Managing Policy Updates in Security-Typed Languages

Nikhil Swamy Michael Hicks
University of Maryland

Stephen Tse Steve Zdancewic
University of Pennsylvania

Abstract

This paper presents RX*, a new security-typed programming language with features intended to make the management of information-flow policies more practical. Security labels in* RX*, in contrast to prior approaches, are defined in terms of* owned roles*, as found in the RT role-based trust-management framework. Role-based security policies allows flexible delegation, and our language* RX *provides constructs through which programs can robustly update policies and react to policy updates dynamically. Our dynamic semantics use statically verified* transactions *to eliminate illegal information flows across updates, which we call* transitive flow*. Because policy updates can be observed through dynamic queries, policy updates can potentially reveal sensitive information. As such,* RX *considers policy statements themselves to be potentially confidential information and subject to information-flow* metapolicies.

1 Introduction

A large body of work has demonstrated that type systems and other forms of static analysis can prove the absence of undesirable information flows within a program [21]. *Security-typed programming languages* extend standard types with labels that specify *security policies* on the allowable uses of typed data. Such labels are typically ordered by a lattice that expresses multi-level security policies for properties like confidentiality. For example, labels may denote principals like Bob and $Alice$, and if, according to the security lattice, $Alice \sqsubseteq Bob$ holds, then any data labeled $Alice$ can be viewed by Bob. In security-typed languages, compile-time type-checking ensures that the policies expressed by labels are enforced, and thus one can prove, in advance of program execution, that a program adheres to a particular information-flow policy.

Most existing security-typed languages assume that a program's security policy does not change once the program begins its execution. This is an unrealistic assumption for long-running programs. For operating systems, network servers, and database systems, the privileges of principals are likely to change, such as new principals entering a system, or existing principals leaving or changing duties.

On the other hand, it would be unwise to simply allow the policy to change at arbitrary program points. For example, if the program is unaware of a revocation in the security lattice it could allow a principal to view data illegally. More subtly, a combination of policy changes could violate separation of duty, inadvertently allowing a principal to view data that he is permitted to see by neither the old nor the new policy. We call this channel of information leaks across updates *transitive flow*.

This paper presents a new security-typed language RX that permits security policies to change during program execution. RX has two distinguishing features. First, labels in RX are defined in terms of *roles* as found in the role-based access control languages of the RT framework [14]. A role names a set of principals, and role ordering in the security lattice is defined by subset. Second, RX programs are permitted to dynamically update the current role definitions and dynamic queries can be used to observe policy changes beyond that point. Programmers can use database-style transactions to denote code that must use a single consistent policy, preventing unintended information flows across updates. Policy updates that would violate this consistency cause the program to roll back to a consistent state.

In designing RX we had to consider how to manage changes to policy most effectively. We found that once we allow policies to change within a program, policies themselves can become conduits of sensitive information, or conversely, sensitive information can be revealed by changes to policy. To prevent these situations, RX uses *metapolicies* that define which principals can view a particular role, and which principals trust a role's definition. The inherent administrative model of the RT policy languages suggests natural choices for these metapolicies. For example, in the RT framework, a role has a *designated owner* that is responsible for administering the role's contents. That is, only when the program is acting in a way trusted by that owner may the role be changed.

The RT policy language has other useful features that ease the administration of policy in use by a security-typed program. RT supports fine-grained delegation which can limit the impact of policy changes on information flows. Also, using named roles as labels provides a useful level of indirection: the constituents of a role may change when the name of the role does not. This may reduce the need for data to be relabeled to effect a policy change. As far as we are aware, RX is the first programming language to employ a role-based specification language for defining security policies.

The rest of this paper is structured as follows. Section 3 presents RX_{core}, the mostly-standard core of RX for which security labels are defined as RT roles. Section 3 presents the full RX language, which extends the RX_{core} label model to support the added features of policy queries, policy updates, and transactions. Section 4 states security theorems that hold for RX. The paper concludes with a discussion of related work in Section 5 and future directions in Section 6.

