambda$ holds. This is because (i) implies that the next λ-downgrades that are reached by these two λ-pieces are not found at the same program point. So, the reference point of these λ-downgrades is being compromised since it depends on information that is not allowed to flow to λ. When (ii) occurs, updates to x have been compromised because they are influenced by information that is not allowed to flow to λ. In sum, (i) and (ii) expose compromises of *what* is allowed to flow to λ and *where* in the program.

In this paper, we avoid these compromises to (i) and (ii) by rejecting certain pairs of λ-pieces $\overset{\rightarrow\lambda}{\tau}$ and $\overset{\rightarrow\lambda}{\tau'}$:

$$\begin{aligned}
&(\overset{\rightarrow\lambda}{\tau}.\mathcal{C} = \overset{\rightarrow\lambda}{\tau'}.\mathcal{C} \quad \wedge \quad \overset{\rightarrow\lambda}{\tau}.\mathcal{M} =_\lambda \overset{\rightarrow\lambda}{\tau'}.\mathcal{M}) \\
&\quad \wedge \quad (\forall \mathcal{E} \in \Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \tau.\mathcal{C}):\ \tau.\mathcal{M}(\mathcal{E}) = \tau'.\mathcal{M}(\mathcal{E})) \\
&\Rightarrow \quad (\tau[\,|\overset{\rightarrow\lambda}{\tau}|\,].\mathcal{C} = \tau[\,|\overset{\rightarrow\lambda}{\tau'}|\,].\mathcal{C} \ \wedge\ \overset{\rightarrow\lambda}{\tau} =_\lambda \overset{\rightarrow\lambda}{\tau'})
\end{aligned} \tag{27}$$

By iterating through successive corresponding λ-pieces in two traces τ and τ', the following formula uses the approach embodied by (27) to identify λ-pieces that evidence an illicit λ-flow and thus provides the formal guarantee associated with downgrades.

Definition (*dpNI*)**.**

$$
\begin{aligned}
dpNI(\lambda,\tau,\tau') \;\triangleq\; (\tau \neq \epsilon \;\wedge\; \tau' \neq \epsilon \;\wedge\; \overset{\rightarrow\lambda}{\tau}.\mathcal{C} = \overset{\rightarrow\lambda}{\tau'}.\mathcal{C} \;\wedge\; \overset{\rightarrow\lambda}{\tau}.\mathcal{M} =_\lambda \overset{\rightarrow\lambda}{\tau'}.\mathcal{M}) \\
\wedge\; (\forall \mathcal{E} \in \Delta^-_{\widehat{\mathcal{C}}}(\lambda, \tau.\mathcal{C})\colon\; \tau.\mathcal{M}(\mathcal{E}) = \tau'.\mathcal{M}(\mathcal{E}))) \\
\Rightarrow\; (\tau[\,|\overset{\rightarrow\lambda}{\tau}|\,].\mathcal{C} = \tau[\,|\overset{\rightarrow\lambda}{\tau'}|\,].\mathcal{C} \;\wedge\; \overset{\rightarrow\lambda}{\tau} =_\lambda \overset{\rightarrow\lambda}{\tau'} \\
\wedge\; dpNI(\lambda, \overset{\lambda\rightarrow}{\tau}, \overset{\lambda\rightarrow}{\tau'}))
\end{aligned}
\tag{28}
$$

Notice, if traces τ and τ' each are a single λ-piece (i.e., $\tau = \overset{\rightarrow\lambda}{\tau}$ and $\tau' = \overset{\rightarrow\lambda}{\tau'}$ for all λ) then $dpNI(\lambda,\tau,\tau')$ is equivalent to (27), since $dpNI(\lambda, \overset{\lambda\rightarrow}{\tau}, \overset{\lambda\rightarrow}{\tau'})$ in the consequent of (28) would be $dpNI(\lambda,\epsilon,\epsilon)$, which trivially holds.

A characterization similar to (23) now handles downgrades that appear anywhere in traces that result from executing a command $\mathcal{C}$.

$$
(\forall \lambda, \mathcal{M}, \mathcal{M}', \tau, \tau'\colon\;\; \tau = \Upsilon(\mathcal{C},\mathcal{M}) \;\wedge\; \tau' = \Upsilon(\mathcal{C},\mathcal{M}') \;\Rightarrow\; dpNI(\lambda,\tau,\tau')) \tag{29}
$$

Downgrade Examples. Some example programs illustrate nuances of (29). These programs use RIF labels $\Lambda_{\mathsf{LH}} \triangleq \{\mathsf{L},\mathsf{H}\}$ with $\mathsf{L} \sqsubset_\Lambda \mathsf{H}$ and reclassifiers $\mathcal{F} \triangleq \{\downarrow,\uparrow\}$ satisfying $\mathcal{T}(\mathsf{H},\downarrow) \triangleq \mathsf{L}$ and $\mathcal{T}(\mathsf{L},\uparrow) \triangleq \mathsf{H}$. Assume that $\Gamma(low) = \Gamma(low') = \mathsf{L}$ and $\Gamma(high) = \Gamma(high') = \mathsf{H}$.

The first example program assigns in $\mathcal{C}_3$ a value with RIF label H to a variable with RIF label L without use of a reclassifying expression and, thus, would seem to exhibit an illicit L-flow.

$$
\begin{aligned}
&\mathcal{C}_1\colon\;\; low := [\, op(high)\,]_\downarrow; \\
&\mathcal{C}_2\colon\;\; low' := [\, op(high')\,]_\downarrow; \\
&\mathcal{C}_3\colon\;\; low := op(high)
\end{aligned}
\tag{30}
$$

Traces for program (30) comprise two L-pieces.[12] One L-piece starts with command $\mathcal{C}_1$, and the other L-piece starts with command $\mathcal{C}_2$ (and includes $\mathcal{C}_3$), due to the following.

$$
\Delta^-_{(30)}(\mathsf{L},\mathcal{C}_1) = \{op(high)\} \quad \Delta^-_{(30)}(\mathsf{L},\mathcal{C}_2) = \{op(high')\} \quad \Delta^-_{(30)}(\mathsf{L},\mathcal{C}_3) = \emptyset.
$$

[12] Each trace is also a single H-piece. For that case, $dpNI(\mathsf{H},\tau,\tau')$ holds because $\tau.\mathcal{M} =_{\mathsf{H}} \tau'.\mathcal{M}$ is equivalent to $\tau.\mathcal{M} = \tau'.\mathcal{M}$. The same applies to example programs (31) and (32) to come.

Checking, we find (29) is satisfied despite our earlier premonition about $\mathcal{C}_3$. A close look shows why the flow to *low* in assignment $\mathcal{C}_3$ actually ought to be allowed, as characterization (29) does: $\mathcal{C}_3$ is assigning an L-downgraded value, since the value of $op(high)$ in the right hand side of $\mathcal{C}_3$ was an L-downgraded value in $\mathcal{C}_1$ and has not been changed since.

Program (30) highlights a difference in how downgrades are treated by PWNI as compared with proposals using *memory-reset* (e.g., [12]). Under PWNI, a downgraded value remains so for the remainder of the trace; with memory-reset, a downgraded value remains so only until the next reclassification operation. So, program (30) would be considered to exhibit a leak according to the memory-reset approach. Gradual release [3], which does not rely on memory-reset, deems program (30) secure, like PWNI does.

A second example program changes in $\mathcal{C}_2$ the value in *high* after the L-downgrade in $\mathcal{C}_1$.

$$\begin{array}{ll} \mathcal{C}_1\colon & low := [op(high)]_{\downarrow}\,; \\ \mathcal{C}_2\colon & high := high'; \\ \mathcal{C}_3\colon & low := op(high) \end{array} \tag{31}$$

Traces for (31) comprise a single L-piece that starts with command $\mathcal{C}_1$ because:

$$\Delta^{-}_{(31)}(\mathsf{L},\mathcal{C}_1) = \{op(high)\} \qquad \Delta^{-}_{(31)}(\mathsf{L},\mathcal{C}_2) = \emptyset \qquad \Delta^{-}_{(31)}(\mathsf{L},\mathcal{C}_3) = \emptyset$$

Program (31) does not satisfy (29), because traces τ and τ' exist that generate observations that are not (but should be) indistinguishable. This is because there exist memories $\mathcal{M}$ and $\mathcal{M}'$ satisfying $\mathcal{M} =_{\mathsf{L}} \mathcal{M}'$ and $\mathcal{M}(high') \neq \mathcal{M}'(high')$. When alternative executions of $\mathcal{C}_1$ are started in these two memories, $\mathcal{C}_3$ generates different updates to *low*. And having (29) not satisfied for this program is what we should desire—the value in *high* when $\mathcal{C}_3$ executes is not the value that was L-downgraded, so in program (31) a value with RIF label H that has not been L-downgraded is being used to update a variable with RIF label L.

We also have examined *dpNI* with respect to downgrades for the examples used by Askarov and Sabelfeld [4] when discussing *localized delimited release*. Both *dpNI* and localized delimited release yield the same judgments for these examples, although in general the two policies are incomparable.

A final program illustrates the role of conjunct $\tau[\,|\overset{\rightarrow\lambda}{\tau}|\,].\mathcal{C} = \tau[\,|\overset{\rightarrow\lambda}{\tau'}|\,].\mathcal{C}$ in

the consequent of $dpNI(\lambda, \tau, \tau')$.

$$\begin{array}{lll} \mathcal{C}_1\text{: } \textbf{if } high' > 0 \textbf{ then} & \mathcal{C}_2\text{: } high := 3 \\ & \mathcal{C}_3\text{: } low := [high + 2]_{\downarrow} \\ \qquad\qquad\qquad\qquad \textbf{else} & \mathcal{C}_4\text{: } high := 4 \\ & \mathcal{C}_5\text{: } low := [high \bmod 2]_{\downarrow} \end{array} \tag{32}$$

Consider traces $\tau = \Upsilon(\mathcal{C}_1, \mathcal{M})$ and $\tau' = \Upsilon(\mathcal{C}_1, \mathcal{M}')$, where the following hold: $\mathcal{M} =_{\mathsf{L}} \mathcal{M}'$, $\mathcal{M}(high' > 0) = true$, and $\mathcal{M}'(high' > 0) = false$. Trace τ comprises a first L-piece that starts with $\mathcal{C}_1$ (and includes $\mathcal{C}_2$), followed by a second L-piece that starts with $\mathcal{C}_3$; trace τ' has a first L-piece that starts with $\mathcal{C}_1$ (and includes $\mathcal{C}_4$), followed by a second L-piece that starts with $\mathcal{C}_5$:

$$\tau = \underbrace{\langle C_1,\ \mathcal{M}\rangle \to \langle C_2,\ \mathcal{M}_2\rangle \to \overbrace{\langle C_3,\ \mathcal{M}_3\rangle}}_{\text{L-piece}}{}^{\text{L-piece}} \to \langle \bullet,\ \mathcal{M}_t\rangle$$

$$\tau' = \underbrace{\langle C_1,\ \mathcal{M}'\rangle \to \langle C_4,\ \mathcal{M}_4\rangle \to \overbrace{\langle C_5,\ \mathcal{M}_5\rangle}}_{\text{L-piece}}{}^{\text{L-piece}} \to \langle \bullet,\ \mathcal{M}_{t'}\rangle$$

Program (32) does not satisfy (29) because both $\tau[\,|\overrightarrow{\tau}^{\lambda}|\,].\mathcal{C} = \mathcal{C}_3$ and $\tau[\,|\overrightarrow{\tau'}^{\lambda}|\,].\mathcal{C} = \mathcal{C}_5$ hold, so conjunct $\tau[\,|\overrightarrow{\tau}^{\lambda}|\,].\mathcal{C} = \tau[\,|\overrightarrow{\tau'}^{\lambda}|\,].\mathcal{C}$ in the consequent of $dpNI(\mathsf{L}, \tau, \tau')$ does not hold. And having (29) not satisfied is what we should desire, because the choice of $\mathcal{C}_3$ or $\mathcal{C}_5$ leaks information about the value of $high'$ to low.

Instead of command equality (i.e., $\tau[\,|\overrightarrow{\tau}^{\lambda}|\,].\mathcal{C} = \tau[\,|\overrightarrow{\tau'}^{\lambda}|\,].\mathcal{C}$), we could have extended the *bisimulation* used in [4] to handle downgrades of arbitrary expressions during execution. However, with this bisimulation, we would risk accepting insecure programs. This is because this bisimulation does not seem to set intermediate reference points for the downgrades. For example, consider program (32), which is insecure. A bisimulation that supports downgrades with reference points on intermediate states would accept the program as secure. The case to consider concerns the execution steps $\langle C_3,\ M_3\rangle \to \langle \bullet,\ M_t\rangle$ and $\langle C_5,\ M_5\rangle \to \langle \bullet,\ M_{t'}\rangle$, where $M_3 =_{\mathsf{L}} M_5$. Adapting the bisimulation definition in [4] to support downgrades with reference points on intermediate states, these two execution steps constitute a bisimulation if

(i) $M_3([high+2]_{\downarrow}) = M_5([high+2]_{\downarrow})$ if and only if $M_3([high \bmod 2]_{\downarrow}) = M_5([high \bmod 2]_{\downarrow})$, and

(ii) if $M_3([high+2]_{\downarrow}) = M_5([high+2]_{\downarrow})$, then $M_t =_{\mathsf{L}} M_{t'}$ and states $\langle \bullet,\ M_t \rangle$, $\langle \bullet,\ M_{t'} \rangle$ are bisimilar.

Condition (i) holds because, for any such reachable memories M_3 and M_5, both equalities are false. Because these equalities are false, condition (ii) is vacuously true. So these execution steps are (vacuously) bisimulations, and thus this insecure program is accepted as secure. Another form of bisimulation might fare better, but we have not found one.

Handling Upgrades. A reclassifying expression $[\mathcal{E}]_{\bar{\mathsf{f}}}$ is considered to perform a λ-*upgrade* if $\Gamma(\mathcal{E}) \sqsubseteq_\Lambda \lambda$ and $\Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq_\Lambda \lambda$ hold. Characterization (29) does not detect illicit λ-flows caused by λ-upgrades. Consider, for example, a program comprising assignment

$$low := [low']_{\uparrow} \tag{33}$$

that uses RIF labels from Λ_{LH}. Assuming $\Gamma(low) = \mathsf{L}$ and $\Gamma(low') = \mathsf{L}$ hold, this program satisfies (29). Yet the program exhibits an illicit L-flow: differences in the initial value of an L-upgraded expression ($[low']_{\uparrow}$) with RIF label H result in differences in updates to a variable (low) with RIF label L where $\Gamma([low']_{\uparrow}) \not\sqsubseteq_\Lambda \Gamma(low)$ holds.

In checking for evidence of illicit λ-flows, (29) compares traces τ and τ' that differ in initial values of variables whose RIF labels λ' satisfy $\lambda' \not\sqsubseteq_\Lambda \lambda$. That set of comparisons covers some expressions whose RIF labels λ' satisfy $\lambda' \not\sqsubseteq_\Lambda \lambda$ but it does not cover all expressions—it ignores expressions that involve λ-upgrades.

A programming language that supports RIF labels will provide syntax that allows programmers to specify λ-upgrades by using reclassifying expressions. To be independent of language constructs, we posit that the language semantics includes a function $\Delta^+_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ that satisfies:

$$\Delta^+_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i) \ \supseteq\ \{\mathcal{E} \mid [\mathcal{E}]_{\bar{\mathsf{f}}} \in \Delta(\mathcal{C}_i) \ \wedge\ \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq_\Lambda \lambda \ \wedge\ \Gamma(\mathcal{E}) \sqsubseteq_\Lambda \lambda\} \tag{34}$$

So $\Delta^+_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ contains all expressions that satisfy the definition of a λ-upgrade and are evaluated by $\mathcal{C}_i$ (within $\widehat{\mathcal{C}}$) to make its transition.

Since (29) correctly identifies illicit λ-flows from variables x for which $\Gamma(x) \not\sqsubseteq_\Lambda \lambda$ holds, we transform λ-upgraded expressions to such variables as a

way to identify illicit λ-flows for λ-upgraded expressions. Define translation $\mathbb{T}(\lambda, \widehat{\mathcal{C}})$ to be the command that results from, in every subcommand $\mathcal{C}_i$ of $\widehat{\mathcal{C}}$, substituting an expression $next(h_{\mathcal{E}})$ for every expression $[\mathcal{E}]_{\bar{\mathsf{f}}} \in \Delta(\mathcal{C}_i)$ where $\mathcal{E} \in \Delta^{+}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ and $\Gamma(next(h_{\mathcal{E}})) = \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}})$ hold.[13] Here, $h_{\mathcal{E}}$ is a fresh variable that stores a list of values that $\mathcal{E}$ could assume, and successive evaluations of $next(h_{\mathcal{E}})$ return successive elements of that list. If a program $\widehat{\mathcal{C}}$ exhibits an illicit λ-flow from a λ-upgraded expression $\mathcal{E}$ then, by construction, $\mathbb{T}(\lambda, \widehat{\mathcal{C}})$ exhibits an illicit λ-flow from $next(h_{\mathcal{E}})$. Moreover, because $\mathbb{T}(\lambda, \widehat{\mathcal{C}})$ contains no λ-upgraded expressions, it exhibits no illicit λ-flows from upgraded λ-expressions *per se*. That means we can use (29) to check for illicit λ-flows in $\widehat{\mathcal{C}}$ by checking $\mathbb{T}(\lambda, \widehat{\mathcal{C}})$ for illicit λ-flows. A formal definition of $\mathbb{T}(\lambda, \widehat{\mathcal{C}})$ can be found in the Appendix B.

The resulting characterization of *piecewise noninterference* (PWNI) for a program $\widehat{\mathcal{C}}$ employs $dpNI(\lambda, \tau, \tau')$—defined in (28)—by extending (29) to handle λ-flows from λ-upgraded expressions.

Definition (Piecewise Noninterference)**.**

$$\begin{array}{l} PWNI(\widehat{\mathcal{C}}) \;\triangleq \\ \quad (\forall \lambda, \mathcal{M}, \mathcal{M}', \tau, \tau'\colon \quad \tau = \Upsilon(\,\mathbb{T}(\lambda, \widehat{\mathcal{C}}),\, \mathcal{M}) \;\wedge\; \tau' = \Upsilon(\,\mathbb{T}(\lambda, \widehat{\mathcal{C}}),\, \mathcal{M}') \\ \qquad\qquad\qquad\qquad\quad \Rightarrow \;\; dpNI(\lambda, \tau, \tau')) \end{array} \tag{35}$$

The election example and the conference-management example of §1 both satisfy this definition when coded in a straightforward manner.

Upgrade Examples. $PWNI(\widehat{\mathcal{C}})$ as defined in (35) considers executions of a translation in which L-upgraded expressions $\mathcal{E}$ are replaced by different fresh variables $h_{\mathcal{E}}$. So, equivalent expressions within an execution and in different executions of the original program may take different values in executions of the translation. Here is an illustration, where RIF labels are from Λ_{LH}.

$$\mathcal{C}_1\colon\; \textbf{if}\ [low = low]_{\uparrow}\ \textbf{then}\ \ \mathcal{C}_2\colon low := 1\ \textbf{else}\ \ \mathcal{C}_3\colon low := 2 \tag{36}$$

$PWNI(\mathcal{C}_1)$ creates translated program $\mathbb{T}(\lambda, \mathcal{C}_1)$,

$$\mathcal{C}_1\colon\; \textbf{if}\ next(h_{low=low})\ \textbf{then}\ \ \mathcal{C}_2\colon low := 1\ \textbf{else}\ \ \mathcal{C}_3\colon low := 2 \tag{37}$$

where $\Gamma(next(h_{low=low})) = \mathsf{H}$ holds. Boolean guard $[low = low]_{\uparrow}$ in (36) is always *true*, so execution of (36) produces no traces in which $\mathcal{C}_3$ executes. In

[13]See (36) and (37) below for an example of the translation.

translated program (37), Boolean guard $next(h_{low=low})$ takes values *true* and *false*, so there are traces of (37) where $\mathcal{C}_3$ executes as well as traces where $\mathcal{C}_2$ executes. Thus, $PWNI(\mathcal{C}_1)$ is not satisfied for (37) due to the illicit L-flow from upgraded expression $[low = low]_\uparrow$ to low, even though in program (36) upgraded expression $[low = low]_\uparrow$ never actually flows to low. One way to circumvent this conservatism is to restrict the set of values that a fresh variable can take. In particular, that set can be limited to containing values that the upgrading expression can take during program execution.

Assessing PWNI . Whether a program satisfies $PWNI(\widehat{\mathcal{C}})$ depends, in part, on flexibility that (22) allows in the definition of $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ and that (34) allows in the definition of $\Delta^+_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$.

- $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ is constrained only by (22), so $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ may omit expressions that cause λ-downgrades. With fewer expressions in $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$:
  - There are more pairs of memories from which execution results in a pair of λ-pieces whose subtraces $\overset{\rightarrow\lambda}{\tau}$ and $\overset{\rightarrow\lambda}{\tau'}$ must satisfy (27). So the omissions may cause $PWNI(\widehat{\mathcal{C}})$ to reject programs that do not actually exhibit illicit λ-flows.
  - Fewer λ-pieces need to be checked by evaluating $dpNI(\lambda, \tau, \tau')$. So the omissions may cause $PWNI(\widehat{\mathcal{C}})$ to accept more programs that do not exhibit illicit λ-flows.
- $\Delta^+_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ is constrained only by (34), so $\Delta^+_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ may include expressions that do not cause λ-upgrades. The additional expressions $\mathcal{E}$ are replaced in each $\mathcal{C}_i$ by expressions involving list $h_{\mathcal{E}}$. That means there could be more pairs of memories (i.e., those that differ in $h_{\mathcal{E}}$) from which execution results in a pair of λ-pieces whose subtraces $\overset{\rightarrow\lambda}{\tau}$ and $\overset{\rightarrow\lambda}{\tau'}$ must satisfy (27). So the additions may cause $PWNI(\widehat{\mathcal{C}})$ to reject programs that do not exhibit illicit λ-flows.

Absent a widely accepted semantic definition for information flow in the presence of reclassifications—and none has yet appeared in the literature—examples must suffice as a starting point for any discussion of whether definitions being given for $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ and $\Delta^+_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ are sensible.

The larger question is whether PWNI constitutes a reasonable approach to defining secure programs when reclassifications are possible, independent

of the specific choices for $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ and $\Delta^+_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$. Here, besides considering various examples, the declassification principles described in Sabelfeld and Sand [53] offer criteria for evaluation. The principles are:

- *Conservativity*: In programs that do not involve reclassification, satisfying PWNI implies satisfying noninterference. Thus, PWNI satisfies conservativity.

- *Non-occlusion*: PWNI satisfies non-occlusion because declassifications do not allow other secret values to be leaked. The syntactic equality on commands at the end of λ-pieces, which some might find too conservative, is here seen as ensuring PWNI satisfies non-occlusion.

- *Semantic Consistency*: PWNI does not satisfy semantic consistency. Transformation of a declassification-free program in a way that preserves semantics might not preserve PWNI. However, we know of no policy that satisfies semantic consistency and also (i) specifies a *where* dimension for downgrades, (ii) has the capability to allow downgrades of expressions in the midst of the computation (rather than restricting downgrades to initial values), (iii) does not employ memory-reset (which was already shown problematic for program (30) above), and (iv) satisfies Conservativity and Non-occulusion. An interesting open research problem is to determine whether (i)–(iv) necessarily implies that semantic consistency cannot be satisfied.

- *Monotonicity of Release*: PWNI does not satisfy monotonicity of release, because adding declassification annotations might render a secure program insecure. The example used in [4], which shows that localized delimited release does not satisfy monotonicity of release, applies to PWNI as well.