2 A Role-based Security-Typed Language

We begin with by motivating using roles as labels in a language that supports policy updates. We follow this with a presentation of the core features of RT_0, the simplest member of the RT family of role-based policy languages. Finally, we present RX_{core}, an imperative security-typed language for which security labels are defined as roles.

2.1 Existing Label Models

Most existing security-typed languages use the *lattice model of information flow* [28] in which an information flow policy is defined by a lattice $(\mathcal{L}, \sqsubseteq)$, where $\ell \in \mathcal{L}$ is a *label* (or *security level*), and labels are ordered by the relation $\sqsubseteq$. A typical choice (e.g., in FlowCaml [23]) is to define the members of $\mathcal{L}$ as atomic names, and to define $\sqsubseteq$ by a policy Π that defines the ordering among the names. This kind of label model allows a program to define labels like L and H, which mean "low" and "high" security, respectively, and a policy $\Pi = \{L \sqsubseteq H\}$, which indicates that L is less restrictive than H. Generally speaking, labels can either be *atomic*—H and L in this example—or the join $\ell_1 \sqcup \ell_2$ of labels ℓ_1 and ℓ_2; here $\sqcup$ is induced by the $\sqsubseteq$ relation.

The language Jif [16] supports the more sophisticated labels of the *decentralized label model* (DLM). DLM labels are defined in terms of *principals*, and have three parts: an owner, a reader set (those principals allowed to read the value), and an integrity set (those principals who trust the value). Jif policies Π define delegation relationships between principals: if $\Pi \vdash P_1 \sqsubseteq P_2$ then P_2 may "act for" P_1. The ordering on labels is induced by this ordering among principals. For example, any data labeled solely by owner P_1 may be read or written by P_2 (as well as any principals which may act for P_2). DLM labels have been used in a number of interesting ways such as supporting distributed computation [32].

2.2 Motivations for Roles

The problem with these label models is that they offer no administrative support for changes to policy. This is not surprising because existing languages typically make the simplifying assumption that the security policy is fixed for the duration of a program's execution. If policy updates are to be supported, a reasonable administrative model should be able to provide answers to the following questions. (1) *Who* is allowed to make changes to the security policy? (2) *What* parts of the policy are permitted to change? (3) *How* should those changes be reflected in the running program? (4) *When* are such changes permitted to take place?

Rather than develop an administrative model for the DLM, we looked instead to the body of work on formal policy languages for which administrative models already exist. *Role-based* policy languages [19, 2, 7, 14] in particular suggest a natural label model. In particular, a *role*, which is a name that represents a set of principals, can be treated as a label, and the ordering between labels can be defined in terms of subset on the contents of roles according to the policy. Indeed, in the simple example above, the two atomic labels L and H are essentially being treated as roles.

We chose to use RT_0 as the core of the label model for RX. RT_0 is the simplest member of the role-based policy language framework RT [14]. Using RT roles as labels has a number of attractive administrative features:

1. *Ownership*: An RT role defined as having an *owner* responsible for the role's definition; a given principal can own many roles. Only a role's owner is allowed to change the definition of that role.

2. *Membership and Delegation*: An RT policy permits delegation at the granularity of roles, in which one role may be defined in part by the contents of another role. This provides better control than the DLM, which only permits delegation between principals. To see the distinction, say that in Jif we define a special principal $Manager$ that represents the role of Manager in a company. To express that $Alice$ is a member of this role, a DLM policy Π would include the statement $Manager \sqsubseteq Alice$; e.g., whatever a Manager can view, Alice can view as well. Assuming an administrative model that would allow $Alice$ to delegate to whomever she wishes, $Alice$ can state that $Alice \sqsubseteq Bob$, with the effect of making Bob a manager since $Manager \sqsubseteq Alice \sqsubseteq Bob$. By contrast, role membership and role delegation in RT are separate concepts. Roles have an owner, and membership is strictly under the owner's control: the owner can either include a principal in a

role, or delegate (part of) the definition of a role to another role. Membership does not imply delegation.

3. *Indirection*: Defining labels as roles provides a useful level of indirection because the membership of a role may change while the label on data stays the same. That is, a security policy of a data can be modified without requiring the data to be relabeled.