5 Static Enforcement of PWNI

Type-checking allows static enforcement of $PWNI(\widehat{\mathcal{C}})$ when $\widehat{\mathcal{C}}$ is written in a programming language having a type system based on some checkable class of RIF labels. A simple imperative language provides a vehicle for demonstration.

$$
\begin{array}{rcl}
\mathcal{E} & ::= & \nu \mid x \mid op(\mathcal{E}_1, \ldots, \mathcal{E}_n) \\
\mathcal{C} & ::= & \textbf{skip} \\
& \mid & x := [\mathcal{E}]_{\bar{\mathsf{f}}} \\
& \mid & \textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } \mathcal{C}_t \textbf{ else } \mathcal{C}_e \\
& \mid & \textbf{while } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ do } \mathcal{C}_t \\
& \mid & \mathcal{C}_1; \mathcal{C}_2
\end{array}
$$

Figure 4: Syntax of simple imperative language

Language and Semantics. Figure 4 gives a syntax of a simple programming language for defining commands. There, ν ranges over constants, x ranges over program variables, $\mathcal{E}$, $\mathcal{E}_1$, $\mathcal{E}_2, \ldots$ range over ordinary expressions, and $\mathcal{C}_t$, $\mathcal{C}_e$, $\mathcal{C}_1$, $\mathcal{C}_2$, $\ldots$ range over commands. Note, allowing only reclassifying expressions (rather than ordinary expressions) in the language syntax for commands is not a limitation—due to (5), identity reclassifier ϵ is handled by every RIF label and reclassifying expression $[\mathcal{E}]_\epsilon$ has the same value and RIF label as ordinary expression $\mathcal{E}$.

Figure 5 gives an operational semantics for the programming language of Figure 4. We write $\mathcal{M}[x \mapsto \nu]$ there to denote a mapping that is identical to $\mathcal{M}$ except $\mathcal{M}(x) = \nu$. The rules in Figure 5 define partial function $\Upsilon(\mathcal{C}, \mathcal{M})$.

The final part of the semantics for this programming language is definitions for $\Delta(\mathcal{C}_i)$, $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$, and $\Delta^+_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$. These are given in Figure 6 through Figure 8. In defining $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$, we write $lhs(\mathcal{C}_i)$ to denote the set of target variables in assignments appearing in a command $\mathcal{C}_i$.

- $\Delta^-_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ excludes λ-downgrades that cannot influence the value being assigned to a variable x where $\Gamma(x) \sqsubseteq_\Lambda \lambda$ holds. No illicit λ-flow is possible without such an assignment.

- $\Delta^+_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$ for **if** and **while** is the smallest set that satisfies (34). To exclude λ-upgrade guards because an assignment does not appear in the body of an **if** or **while** (as done in the definition of $\Delta^+_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$) would invalidate (34) by decreasing the size of $\Delta^+_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_i)$.

Typing Rules. Figure 9 gives rules to associate a type with each expression. The rules for ordinary expressions are standard. Expr-T instantiates (9); AnnExpr-T is based on (10).

$$\textsc{Skip}: \frac{}{\langle \textbf{skip},\ \mathcal{M}\rangle \rightarrow \langle \bullet,\ \mathcal{M}\rangle}$$

$$\textsc{Asgn}: \frac{\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \nu}{\langle x := [\mathcal{E}]_{\bar{\mathsf{f}}}\,,\ \mathcal{M}\rangle \rightarrow \langle \bullet,\ \mathcal{M}[x \mapsto \nu]\rangle}$$

$$\textsc{Brch1}: \frac{\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathit{true}}{\langle \textbf{if}\ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \textbf{then}\ \mathcal{C}_t\ \textbf{else}\ \mathcal{C}_e,\ \mathcal{M}\rangle \rightarrow \langle \mathcal{C}_t,\ \mathcal{M}\rangle}$$

$$\textsc{Brch2}: \frac{\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathit{false}}{\langle \textbf{if}\ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \textbf{then}\ \mathcal{C}_t\ \textbf{else}\ \mathcal{C}_e,\ \mathcal{M}\rangle \rightarrow \langle \mathcal{C}_e,\ \mathcal{M}\rangle}$$

$$\textsc{Loop1}: \frac{\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathit{true}}{\langle \textbf{while}\ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \textbf{do}\ \mathcal{C}_t,\ \mathcal{M}\rangle \rightarrow \langle \mathcal{C}_t; \textbf{while}\ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \textbf{do}\ \mathcal{C}_t,\ \mathcal{M}\rangle}$$

$$\textsc{Loop2}: \frac{\mathcal{M}([\mathcal{E}]_{\bar{\mathsf{f}}}) = \mathit{false}}{\langle \textbf{while}\ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \textbf{do}\ \mathcal{C}_t,\ \mathcal{M}\rangle \rightarrow \langle \bullet,\ \mathcal{M}\rangle}$$

$$\textsc{Seq1}: \frac{\langle \mathcal{C}_1,\ \mathcal{M}\rangle \rightarrow \langle \bullet,\ \mathcal{M}_1\rangle}{\langle \mathcal{C}_1; \mathcal{C}_2,\ \mathcal{M}\rangle \rightarrow \langle \mathcal{C}_2,\ \mathcal{M}_1\rangle}$$

$$\textsc{Seq2}: \frac{\langle \mathcal{C}_1,\ \mathcal{M}\rangle \rightarrow \langle \mathcal{C},\ \mathcal{M}_1\rangle \quad \mathcal{C} \neq \bullet}{\langle \mathcal{C}_1; \mathcal{C}_2,\ \mathcal{M}\rangle \rightarrow \langle \mathcal{C}; \mathcal{C}_2,\ \mathcal{M}_1\rangle}$$

Figure 5: Operational Semantics

$$\begin{aligned}
\Delta(\textbf{skip}) &\triangleq \emptyset \\
\Delta(x := [\mathcal{E}]_{\bar{\mathsf{f}}}) &\triangleq \{[\mathcal{E}]_{\bar{\mathsf{f}}}\} \\
\Delta(\textbf{if}\ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \textbf{then}\ \mathcal{C}_t\ \textbf{else}\ \mathcal{C}_e) &\triangleq \{[\mathcal{E}]_{\bar{\mathsf{f}}}\} \\
\Delta(\textbf{while}\ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \textbf{do}\ \mathcal{C}_w\ ) &\triangleq \{[\mathcal{E}]_{\bar{\mathsf{f}}}\} \\
\Delta(\mathcal{C}_1; \mathcal{C}_2) &\triangleq \Delta(\mathcal{C}_1)
\end{aligned}$$

Figure 6: Definition of $\Delta(\mathcal{C}_i)$

$$\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathbf{skip}) \triangleq \emptyset$$

$$\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, x := [\mathcal{E}]_{\bar{\mathsf{f}}}) \triangleq \begin{cases} \{\mathcal{E}\} & \text{if } \Gamma(\mathcal{E}) \not\sqsubseteq_{\Lambda} \lambda \ \wedge \ \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \sqsubseteq_{\Lambda} \lambda \ \wedge \ \Gamma(x) \sqsubseteq_{\Lambda} \lambda \\ \emptyset & \text{otherwise} \end{cases}$$

$$\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathbf{if}\ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \mathbf{then}\ \mathcal{C}_t\ \mathbf{else}\ \mathcal{C}_e) \triangleq \begin{cases} \{\mathcal{E}\} & \text{if } \Gamma(\mathcal{E}) \not\sqsubseteq_{\Lambda} \lambda \ \wedge \ \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \sqsubseteq_{\Lambda} \lambda \\ & \quad \wedge (\exists x \in \mathit{lhs}(\mathcal{C}_t) \cup \mathit{lhs}(\mathcal{C}_e) \colon \Gamma(x) \sqsubseteq_{\Lambda} \lambda) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathbf{while}\ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \mathbf{do}\ \mathcal{C}_w) \triangleq \begin{cases} \{\mathcal{E}\} & \text{if } \Gamma(\mathcal{E}) \not\sqsubseteq_{\Lambda} \lambda \ \wedge \ \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \sqsubseteq_{\Lambda} \lambda \\ & \quad \wedge (\exists x \in \mathit{lhs}(\mathcal{C}_w) \colon \Gamma(x) \sqsubseteq_{\Lambda} \lambda) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_1; \mathcal{C}_2) \triangleq \Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_1)$$

Figure 7: Definition of $\Delta^{-}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C})$

$$\Delta^{+}_{\widehat{\mathcal{C}}}(\lambda, \mathbf{skip}) \triangleq \emptyset$$

$$\Delta^{+}_{\widehat{\mathcal{C}}}(\lambda, x := [\mathcal{E}]_{\bar{\mathsf{f}}}) \triangleq \begin{cases} \{\mathcal{E}\} & \text{if } \Gamma(\mathcal{E}) \sqsubseteq_{\Lambda} \lambda \ \wedge \ \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq_{\Lambda} \lambda \\ \emptyset & \text{otherwise} \end{cases}$$

$$\Delta^{+}_{\widehat{\mathcal{C}}}(\lambda, \mathbf{if}\ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \mathbf{then}\ \mathcal{C}_t\ \mathbf{else}\ \mathcal{C}_e) \triangleq \begin{cases} \{\mathcal{E}\} & \text{if } \Gamma(\mathcal{E}) \sqsubseteq_{\Lambda} \lambda \ \wedge \ \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq_{\Lambda} \lambda \\ \emptyset & \text{otherwise} \end{cases}$$

$$\Delta^{+}_{\widehat{\mathcal{C}}}(\lambda, \mathbf{while}\ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \mathbf{do}\ \mathcal{C}_w) \triangleq \begin{cases} \{\mathcal{E}\} & \text{if } \Gamma(\mathcal{E}) \sqsubseteq_{\Lambda} \lambda \ \wedge \ \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq_{\Lambda} \lambda \\ \emptyset & \text{otherwise} \end{cases}$$

$$\Delta^{+}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_1; \mathcal{C}_2) \triangleq \Delta^{+}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C}_1)$$

Figure 8: Definition of $\Delta^{+}_{\widehat{\mathcal{C}}}(\lambda, \mathcal{C})$

$$\text{VAL-T:}\ \frac{}{\Gamma \vdash \nu{:}\bot} \qquad \text{VAR-T:}\ \frac{\Gamma(x) = \lambda}{\Gamma \vdash x{:}\lambda}$$

$$\text{EXPR-T:}\ \frac{\Gamma \vdash \mathcal{E}_1{:}\lambda_1 \quad \ldots \quad \Gamma \vdash \mathcal{E}_n{:}\lambda_n}{\Gamma \vdash \ op(\mathcal{E}_1, \ldots, \mathcal{E}_n) : \lambda_1 \sqcup_\Lambda \cdots \sqcup_\Lambda \lambda_n}$$

$$\text{ANNEXPR-T:}\ \frac{\Gamma \vdash \mathcal{E}_1{:}\lambda_1 \quad \ldots \quad \Gamma \vdash \mathcal{E}_n{:}\lambda_n}{\Gamma \vdash \ [op(\mathcal{E}_1, \ldots, \mathcal{E}_n)]_{\mathsf{f}_1,\ldots,\mathsf{f}_n} : \mathcal{T}(\lambda_1, \mathsf{f}_1) \sqcup_\Lambda \cdots \sqcup_\Lambda \mathcal{T}(\lambda_n, \mathsf{f}_n)}$$

Figure 9: Typing rules for expressions

$$\text{SKIP-T:}\ \frac{}{\Gamma, \lambda_\kappa \vdash \mathbf{skip}}$$

$$\text{ASGN-T:}\ \frac{\Gamma \vdash [\mathcal{E}]_{\bar{\mathsf{f}}} {:}\lambda_g \quad \Gamma \vdash x{:}\lambda_x \quad \lambda_\kappa \sqcup_\Lambda \lambda_g \sqsubseteq_\Lambda \lambda_x}{\Gamma, \lambda_\kappa \vdash x := [\mathcal{E}]_{\bar{\mathsf{f}}}}$$

$$\text{BRCH-T:}\ \frac{\Gamma \vdash [\mathcal{E}]_{\bar{\mathsf{f}}} {:}\lambda_g \quad \Gamma, \lambda_\kappa \sqcup_\Lambda \lambda_g \vdash \mathcal{C}_t \quad \Gamma, \lambda_\kappa \sqcup_\Lambda \lambda_g \vdash \mathcal{C}_e}{\Gamma, \lambda_\kappa \vdash \mathbf{if}\ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \mathbf{then}\ \mathcal{C}_t\ \mathbf{else}\ \mathcal{C}_e}$$

$$\text{LOOP-T:}\ \frac{\Gamma \vdash [\mathcal{E}]_{\bar{\mathsf{f}}} {:}\lambda_g \quad \Gamma, \lambda_\kappa \sqcup_\Lambda \lambda_g \vdash \mathcal{C}_t}{\Gamma, \lambda_\kappa \vdash \mathbf{while}\ [\mathcal{E}]_{\bar{\mathsf{f}}}\ \mathbf{do}\ \mathcal{C}_t} \qquad \text{SEQ-T:}\ \frac{\Gamma, \lambda_\kappa \vdash \mathcal{C}_1 \quad \Gamma, \lambda_\kappa \vdash \mathcal{C}_2}{\Gamma, \lambda_\kappa \vdash \mathcal{C}_1; \mathcal{C}_2}$$

Figure 10: Typing rules for commands

Figure 10 shows the familiar rules to deduce whether a command is type-correct. Judgment $\Gamma, \lambda_\kappa \vdash \mathcal{C}$ signifies that a command $\mathcal{C}$ is type correct. Parameter λ_κ in these rules is called *pc-label*. It is used in checking for implicit flows. A pc-label λ_κ associates a type with the conjunction of the guards for all of the enclosing conditional commands; λ_κ is combined with the type of the right hand side of an assignment statement.

The following theorem states that if a program is type correct, then this program satisfies PWNI.

Theorem 1. *If $\Gamma, \lambda_\kappa \vdash \mathcal{C}$, then PWNI($\mathcal{C}$) holds.*

Proof. See [33, Appendix A.1] □

It was not a coincidence that we could enforce our expressive RIF labels

by adapting a type system like the one used in Volpano *et al.* [59]. The type system in [59] can enforce any set of labels that form a join semi-lattice, and RIF labels form a join semi-lattice. In general, any enforcement mechanism that enforces noninterference for traditional label lattices, should be easily extendable to enforce piecewise noninterference for RIF labels, since piecewise noninterference is the conjunction of noninterference for successive λ-pieces of execution.

6 κ-Labels: Illustrating Leverage from Type-Checking RIF Labels

Although RIF automata are a checkable class of RIF labels, they are not expressive enough to specify allowed flows of information in the presence of cryptographic operations. In this section we give another example of a checkable class of RIF labels: κ-labels. Programs written in terms of a type system based on κ-labels can be checked by a compiler in order to get assurance that a value v flows to v' only if principals authorized to read v' are (i) authorized to read v or (ii) unable to reverse some cryptographic operation used in generating v' from v. So the type system allows us to conclude that encryptions and decryptions are successfully keeping secrets. Our modelling of cryptographic operations follows the symbolic formulation of Dolev and Yao [22].[14]

The starting point for defining allowed flows based on κ-labels is mapping KN to sets of principals from variables and from cryprographic keys.

- $KN(x)$ is the set of principals allowed to know the initial value of variable x.
- $KN(k)$ is the set of principals allowed to know the value of cryptographic key k, which is considered a constant.

Reclassifiers for κ-labels are associated with cryptographic operations, and rewrite rules for sequences of reclassifiers characterize when cryptographic operations are inverses. Different cryptosystems and cryptographic operations give rise to different reclassifiers along with different sets of rewrite rules.

[14] Notice that we do not consider a Dovel-Yao attacker. Our threat model is the one defined in subsection 4.1.

κ-Atoms. A κ-*atom* κ_i for a value v_i to which v has flowed is a pair $\langle F_i, \beta_i \rangle$, where $F_i \in \mathcal{F}_\kappa^*$ gives the sequence of cryptographic operations involved in deriving v_i from v, and β_i is a set of principals authorized to read v. Let $\mathcal{A}_\kappa$ denote the set of κ-atoms. Reclassifier $\theta(a_1, \ldots, a_n) \in \mathcal{F}_\kappa$ is associated with a cryptographic operation θ whose invocation has a_1, ..., a_n in its list of arguments, which may be cryptographic keys or other expressions.

A *rewrite rule* will have form

$$\theta(a)\, \theta'(a') \rightarrowtail \epsilon$$

where $\theta(a)$ and $\theta'(a')$ are reclassifiers. This rewrite rule defines $\theta'(a')$ to be the *complement* of $\theta(a)$ and allows appearances of "$\theta(a)\ \theta'(a')$" to be deleted in a sequence of reclassifiers. Notation $\theta(a)^{\mathsf{c}}$ will be used to denote the complement of reclassifier $\theta(a)$. Reclassifiers are not required to have complements, but if complements $\theta(a)^{\mathsf{c}}$ and $(\theta(a)^{\mathsf{c}})^{\mathsf{c}}$ both do exist then we require that $(\theta(a)^{\mathsf{c}})^{\mathsf{c}} = \theta(a)$ holds.

To illustrate, we formulate κ-atoms for a symmetric-key cryptosystem having two cryptographic operations—encryption and decryption. $Enc(p, k)$ produces a *ciphertext* from *plaintext* p and key k. Its reclassifiers are defined by regarding appearances of $Enc(p, k)$ to be instances of reclassifying expression

$$[Enc(p, k)]_{\theta^1_{Enc}(k),\, \theta^2_{Enc}(p)}$$

which posits flows occur from p (associated with reclassifier $\theta^1_{Enc}(k)$) and from k (associated with reclassifier $\theta^2_{Enc}(p)$) to the value that $Enc(p, k)$ produces. $Dec(c, k)$ recovers the plaintext iff ciphertext c was previously encrypted using k. It has reclassifiers for the flows from c and from k to the value that $Dec(c, K)$ produces:

$$[Dec(c, k)]_{\theta^1_{Dec}(k),\, \theta^2_{Dec}(c)}$$

And we assume *Enc* and *Dec* satisfy the expected properties.

$$Dec(Enc(v, k), k) = v \text{ holds for all values } v \text{ and keys } k. \tag{38}$$

$$Dec(c, k) \text{ is the only way to recover } p \text{ from } c = Enc(p, k). \tag{39}$$

$$\text{Key } k \text{ cannot be recovered from ciphertexts created using } k. \tag{40}$$

Property (38) stipulates that the effects of $Enc(\cdot, k)$ on plaintext can be reversed by using $Dec(\cdot, k)$; it is that basis for rewrite rule

$$\forall k\colon \quad \theta^1_{Enc}(k)\, \theta^1_{Dec}(k) \rightarrowtail \epsilon. \tag{41}$$

Property (39) further implies that $\theta^1_{Dec}(k)$ is the only complement of $\theta^1_{Enc}(k)$. Thus, (39) stipulates an absence of certain rewrite rules. Finally, (40) prohibits rewrite rules that would define complements for reclassifiers associated with flows from k: $\theta^2_{Enc}(\cdot)$ and $\theta^2_{Dec}(\cdot)$. This prohibition corresponds to forbidding cryptographic functions where a key used as input to that function can be recovered from the output. It also rules out cryptographic systems that admit plaintext attacks.[15]

Reduced sequence $(\!|F|\!)$ denotes the sequence obtained by repeatedly applying some given rewrite rules to sequence F of reclassifiers, until no longer possible. By definition, no reclassifier is followed by its complement in a reduced sequence. If the full set of rewrite rules defines at most one complement for each reclassifier then (i) reduced sequence $(\!|F|\!)$ is unique and (ii) the order in which rewrite rules are applied does not matter, so $(\!|F|\!)$ is associative; see [33, Appendix A.3] for proofs.

All reclassifiers in rewrite rule (41) have a single argument k; it is a key. We define set $\mathcal{X}(\theta(k))$ of principals that can recover a value produced by a cryptographic operation associated with a reclassifier $\theta(k)$ as follows.

$$\mathcal{X}(\theta(k)) \triangleq \begin{cases} \emptyset & \text{if complement } \theta(k)^{\mathsf{c}} \text{ does not exist} \\ KN(k') & \text{if } \theta(k)^{\mathsf{c}} = \theta'(k') \end{cases}$$

Set complement $\overline{\mathcal{X}(\mathsf{f})}$ thus contains those principals that cannot recover a value produced by the cryptographic operation identified by f.

If a value v' is derived from v by performing a sequence of cryptographic operations described with sequence F of reclassifiers then v can be recovered from v' only by those principals able to perform inverses of all operations mentioned in $(\!|F|\!)$. So the set $\overline{\mathcal{X}}(F)$ of principals that cannot recover a value produced using a sequence F of cryptographic operations are those principals

[15]A *plaintext attack* uses plaintext p and corresponding ciphertext c to recover the encryption key k satisfying $c = Enc(p, k)$. The attack can be viewed as having a cryptographic function $[PA(p, c)]_{\theta^1_{PA}(c), \theta^2_{PA}(p)}$ that satisfies:

$$PA(p, c) = k \quad \text{if } c = Enc(p, k)$$

This semantics for $PA(p, c)$ would lead to rewrite rules—precluded by (40) stipulating that plaintext attacks are infeasible—for the flows from k in $Enc(p, k)$:

$$\theta^2_{Enc}(p)\, \theta^2_{PA}(p) \rightarrowtail \epsilon$$

that cannot invert at least one of the operations in F. We characterize that set formally as follows, writing $\mathsf{f} \in (\!|F|\!)$ to indicate that f ranges over the reclassifiers appearing in reduced sequence $(\!|F|\!)$.

$$\overline{\mathcal{X}}(F) \triangleq \bigcup_{\mathsf{f} \in (\!|F|\!)} \overline{\mathcal{X}(\mathsf{f})}$$

κ-atoms concern confidentiality for principals in a set *Prins*. Subsets are more restrictive, so we use $\langle 2^{Prins}, \cap, \supseteq \rangle$ as the join semilattice $\langle R, \sqcup_R, \sqsubseteq_R \rangle$ of underlying restrictions. $\mathcal{R}_{\mathcal{A}}(\langle F, \beta \rangle)$ is defined to be the set of principals to which the value associated with $\langle F, \beta \rangle$ may flow—the set of principals in β along with principals $\overline{\mathcal{X}}(F)$ that cannot recover an input to the sequence F of cryptographic operations:

$$\mathcal{R}_{\mathcal{A}}(\langle F, \beta \rangle) \triangleq \beta \cup \overline{\mathcal{X}}(F)$$

And $\mathcal{T}_{\mathcal{A}}(\langle F, \beta \rangle, \mathsf{f})$ specifies the flows allowed by $\langle F, \beta \rangle$ when its associated value is transformed by some cryptographic operation being identified with reclassifier f.

$$\mathcal{T}_{\mathcal{A}}(\langle F, \beta \rangle, \mathsf{f}) \triangleq \langle F\mathsf{f}, \beta \rangle$$

For example, an initial value v that can flow to principals in β_v would be associated with κ-atom $\langle \epsilon, \beta_v \rangle$. And κ-atom $\mathcal{T}_{\mathcal{A}}(\langle F, \beta \rangle, \mathsf{f})$ associated with a transformed value potentially changes the set of principals where that transformed value might flow.[16]

The definition of restrictiveness relation $\sqsubseteq_{\mathcal{A}}$ on κ-atoms is obtained by substituting $\mathcal{R}_{\mathcal{A}}$ and $\mathcal{T}_{\mathcal{A}}$ for $\mathcal{R}$ and $\mathcal{T}$ in $\sqsubseteq_{\Lambda}$ definition (8). Notice that for all $F \in \mathcal{F}_{\kappa}^*$

$$\langle F, \beta \rangle = \langle (\!|F|\!), \beta \rangle$$

and, therefore, an implementation of κ-atoms need only store reduced sequences.

A computable test to decide $\sqsubseteq_{\mathcal{A}}$ for κ-atoms can be obtained by extending complement sequences from Dolev-Yao [22], as follows. For any finite sequence F of reclassifiers, define *maximal complement sequence* F^{c} to be

[16]The new set of principals is a superset when $(\!|F\mathsf{f}|\!)$ extends $(\!|F|\!)$ and $\overline{\mathcal{X}(\mathsf{f})} \not\subseteq \overline{\mathcal{X}}(F)$ holds. It is a subset when $(\!|F\mathsf{f}|\!)$ is a proper prefix of $(\!|F|\!)$, $\overline{\mathcal{X}(\mathsf{f})} \not\subseteq \overline{\mathcal{X}}(F\mathsf{f})$ and $\overline{\mathcal{X}(\mathsf{f}^{\mathsf{c}})} \not\subseteq \overline{\mathcal{X}}(F\mathsf{f})$ hold. (If $(\!|F\mathsf{f}|\!)$ is a proper prefix of $(\!|F|\!)$ then f is the complement of the final reclassifier in $(\!|F|\!)$, so $(\!|F\mathsf{f}|\!)$ contains neither the final reclassifier in $(\!|F|\!)$ nor the f being added.)

the sequence of reclassifiers that minimizes the length of $(\!|F\,F^{\mathsf{c}}|\!)$. F^{c} can be computed by taking the complement of each element in F, starting at the end.

$$F^{\mathsf{c}} \triangleq \begin{cases} \epsilon & \text{if } F = \epsilon \\ \mathsf{f}^{\mathsf{c}} F_1^{\mathsf{c}} & \text{if } F = F_1\mathsf{f} \text{ and } \mathsf{f}^{\mathsf{c}} \text{ exists} \\ \epsilon & \text{otherwise} \end{cases}$$

A test for $\sqsubseteq_{\mathcal{A}}$ is then given by:

$$\langle F_1, \beta_1\rangle \sqsubseteq_{\mathcal{A}} \langle F_2, \beta_2\rangle \;=\; \mathcal{R}_{\mathcal{A}}(\langle\, F_1 F_1^{\mathsf{c}},\ \beta_1\rangle) \supseteq \mathcal{R}_{\mathcal{A}}(\langle\, F_2 F_1^{\mathsf{c}},\ \beta_2\rangle) \tag{42}$$

Soundness and completeness proofs for this test are given in [33, Appendix A.3].

κ-Labels. A κ-label $\mathcal{K}$ is a finite set of κ-atoms. If we adopt the following definitions for $\mathcal{R}_{\mathcal{L}}(\mathcal{K})$, and $\mathcal{T}_{\mathcal{L}}(\mathcal{K}, \mathsf{f})$,

$$\mathcal{R}_{\mathcal{L}}(\mathcal{K}) \triangleq \bigsqcup_R \kappa \in \mathcal{K}\colon\ \mathcal{R}_{\mathcal{A}}(\kappa) \tag{43}$$

$$\mathcal{T}_{\mathcal{L}}(\mathcal{K}, \mathsf{f}) \triangleq \{\mathcal{T}_{\mathcal{A}}(\kappa, \mathsf{f}) \mid \kappa \in \mathcal{K}\} \tag{44}$$

then κ-labels form a join semilattice $\langle 2^{\mathcal{A}_\kappa},\ \sqcup_{\mathcal{L}},\ \sqsubseteq_{\mathcal{L}}\rangle$ where $\sqcup_{\mathcal{L}}$ is defined by

$$\mathcal{K} \sqcup_{\mathcal{L}} \mathcal{K}' \triangleq \mathcal{K} \cup \mathcal{K}' \tag{45}$$

and restrictiveness relation $\sqsubseteq_{\mathcal{L}}$ is defined by substituting $\mathcal{R}_{\mathcal{L}}$ and $\mathcal{T}_{\mathcal{L}}$ into $\sqsubseteq_{\Lambda}$ definition (8). The RIF class is then characterized by:

$$\langle\langle R,\ \sqcup_R,\ \sqsubseteq_R\rangle, \langle 2^{\mathcal{A}_\kappa},\ \sqcup_{\mathcal{L}},\ \sqsubseteq_{\mathcal{L}}\rangle, \mathcal{F}_\kappa, \mathcal{R}_{\mathcal{L}}, \mathcal{T}_{\mathcal{L}}\rangle \tag{46}$$

Examples of κ-labels. To illustrate, we typecheck a simple command that invokes operations of the symmetric cryptosystem formalized above. Suppose x is allowed to flow to some subset P_x of set *Prins* of all principals, principals in P_x are authorized to know key k, y is allowed to flow to some subset P_y of *Prins*, and principals in P_y are authorized to know key k':

$$KN(x) = P_x \qquad KN(k) = P_x \qquad KN(y) = P_y \qquad KN(k') = P_y$$

Here is a command that combines the symmetric-key cryptographic operations discussed above.

$$\begin{aligned} &w := [Enc(x, k)]_{\theta^1_{Enc}(k),\, \theta^2_{Enc}(x)}\,; \\ &y := [Dec(w, k')]_{\theta^1_{Dec}(k'),\, \theta^2_{Dec}(w)} \end{aligned} \tag{47}$$

A first assignment encrypts x using key k and then a second assignment attempts to decrypt that ciphertext using a key k'.

Given the assumptions above about x and k, we posit the following type declarations for the variables named in the first assignment of (47). Notice, the type for w corresponds to a value that has been encrypted under key k.

$$\Gamma(x) \triangleq \{\langle\epsilon, P_x\rangle\} \qquad \Gamma(k) \triangleq \{\langle\epsilon, P_x\rangle\} \qquad \Gamma(w) \triangleq \{\langle\theta^1_{Enc}(k), P_x\rangle\} \tag{48}$$

Using these types and typing rule AnnExpr-T in Figure 9, we obtain a type for the RHS of the first assignment of (47).

$$\mathcal{T}_{\mathcal{L}}(\,\Gamma(x),\ \theta^1_{Enc}(k)\,) \ \sqcup_{\mathcal{L}}\ \mathcal{T}_{\mathcal{L}}(\,\Gamma(k),\ \theta^2_{Enc}(x)\,)$$

So, according to typing rule Asgn-T in Figure 10, that assignment is type correct provided the following holds.

$$(\mathcal{T}_{\mathcal{L}}(\,\Gamma(x),\ \theta^1_{Enc}(k)\,) \sqcup_{\mathcal{L}} \mathcal{T}_{\mathcal{L}}(\,\Gamma(k),\ \theta^2_{Enc}(x)\,)) \ \sqsubseteq_{\mathcal{L}}\ \Gamma(w) \tag{49}$$

Substituting according to type declarations (48), obligation (49) for type correctness simplifies as follows:

$$\begin{aligned}
&\{\langle\theta^1_{Enc}(k), P_x\rangle\} \ \cup\ \{\langle\theta^2_{Enc}(x), P_x\rangle\} \sqsubseteq_{\mathcal{L}} \ \{\langle\theta^1_{Enc}(k), P_x\rangle\}\\
&= \ \{\langle\theta^1_{Enc}(k), P_x\rangle,\ \langle\theta^2_{Enc}(x), P_x\rangle\} \sqsubseteq_{\mathcal{L}} \ \{\langle\theta^1_{Enc}(k), P_x\rangle\}\\
&= \ (\forall F\colon\ \mathcal{R}_{\mathcal{L}}(\{\langle\theta^1_{Enc}(k)F, P_x\rangle,\ \langle\theta^2_{Enc}(x)F, P_x\rangle\}) \sqsubseteq_{\mathcal{R}} \ \mathcal{R}_{\mathcal{L}}(\{\langle\theta^1_{Enc}(k)F,\ P_x\rangle\}))\\
&= (\forall F\colon\ \mathcal{R}_{\mathcal{A}}(\langle\theta^1_{Enc}(k)F, P_x\rangle) \sqcup_R \mathcal{R}_{\mathcal{A}}(\langle\theta^2_{Enc}(x)F, P_x\rangle) \sqsubseteq_{\mathcal{R}} \ \mathcal{R}_{\mathcal{A}}(\langle\theta^1_{Enc}(k)F,\ P_x\rangle))\\
&= (\forall F\colon\ (P_x \cup \overline{\mathcal{X}}(\theta^1_{Enc}(k)F)) \cap (P_x \cup \overline{\mathcal{X}}(\theta^2_{Enc}(x)F) \ \supseteq\ P_x \cup \overline{\mathcal{X}}(\theta^1_{Enc}(k)F))\\
&= (\forall F\colon\ (P_x \cup \overline{\mathcal{X}}(\theta^1_{Enc}(k)F)) \cap (P_x \cup \mathit{Prins}) \ \supseteq\ P_x \cup \overline{\mathcal{X}}(\theta^1_{Enc}(k)F))\\
&= (\forall F\colon\ (P_x \cup \overline{\mathcal{X}}(\theta^1_{Enc}(k)F) \ \supseteq\ P_x \cup \overline{\mathcal{X}}(\theta^1_{Enc}(k)F))
\end{aligned}$$

The final formula is trivially equivalent to *true*.

Analogous reasoning establishes that the second assignment is type correct provided the following holds.

$$(\mathcal{T}_{\mathcal{L}}(\,\Gamma(w),\ \theta^1_{Dec}(k')\,) \sqcup_{\mathcal{L}} \mathcal{T}_{\mathcal{L}}(\,\Gamma(k'),\ \theta^2_{Dec}(w)\,)) \ \sqsubseteq_{\mathcal{L}}\ \Gamma(y) \tag{50}$$

The validity of (50) depends on whether $k = k'$ holds and on the choice of P_y and $P_{k'}$ in the following proposed types for $\Gamma(y)$ and $\Gamma(k')$:

$$\Gamma(y) \triangleq \{\langle\epsilon, P_y\rangle\} \qquad\qquad \Gamma(k') \triangleq \{\langle\epsilon, P_{k'}\rangle\}$$

- Case $k = k'$: From $\Gamma(k') = \Gamma(k)$, we conclude from $\Gamma(k)$ and $\Gamma(k')$ that $P_{k'} = P_x$ holds. Expanding (50) gets

  $(\forall \mathsf{F}\colon\ (P_x \cup \overline{\mathcal{X}}(\theta^1_{Enc}(k)\,\theta^1_{Dec}(k)\,\mathsf{F})) \cap (P_x \cup \overline{\mathcal{X}}(\theta^2_{Dec}(w)\mathsf{F})) \ \supseteq\ (P_y \cup \overline{\mathcal{X}}(\mathsf{F})))$
  $=$ due to (40), $\overline{\mathcal{X}}(\theta^2_{Dec}(w)\mathsf{F}) = \mathit{Prins}$
  $(\forall \mathsf{F}\colon\ P_x \cup \overline{\mathcal{X}}(\theta^1_{Enc}(k)\,\theta^1_{Dec}(k)\,\mathsf{F}) \ \supseteq\ (P_y \cup \overline{\mathcal{X}}(\mathsf{F})))$
  $=$ since $\overline{\mathcal{X}}(\theta^1_{Enc}(k)\,\theta^1_{Dec}(k)\,\mathsf{F})$ simplifies to $\overline{\mathcal{X}}(\mathsf{F})$
  $(\forall \mathsf{F}\colon\ P_x \cup \overline{\mathcal{X}}(\mathsf{F}) \ \supseteq\ (P_y \cup \overline{\mathcal{X}}(\mathsf{F})))$

  This final formula is valid provided $P_x \supseteq P_y$ holds. So we have established that if $k = k'$ holds then the assignment is type correct provided $P_x \supseteq P_y$ holds. This conclusion should not be surprising, since if $P_x \not\supseteq P_y$ holds then once w has been decrypted, the value of x (which would now be stored in y) could flow to principals not in P_x (because they are in $P_y - P_x$).

- Case $k \neq k'$: Expanding (50) yields::

  $(\forall \mathsf{F}\colon (P_x \cup \overline{\mathcal{X}}(\theta^1_{Enc}(k)\,\theta^1_{Dec}(k')\,\mathsf{F})) \cap (P_{k'} \cup \overline{\mathcal{X}}(\theta^2_{Dec}(w)\mathsf{F})) \ \supseteq\ (P_y \cup \overline{\mathcal{X}}(\mathsf{F})))$

  This formula is valid for all values of P_x and P_y since, by definition of $\overline{\mathcal{X}}(\cdot)$,

  $$\overline{\mathcal{X}}(\theta^1_{Enc}(k)\,\theta^1_{Dec}(k')\,\mathsf{F}) = \mathit{Prins} \text{ and } \overline{\mathcal{X}}(\theta^2_{Dec}(w)\mathsf{F}) = \mathit{Prins}$$

  because complements $\theta^1_{Dec}(k')^{\mathsf{c}}$ and $\theta^2_{Dec}(w)^{\mathsf{c}}$ do not exist. Thus, we conclude the assignment is type correct when $k \neq k'$, for any choices of P_y and $P_{k'}$. Again, this outcome should not be surprising. With no way to recover x from the value stored in y by the second assignment (since we have assumed that cryptographic operation $Dec(w, k')$ has no inverse), the second assignment cannot cause a flow violation.

Type-correctness guarantee for κ-labels. The connection between type correctness and flows proved for this example command is an instance of a more-general guarantee about type correctness. To formalize this guarantee, note that any value a principal p could read must be tagged with a κ-label $\mathcal{K}_L$ satisfying $\mathcal{K}_L \sqsubseteq_{\mathcal{L}} \mathcal{K}_{\mathsf{p}}$, where

$$\mathcal{K}_{\mathsf{p}} \ \triangleq\ \{\langle \mathsf{F}, \beta\rangle \in \mathcal{A}_{\kappa} \mid \mathsf{p} \in \mathcal{R}_{\mathcal{A}}(\,\langle \mathsf{F}, \beta\rangle\,)\}.$$

We consider a κ-label $\mathcal{K}$ to be p-*low* iff $\mathcal{K} \sqsubseteq_{\mathcal{L}} \mathcal{K}_{\mathsf{p}}$ holds and to be p-*high* iff $\mathcal{K} \not\sqsubseteq_{\mathcal{L}} \mathcal{K}_{\mathsf{p}}$ holds. Thus the type-correctness guarantee provided by (any) κ-labels can be formulated as limiting what p can learn from values that flow from p-high to p-low—in particular, type correctness ensures that p cannot recover a p-high value from p-low values.

To show why this guarantee persists when using the κ-labels we defined above for the symmetric-key cryptosystem, consider a command $\mathcal{C}$ that is type correct. Theorem 1 implies $PWNI(\mathcal{C})$ will hold and, therefore, if a value flows from p-high to p-low, then it does so through some operation that performs a $\mathcal{K}_{\mathsf{p}}$-downgrade. Since cryptographic operations $Enc(\cdot,\cdot)$ and $Dec(\cdot,\cdot)$ are the sole reclassifying expressions for our κ-labels, we conclude that all $\mathcal{K}_{\mathsf{p}}$-downgrades are caused by these operations. Without loss of generality, let such an operation be described by

$$[\theta(\mathcal{E}, k)]_{\theta^1(k),\, \theta^2(\mathcal{E})} \tag{51}$$

which has a flow from $\mathcal{E}$ (associated with reclassifier $\theta^1(k)$) and a flow from k (associated with reclassifier $\theta^2(\mathcal{E})$) to the value that $\theta(\mathcal{E}, k)$ produces.

By definition, for (51) to exhibit a $\mathcal{K}_{\mathsf{p}}$-downgrade then

$$\Gamma([\theta(\mathcal{E}, k)]_{\theta^1(k),\, \theta^2(\mathcal{E})}) \sqsubseteq_{\mathcal{L}} \mathcal{K}_{\mathsf{p}} \quad \wedge \quad \Gamma(\theta(\mathcal{E}, k)) \not\sqsubseteq_{\mathcal{L}} \mathcal{K}_{\mathsf{p}}$$

must hold. From the first conjunct we get:

$$\mathsf{p} \in \mathcal{R}_{\mathcal{L}}(\ \Gamma([\theta(\mathcal{E}, k)]_{\theta^1(k),\, \theta^2(\mathcal{E})})\ ) \tag{52}$$

To validate the type-correctness guarantee, it suffices to prove that p cannot recover either $\mathcal{E}$ or k from $\theta(\mathcal{E}, k)$ or, equivalently that the following hold.

$$\mathsf{p} \in \overline{\mathcal{X}(\theta^1(k))} \qquad\qquad \mathsf{p} \in \overline{\mathcal{X}(\theta^2(\mathcal{E}))}$$

- Establishing $\mathsf{p} \in \overline{\mathcal{X}(\theta^1(k))}$.

$$\begin{aligned}
& \mathsf{p} \in \mathcal{R}_{\mathcal{L}}(\ \Gamma([\theta(\mathcal{E}, k)]_{\theta^1(k),\, \theta^2(\mathcal{E})}\ ) \\
= \;& \textsc{AnnExpr-T} \\
& \mathsf{p} \in \mathcal{R}_{\mathcal{L}}(\ \mathcal{T}_{\mathcal{L}}(\Gamma(\mathcal{E}),\ \theta^1(k)) \ \sqcup_{\mathcal{L}}\ \mathcal{T}_{\mathcal{L}}(\Gamma(k),\ \theta^2(\mathcal{E}))\ ) \\
= \;& \text{definition of } \mathcal{R}_{\mathcal{L}} \text{ where } \sqcup_R \text{ is } \cap \\
& \mathsf{p} \in \mathcal{R}_{\mathcal{L}}(\ \mathcal{T}_{\mathcal{L}}(\Gamma(\mathcal{E}),\ \theta^1(k))\ ) \ \cap\ \mathcal{R}_{\mathcal{L}}(\ \mathcal{T}_{\mathcal{L}}(\Gamma(k),\ \theta^2(\mathcal{E}))\ )
\end{aligned}$$

$$
\begin{aligned}
= &\ (\theta^2(\mathcal{E}))^{\mathsf{c}} \text{ not defined so } \mathcal{R}_{\mathcal{L}}(\ \mathcal{T}_{\mathcal{L}}(\Gamma(k),\ \theta^2(\mathcal{E}))\ ) = \mathit{Prins} \\
&\ \mathsf{p} \in \mathcal{R}_{\mathcal{L}}(\ \mathcal{T}_{\mathcal{L}}(\Gamma(\mathcal{E}),\ \theta^1(k))\ ) \\
= &\ \text{ let } \Gamma(\mathcal{E}) = \{\langle F_1, \beta_1\rangle, \ldots, \langle F_n, \beta_n\rangle\} \\
&\ \mathsf{p} \in (\ \mathcal{R}_{\mathcal{A}}(\langle F_1\theta^1(k),\ \beta_1\rangle) \sqcup_R \cdots \sqcup_R \mathcal{R}_{\mathcal{A}}(\langle F_n\theta^1(k),\ \beta_n\rangle)\ ) \\
= &\ \text{ definition of } \mathcal{R}_{\mathcal{A}};\ \sqcup_R \text{ is } \cap \\
&\ \mathsf{p} \in (\ (\beta_1 \cup \overline{\mathcal{X}}(F_1\theta^1(k))) \cap \cdots \cap (\beta_n \cup \overline{\mathcal{X}}(F_n\theta^1(k))) \\
= &\ \Gamma(\theta(\mathcal{E}, k)) \not\sqsubseteq_{\mathcal{L}} \mathcal{K}_{\mathsf{p}} \Rightarrow \mathsf{p} \notin (\ (\beta_1 \cup \overline{\mathcal{X}}(F_1)) \cap \cdots \cap (\beta_n \cup \overline{\mathcal{X}}(F_n))) \\
&\ \mathsf{p} \in \overline{\mathcal{X}(\theta^1(k))}
\end{aligned}
$$

- Establishing $\mathsf{p} \in \overline{\mathcal{X}(\theta^2(\mathcal{E}))}$. According to (40), $\theta^2(\mathcal{E})$ has no complement. Thus, by definition, $\overline{\mathcal{X}(\theta^2(\mathcal{E}))} = \mathit{Prins}$.

7 Related Work

RIF labels specify restrictions that depend on the history of applied operations. And a static type system can ensure that programs using RIF labels will satisfy PWNI, which extends noninterference [27] to accommodate reclassification. Our work thus extends an approach initiated by Volpano *et al.* [59]—a type system based on Denning's lattice model [21, 19, 20] that enforces noninterference.

We are not the first to explore types that depend on a history of applied operations. Strom and Yemini [57] propose types that incorporate *typestate* to summarize the history of operations previously applied to an object and to govern the set of operations that may next be invoked. But Hartson and Hsiao [29] seem to be the first to use history in access control. Contemporary uses of history for defining authorization include *stack inspection* [61] and *history-based access control* [1].

Reclassification: Specification and Enforcement

State Based Reclassification. A good deal of prior work ties reclassification to changes in state predicates. Chong *et al.* [16] is closest to RIF labels. There, information flow policies for confidentiality are defined in terms of state predicates that specify when a value may be reclassified. A static type system enforces these declassification and erasure policies. And when a

reclassification occurs in the approach of Chong *et al.* [16], the value and all derived values are reclassified—in contrast, with RIF labels, reclassifications apply only to the single value.

With ClickRelease, Micinski *et al.* [43] extend the approach in Chong *et al.* [16] by allowing a more-expressive language for the formulas that specify when values may be declassified. In particular, ClickRelease uses temporal logic formulas over events, in contrast to the state predicates used by Chong *et al.* [16]

Paralocks [13] formulates a security policy as $\Sigma \Rightarrow \alpha$, where Σ is a set of state predicates and α is a principal. A value associated with such a policy is allowed to flow to α only if all predicates in Σ are true. Changes to state predicates in Σ alter the allowed flows. Paralocks policies are enforced using a static type system.

A policy in Thoth [24] and its successor system SHAI [23] specifies confidentiality and integrity restrictions for data containers called *conduits*. The policies comprise two layers. The *access control layer* specifies which principals may read and update the associated conduit and under what conditions. The *declassification layer* specifies conditions under which policies for data derived from the associated conduit can change. The conditions employed by a Thoth policy are predicates on conduits' state, which include both data and metadata (including a type) of conduits. Thoth uses dynamic analysis for enforcement, SHAI uses static analysis too.

Jeeves [6] employs *faceted values* [5] to specify declassification between two levels of confidentiality (i.e., public and secret). A faceted value is a pair comprising a real and a dummy value, guarded by a state predicate. If that state predicate (which itself can be a faceted value) holds, then the real, possibly secret, value is allowed to flow to low outputs. Otherwise, the dummy value flows to low outputs.

All of this prior work thus ties reclassification to changes in state predicates. In contrast, RIF labels tie reclassification to expression evaluation. A comparison of RIF labels with reclassification based on state predicates thus depends first on what state predicates are available and second on whether those state predicates could be incorporated into RIF transition function $\mathcal{T}$.

Operation Based Reclassification. Explicit expressions for declassification (for confidentiality) and endorsement (for integrity) have been proposed [44, 46]. However, the approach can be unsatisfying because output restrictions are not connected to input restrictions or to the operation that transforms inputs

to outputs. For example, [44, 46] allow an arbitrary label to be assigned to the result of evaluating any expression—type-casting has the same flavor.

RIF labels tie reclassification to expression evaluation, but it is not the first work to connect restrictions on outputs to an operation. With FJifP [30], a principal declares *trusted methods* to declassify input values.[17] A FJifP trusted method always performs the same declassification of an input whereas, depending on the RIF label associated with an input value, a specific reclassifier in a reclassifying expression could trigger different changes to the restrictions. So reclassifying expressions (in conjunction with RIF labels) are more expressive than trusted methods.

Rocha *et al.* [47] propose *policy graphs* to specify declassification of information from secret to public (as compared to RIF labels, which handle arbitrary reclassification); similar techniques to those presented in [47] are later employed by Hammer *et al.* [28] and Johnson *et al.* [32]. In a policy graph, nodes represent variables and edges represent operation identifiers (similar to our reclassifiers). The tail node of an edge is an input of the corresponding operation, and the head node of that edge is an output of that operation. Some of the nodes in a policy graph are defined *final*. Values in variables of non-final nodes are considered secret, whereas values in variables that correspond to final nodes are declassified and considered public. Data and control-flow analysis is used to check whether some given program satisfies a specific policy graph. Subsequently, Rocha *et al.* [48] introduce a specification language for defining policies that might also depend on how many times a function is applied to a given value. This specification language seems more expressive than Rocha *et al.* [47], although the properties being enforced have not been formally characterized. We believe restrictions defined using any of these policy graphs could be described using a set of RIF automata.

Li and Zdancewic [39] suggest that downgrades between two security levels be specified as lambda terms. Apply one of these lambda terms to a value and the result, by definition, is given a downgraded label. (A type system is given to enforce these policies.) This approach to characterizing downgrades is attractive because it is independent from the program code. Enforcement, however, involves a conservative approach to deciding equivalence of functions because that problem is undecidable in general. Also, the approach is

[17]Trusted methods are similar to *trusted subjects*, first introduced by Bell and La-Padula [8] to handle declassification.

not well suited for handling reclassifications based on how a value has been derived, whereas RIF labels do handle that.

Extending Noninterference for Reclassification

Declassification violates classical noninterference [27], which has prompted researchers to develop alternatives. One example is *conditional noninterference* [27, 26]. It proscribes secret information flows to public information, unless a given predicate is satisfied by (i) the sequence of operations involved in this flow and (ii) the principals invoking those operations. For declassification, conditional noninterference is more expressive than PWNI, because PWNI ignores identities of the principals that invoke operations. PWNI, however, handles reclassification in full generality (i.e., arbitrary lattices of labels and declassification as well as classification), in contrast to conditional noninterference. which only deals with declassification in a 2-level lattice.

Gradual release (GR) [3] dictates that declassification of expressions, encrypted exceptions, and released cryptographic keys are the only execution points where an attacker's knowledge about initial secret values may increase. Other work (*delimited release* [51] and *relaxed noninterference* [39]) specifies what expressions of secret values in the initial state could be declassified, with no restriction about where in the program such expressions are declassified.

PWNI supports declassifications of arbitrary expressions on any secret within the program. For this reason, the program below, which is arguably secure, is accepted by PWNI, but rejected by delimited release [51] (and its extensions, such as localized delimited release [4]):

$$h := avg(h_1, h_2, h_3);\ l := declassify(h) \tag{53}$$

Delimited release (and its extensions) rejects the above program for the same reason it rejects the following program:

$$h' := h;\ l := declassify(h') \tag{54}$$

And the reason is to prevent *laundering attacks*—the declassification of h' inadvertently causing the declassification of h.

PWNI and RIF labels prevent laundering attacks by construction. If a variable is allowed to be declassified (e.g., h' in (54)), then this variable will be tagged with a RIF label that specifies declassification; if a variable is not allowed to be declassified (e.g., h in (54)), then then this variable will be

tagged with a RIF label that does not specify declassification. So, in (54), the RIF label of h' will not be as restrictive as the RIF label of h, and thus, PWNI will be violated by assignment $h' := h$ and the entire program will be rightfully rejected. On the other hand, PWNI rightfully accepts program (53), when all variables h_1, h_2, h_3, h are tagged with a RIF label that specifies such a declassification.

The flexibility that PWNI offers to be able to handle declassifications of any secret expression in the program (and ultimately accept (53)) is a useful advantage. This is because usually the value that we desire to be ultimately declassified is the result of consecutive computations (i.e., an arbitrary expression in the program)—not the result of one clean expression on initial secrets. Those cases can be handled by PWNI, but not by delimited release (and its extensions).

The cost of the flexibility that PWNI offers comes from formally specifying *what* is being declassified and *where* in the program (similar targets to localized delimited release). Specifying *what* is being declassified in terms of initial secrets is in general undecidable when declassifying an arbitrary expression in the program. Consequently we choose to specify *what* is being declassified with respect to secrets that exist at the program point (i.e., command) where declassification is triggered. So, we set the reference point of the declassification to be the program point where this declassification is triggered. One way to select a program point as a reference point is to demand λ-pieces end and start at that same program point (just as the beginning of a program is regarded as a reference point for noninterference, demanding all executions to start from there). For this reason PWNI requires exact code equality at the ends of λ-pieces. And with exact code equality, the *where* dimension of the declassification is identified, too, because the program point at which an expression is declassified becomes explicit.

With *conditional gradual release* [7], as with our approach, any expression in a program may be declassified. However, conditional gradual release allows declassifications to depend on secret guards, causing a declassification to disclose more information than might have been intended. (PWNI does not suffer from this defect, because of the way λ-pieces are defined.)

Non-disclosure policy [42] is a variation of noninterference for handling local declassification constructs. These declassification constructs augment allowed flows during execution of some code M, restoring the previously allowed flows once execution leaves M. To satisfy the non-disclosure policy, noninterference must hold for flow relations that are allowed at every execu-

tion step. This policy, then, embodies a different design choice from PWNI about the scope of a declassification—with PWNI the declassification persists to the end of the execution, whereas with the non-disclosure policy it ends when the declassification construct has completed. The same design choice is adopted in [12] for handling flow-locks [11].

Other alternatives to noninterference associate declassification with special commands, such as match queries[18] [60], one-way functions [58], or various other cryptographic operations. With *computational noninterference* (CNI) [36], disclosure of secret values is permitted only when those values are encrypted; CNI is enforced by type systems introduced in Laud *et al.* [38] for *passive* and Fournet *et al.* [25] for *active* adversaries. Smith *et al.* [55] propose a variant of noninterference that handles both encryption and decryption, and it is enforced using a type system.

κ-labels demonstrate how RIF labels can handle cryptographic operations by treating these operations as special and translating them into reclassifying expressions. The literature on type checking of cryptographic operations has explored two general approaches [17]: computational and symbolic. κ-labels embrace the symbolic approach, using an analysis approach derived from Dolev-Yao [22]. *Cryptographically masked flows* [2] also employs a symbolic analysis, and it too is enforced by type systems. Laud [37] shows that type correctness according to [2], together with some additional conditions, imply CNI, thereby establishing a connection between cryptographically masked flows (which is based on symbolic analysis) and CNI (which is based on computational analysis). Cortier *et al.* [17] generalizes this connection by showing that programs secure according to a symbolic analysis are also secure according to a computational analysis.

Semantic properties have been proposed to handle erasure, too. Del Tedesco *et al.* [18] use knowledge semantics to express a hierarchy of erasure policies. These erasure policies are categorized based on (i) whether they specify erasure of all or part of the information, and (ii) whether erasure depends on program state (either high or low). Erasure is a form of classification, and thus, can be specified by RIF labels. However, with the RIF labels in this paper, erasure only affects a value being derived and cannot be formulated to depend on program state.

Noninterference and its variations (including PWNI) characterize allowed

[18]A *match query* checks whether two objects are equal. For example, a match query is use to check whether a certain string is the password of a given user.

flows of information; they do not handle required flows. Chong [15] gives a semantic definition for *required information release* policies and presents a static type system to enforce these policies. A required information release policy specifies what information should be released and how this information can be learned by the authorized observers. The semantics is based on algorithmic knowledge.

Views of the Reclassification Landscape

The survey by Sabelfeld *et al.* [53] introduces a four-dimension categorization for declassification policies (though the categorization seems applicable reclassification, too): *what* information is declassified, *who* declassifies information, *where* in the system information is declassified, and *when* information can be declassified. RIF labels and our reclassifying expressions specify *what*, *where*, and *when*, but not *who.* The *what* is the value produced by the expression; *where* is the position of the reclassifying expression in the program text; *when* is determined by the program's control flow.

Nothing prevents the semantics of our reclassifiers from incorporating information about *who* is evaluating a given reclassifying expression. The *decentralized label model* (DLM) [45, 44] is an obvious starting point for such an extension. According to DLM, a value may be declassified only if the declassification command is executed on behalf of the value's *owner* or on behalf of a principal that *acts-for* that owner. To adopt this approach, we would add an additional input argument to $\mathcal{T}$—the identity of the principal undertaking the reclassification. The semantics of $\mathcal{T}$ would then be extended so that a reclassifier triggers a transition only for certain principals.

Broberg *et al.* [14] offers an orthogonal view for information flow policies and declassification. It is based on a three-level *hierarchy of control. Level 0 control* is a set of possible *flow relations* between information *sources* (e.g., input variables) and *sinks* (e.g., output channels). A flow relation indicates that information from the source is allowed to flow to the sink. *Level 1 controls* select which flow relations are allowed. *Level 2 controls* constitute a meta policy for controlling the way in which the current flow relations (Level 1 controls) may be changed. RIF label function $\mathcal{T}$ incorporates aspects of *Level 1* and *Level 2* controls.

Acknowledgments

We appreciate the comments we received on early discussions of this work from Eleanor Birrell, Michael Clarkson, Josée Desharnais, Michael George, José Meseguer, Brad Martin, Andrew Myers, David Naumann, Karn Seth, Geoffrey Smith, and Nadia Tawbi. Steve Chong and Deepak Garg, in addition, provided detailed and very helpful feedback on an earlier draft of this paper.

A $\langle \Lambda_{\mathrm{RA},}, \sqsubseteq_{\mathrm{RA}}, \sqcup_{\mathrm{RA}} \rangle$ is a join semilattice

To prove that $\langle \Lambda_{\mathrm{RA},}, \sqsubseteq_{\mathrm{RA}}, \sqcup_{\mathrm{RA}} \rangle$ is a join semilattice requires showing (i) $\lambda_\alpha \sqsubseteq_{\mathrm{RA}} \lambda_\alpha \sqcup_{\mathrm{RA}} \lambda_{\alpha'}$, (ii) $\lambda_{\alpha'} \sqsubseteq_{\mathrm{RA}} \lambda_\alpha \sqcup_{\mathrm{RA}} \lambda_{\alpha'}$, and (iii) that $\lambda_\alpha \sqcup_{\mathrm{RA}} \lambda_{\alpha'}$ is the least upper bound.

To prove (i), it suffices to use decision procedure (15) for $\sqsubseteq_{\mathrm{RA}}$, and prove:

$$(\forall \langle q, \langle q, q'' \rangle \rangle \in Q_{\alpha \otimes (\alpha \sqcup \alpha')}\colon\ \rho_\alpha(q) \sqsubseteq_R \rho_{\alpha \sqcup \alpha'}(\langle q, q'' \rangle)), \tag{55}$$

because reachability condition (13) for $\lambda_\alpha \otimes (\lambda_\alpha \sqcup_{\mathrm{RA}} \lambda_{\alpha \sqcup \alpha'})$ implies that if

$$\langle q, \langle q', q'' \rangle \rangle \in Q_{\alpha \otimes (\alpha \sqcup \alpha')}$$

holds then $q = q'$. Completing the proof of (55) is just a matter of expanding the definition of $\rho_{\alpha \sqcup \alpha'}(\langle q, q'' \rangle)$.

The proof of (ii) is similar.

The proof of (iii) involves showing that $\lambda_\alpha \sqcup_{\mathrm{RA}} \lambda_{\alpha'}$ is the least RIF label satisfying

$$\lambda_\alpha \sqsubseteq_{\mathrm{RA}} \lambda_\alpha \sqcup_{\mathrm{RA}} \lambda_{\alpha'}$$
$$\lambda_{\alpha'} \sqsubseteq_{\mathrm{RA}} \lambda_\alpha \sqcup_{\mathrm{RA}} \lambda_{\alpha'}$$

Our proof by contradiction derives *false* from the assumption that a RIF label $\lambda_{\alpha''}$ satisfying $\lambda_\alpha \sqsubseteq_{\mathrm{RA}} \lambda_{\alpha''}$, $\lambda_{\alpha'} \sqsubseteq_{\mathrm{RA}} \lambda_{\alpha''}$, and $\lambda_{\alpha''} \sqsubset_{\mathrm{RA}} \lambda_\alpha \sqcup_{\mathrm{RA}} \lambda_{\alpha'}$ exists.

From $\lambda_\alpha \sqsubseteq_{\mathrm{RA}} \lambda_{\alpha''}$ we have $\mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda_\alpha, F)) \sqsubseteq_{\mathrm{R}} \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda_{\alpha''}, F))$ for any F; from $\lambda_{\alpha'} \sqsubseteq_{\mathrm{RA}} \lambda_{\alpha''}$ we have $\mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda_{\alpha'}, F)) \sqsubseteq_{\mathrm{R}} \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda_{\alpha''}, F))$ for any F. Since $\langle R, \sqcup_R, \sqsubseteq_R \rangle$ defines a lattice, we conclude

$$\mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda_\alpha, F)) \sqcup_R \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda_{\alpha'}, F)) \sqsubseteq_R \mathcal{R}_{\mathrm{RA}}(\mathcal{T}_{\mathrm{RA}}(\lambda_{\alpha''}, F)). \tag{56}$$

From assumption $\lambda_{\alpha''} \sqsubset_{\mathrm{RA}} \lambda_\alpha \sqcup_{\mathrm{RA}} \lambda_{\alpha'}$ we conclude (i) $\lambda_{\alpha''} \sqsubseteq_{\mathrm{RA}} \lambda_\alpha \sqcup_{\mathrm{RA}} \lambda_{\alpha'}$ and (ii) $\lambda_{\alpha''} \neq \lambda_\alpha \sqcup_{\mathrm{RA}} \lambda_{\alpha'}$. But (56) conjoined with (i) contradicts (ii).

B Definition of $\mathbb{T}(\lambda, \mathcal{C})$

$\mathbb{T}(\lambda, \mathcal{C})$ is defined based on a deterministic generator G of fresh variables, which store lists of values. Specifically, $G.init$ sets G to its initial state, and $G.fresh([\mathcal{E}]_{\bar{\mathsf{f}}})$ returns the next fresh variable $h_{\mathcal{E}}$. Each variable $h_{\mathcal{E}}$ stores a list of values with the same type as $\mathcal{E}$, and it is tagged with a RIF label that satisfies $\Gamma(h_{\mathcal{E}}) = \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}})$.

$\mathbb{T}(\lambda, \mathcal{C})$ substitutes each upgrading expression $[\mathcal{E}]_{\bar{\mathsf{f}}}$ in $\mathcal{C}$ with expression $next(h_{\mathcal{E}})$. So, we extend the evaluation and the RIF labels for expressions $\mathcal{E}$ to accommodate this new expression $next(h_{\mathcal{E}})$. In particular, we define successive $\mathcal{M}(next(h_{\mathcal{E}}))$ evaluations to return successive elements of the list stored in $h_{\mathcal{E}}$. And define $\Gamma(next(h_{\mathcal{E}}))$ to be $\Gamma(h_{\mathcal{E}})$.

We now give the formal definition for $\mathbb{T}(\lambda, \mathcal{C})$, which employs auxiliary procedures T and S.

$$\mathbb{T}(\lambda, \mathcal{C}) \triangleq T(\lambda, \mathcal{C}, G.init)$$

$$\begin{aligned}
T(\lambda, x := [\mathcal{E}]_{\bar{\mathsf{f}}}, G) &\triangleq x := S(\lambda, [\mathcal{E}]_{\bar{\mathsf{f}}}, G) \\
T(\lambda, \textbf{if } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ then } \mathcal{C}_t \textbf{ else } \mathcal{C}_e, G) &\triangleq \textbf{if } S(\lambda, [\mathcal{E}]_{\bar{\mathsf{f}}}, G) \textbf{ then } T(\lambda, \mathcal{C}_t, G) \\
&\qquad \textbf{else } T(\lambda, \mathcal{C}_e, G) \\
T(\lambda, \textbf{while } [\mathcal{E}]_{\bar{\mathsf{f}}} \textbf{ do } \mathcal{C}_w, G) &\triangleq \textbf{while } S(\lambda, [\mathcal{E}]_{\bar{\mathsf{f}}}, G) \textbf{ do } T(\lambda, \mathcal{C}_w, G) \\
T(\lambda, \mathcal{C}_1; \mathcal{C}_2, G) &\triangleq T(\lambda, \mathcal{C}_1, G); T(\lambda, \mathcal{C}_2, G)
\end{aligned}$$

$$S(\lambda, [\mathcal{E}]_{\bar{\mathsf{f}}}, G) \triangleq \begin{cases} [next(G.fresh([\mathcal{E}]_{\bar{\mathsf{f}}}))]_{\epsilon}, & \text{if } \Gamma(\mathcal{E}) \sqsubseteq \lambda \\ & \quad \wedge \Gamma([\mathcal{E}]_{\bar{\mathsf{f}}}) \not\sqsubseteq \lambda \\ [\mathcal{E}]_{\bar{\mathsf{f}}}, & \text{otherwise} \end{cases}$$

There is a reason why upgrading expressions are substituted by variables that store lists of values instead of variables that store ordinary (scalar) values: upgrading expressions in **while** commands. When a **while** command involves an upgrading expression, then this expression might yield as many upgraded values as the the number of iterations of that command. All these values should be protected against leaking to certain principals. So, we substitute them with arbitrarily chosen values (which are elements of these lists) and ensure that they do not influence observations of these principals.

Notice that $\mathbb{T}(\lambda, \mathcal{C})$ is deterministic, by construction. This means that for the same command $\mathcal{C}$, the translated command $\mathbb{T}(\lambda, \mathcal{C})$ is always the same. So, comparing traces $\tau = \Upsilon(\, \mathbb{T}(\lambda, \widehat{\mathcal{C}}),\, \mathcal{M})$ and $\tau' = \Upsilon(\, \mathbb{T}(\lambda, \widehat{\mathcal{C}}),\, \mathcal{M}')$ in PWNI definition (35) is meaningful, because these traces correspond to the exact same translated command.

References

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, pages 107–121, 2003.

[2] A. Askarov, D. Hedin, and A. Sabelfeld. Cryptographically-masked flows. *Theor. Comput. Sci.*, 402(2-3):82–101, July 2008.

[3] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 207–221, 2007.

[4] A. Askarov and A. Sabelfeld. Localized delimited release: Combining the what and where dimensions of information release. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, PLAS '07, pages 53–60, New York, NY, USA, 2007. ACM.

[5] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 165–178, New York, NY, USA, 2012. ACM.

[6] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama. Faceted execution of policy-agnostic programs. In *Proceedings of the 8th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '13, pages 15–26, New York, NY, USA, 2013. ACM.

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

[8] D. Bell and L. La Padula. Secure computer systems: Unified exposition and MULTICS interpretation. Technical Report ESD-TR-75306, Bedford, MA, 1976.

[9] E. Birrell and F. B. Schneider. A reactive approach to use-based privacy. Technical Report 54843, Cornell University, Computing and Information Science, November 2017.

[10] N. Broberg, B. Delft, and D. Sands. Paragon for practical programming with information-flow control. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems*, volume 8301, pages 217–232, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[11] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 180–196, Berlin, Heidelberg, 2006. Springer-Verlag.

[12] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. Technical report, Chalmers University of Technology and Göteborgs University, May 2006.

[13] N. Broberg and D. Sands. Paralocks: Role-based information flow control and beyond. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 431–444, New York, NY, USA, 2010. ACM.

[14] N. Broberg, B. van Delft, and D. Sands. The anatomy and facets of dynamic policies. In *Proceedings of the 28th IEEE Computer Security Foundations Symposium*, pages 122–136, July 2015.

[15] S. Chong. Required information release. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium*, pages 215–227, Piscataway, NJ, USA, July 2010. IEEE Press.

[16] S. Chong and A. C. Myers. End-to-end enforcement of erasure and declassification. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, pages 98–111, June 2008.

[17] V. Cortier, S. Kremer, and B. Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *J. Autom. Reason.*, 46(3-4):225–259, Apr. 2011.

[18] F. Del Tedesco, S. Hunt, and D. Sands. A semantic hierarchy for erasure policies. In *Proceedings of the 7th International Conference on Information Systems Security*, ICISS'11, pages 352–369, Berlin, Heidelberg, 2011. Springer-Verlag.

[19] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[20] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, July 1977.

[21] D. E. R. Denning. *Secure information flow in computer systems.* PhD thesis, Purdue University, West Lafayette, IN, USA, 1975.

[22] D. Dolev and A. C. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, Mar 1983.

[23] E. Elnikety, D. Garg, and P. Druschel. SHAI: Enforcing Data-Specific Policies with Near-Zero Runtime Overhead. Technical report, Max Planck Institute for Software Systems, Saarland Informatics Campus, Germany, January 2018.

[24] E. Elnikety, A. Mehta, A. Vahldiek-Oberwagner, D. Garg, and P. Druschel. Thoth: Comprehensive policy compliance in data retrieval systems. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, pages 637–654. USENIX Association, 2016.

[25] C. Fournet and T. Rezk. Cryptographically sound implementations for typed information-flow security. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 323–335, New York, NY, USA, 2008. ACM.

[26] J. A. Goguen and J. Mesegue. Unwinding and inference control. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 75–87, 1984.

[27] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[28] 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.

[29] H. R. Hartson and D. K. Hsiao. Full protection specifications in the semantic model for database protection languages. In *Proceedings of the 1976 Annual Conference*, ACM '76, pages 90–95, New York, NY, USA, 1976. ACM.

[30] B. Hicks, D. King, P. McDaniel, and M. Hicks. Trusted declassification: High-level policy for a security-typed language. In *Proceedings of the Workshop on Programming Languages and Analysis for Security*, PLAS '06, pages 65–74, New York, NY, USA, 2006. ACM.

[31] T. H. Hinke. Inference aggregation detection in database management systems. In *Proceedings. 1988 IEEE Symposium on Security and Privacy*, pages 96–106, April 1988.

[32] 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*, PLDI '15, pages 291–302, New York, NY, USA, 2015. ACM.

[33] E. Kozyri. *Enhancing Expressiveness of Information Flow Labels: Reclassification and Permissiveness*. PhD thesis, Cornell University, Ithaca, New York, USA, 2018. `https://search.proquest.com/docview/2167492985?accountid=10267`.

[34] E. Kozyri, O. Arden, A. C. Myers, and F. B. Schneider. JRIF: Reactive Information Flow Control for Java. Technical report, Cornell Univarsity, February 2016.

[35] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In

Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07, pages 321–334, New York, NY, USA, 2007. ACM.

[36] P. Laud. Semantics and program analysis of computationally secure information flow. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, ESOP '01, pages 77–91, London, UK, 2001. Springer-Verlag.

[37] P. Laud. On the computational soundness of cryptographically masked flows. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 337–348, New York, NY, USA, 2008. ACM.

[38] P. Laud and V. Vene. A type system for computationally secure information flow. In *Proceedings of the 15th International Conference on Fundamentals of Computation Theory*, FCT'05, pages 365–377, Berlin, Heidelberg, 2005. Springer-Verlag.

[39] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 158–170, New York, NY, USA, 2005. ACM.

[40] P. Li and S. Zdancewic. Practical information-flow control in web-based information systems. In *Proceedings of the 18th IEEE Workshop on Computer Security Foundations*, CSFW '05, pages 2–15, Washington, DC, USA, 2005. IEEE Computer Society.

[41] A. Lux and H. Mantel. Declassification with explicit reference points. In M. Backes and P. Ning, editors, *Computer Security – ESORICS 2009*, pages 69–85, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[42] A. A. Matos and G. Boudol. On declassification and the non-disclosure policy. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 226–240, June 2005.

[43] K. Micinski, J. Fetter-Degges, J. Jeon, J. S. Foster, and M. R. Clarkson. Checking interaction-based declassification policies for Android using symbolic execution. In *Proceedings of the European Symposium on Research in Computer Security*, ESORICS 2015, pages 520–538, Cham, 2015. Springer International Publishing.

[44] A. C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.

[45] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, SOSP '97, pages 129–142, New York, NY, USA, 1997. ACM.

[46] 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.

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

[48] 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.

[49] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 63–74, New York, NY, USA, 2009. ACM.

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

[51] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proceedings of the International Symposium on Software Security (ISSS'03), volume 3233 of LNCS*, pages 174–191. Springer-Verlag, 2004.

[52] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, ESOP '99, pages 40–58, Berlin, Heidelberg, 1999. Springer-Verlag.

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

[54] 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.

[55] G. Smith and R. Alpízar. Secure information flow with random assignment and encryption. In *Proceedings of the 4th ACM Workshop on Formal Methods in Security*, FMSE '06, pages 33–44, New York, NY, USA, 2006. ACM.

[56] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 95–106, New York, NY, USA, 2011. ACM.

[57] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, Jan. 1986.

[58] D. Volpano. Secure introduction of one-way functions. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 246–254, 2000.

[59] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, Jan. 1996.

[60] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 268–276, New York, NY, USA, 2000. ACM.

[61] D. S. Wallach, J. A. Roskind, and E. W. Felten. Flexible, extensible Java security using digital signatures. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 38:59–74, Dec. 1996.