These points taken together answer the first three of the four questions posed previously. The question (4) of when policy changes are allowed to occur depends on what the program is doing when a proposed update is available; we consider this question in the next section.

For the remainder of this section, we first present the RT_0 policy language that forms the core of our label model. Then we present the syntax and typing rules of the Rx$_{\textsf{core}}$, the core of our full language Rx, which uses RT_0 roles for security labels.

2.3 RT_0: A Role-based Policy Language

RT_0 is the simplest member of the RT framework of role-based policy languages [14]; it is summarized in Figure 1. A role ρ in RT_0 has the form $P.r$, where principal P is the role's owner and r is the role's name. We often write A, B, etc. as sample principals P. We use predicate $owner(\rho)$ to extract the owner of ρ (and thus $owner(P.r) = P$).

Policy statements s have two forms[1] $P.r \leftarrow \{P_1, \dots, P_n\}$ and $P_1.r_1 \leftarrow P_2.r_2$. The first form indicates simple membership, that principals P_i are members of role $P.r$. The second form is a simple role delegation statement, which indicates that all members of the role $P_2.r_2$ are also members of $P_1.r_1$. We use the predicate $roledef(s)$ to denote the role ρ defined by the policy statement s: for example, $roledef(A.r \leftarrow \{B\})$ is $A.r$.

The semantics of a role ρ is a set of principals and is determined by the function $[\![\cdot]\!]_\Pi$ according to a policy Π. (Li et al [14] develop a formal definition of this semantics function.) Intuitively, $[\![\rho]\!]_\Pi$ is defined by all of the simple members P_i of ρ where $\rho \leftarrow P_i \in \Pi$, along with the members $[\![\rho_i]\!]_\Pi$ where $\rho \leftarrow \rho_i \in \Pi$.

An example of an RT_0 policy Π is given in Figure 2, which models the privacy of a patient's health care documents. The example defines roles owned by three principals: Pat, a patient; *Clinic*, a specialized medical treatment center where Pat is currently a patient; and *DrPhil*, a doctor not affiliated with the clinic. The policy statements define several roles that capture the affiliations just mentioned. Pat.doctors is defined via two statements. The first says that *DrSue* (a family doctor) is Pat's doctor. The second statement is a delegation to *Clinic*.staff, indicating that Pat's doctors also include the practitioners that work at the clinic, which according to the policy in Figure 2, is currently just the two principals *DrAlice* and *DrBob*. Pat.insurers includes all insurance companies with which Pat has a policy—this is the single company *BCBS* defined through simple membership. *Clinic*.insuranceCos is the set of insurance companies accepted by the clinic. Finally, the last role definition owned by *DrPhil* includes only himself.

[1] RT_0 also includes *intersection* and *linking inclusion*: statements which are supported by our label model, but which we elide for simplicity.

principal	P		
role name	r		
role	ρ	$::=$	$P.r$
policy stmt	s	$::=$	$\rho \leftarrow \{P_1, \dots, P_n\}$
		$\mid$	$\rho_1 \leftarrow \rho_2$
policy	Π	$::=$	$\{s_1, \dots, s_n\}$
role semantics	$[\![\rho]\!]_\Pi$		(see text)

Figure 1. The syntax of RT_0 policy language.

Pat.doctors	$\leftarrow$	$\{DrSue\}$
Pat.doctors	$\leftarrow$	*Clinic*.staff
Pat.insurers	$\leftarrow$	$\{BCBS\}$
Pat.healthRecords	$\leftarrow$	Pat.doctors
Clinic.staff	$\leftarrow$	$\{DrAlice, DrBob\}$
Clinic.insuranceCos	$\leftarrow$	$\{BCBS, Aetna\}$
DrPhil.self	$\leftarrow$	$\{DrPhil\}$

Figure 2. A sample health care policy in RT_0.

For instance, the semantics of the role Pat.doctors and of Pat.insurers according to this sample policy are:

$$[\![Pat.\text{doctors}]\!]_\Pi = \{DrAlice, DrBob, DrSue\}$$
$$[\![Pat.\text{insurers}]\!]_\Pi = \{BCBS\}$$

2.4 The Rx$_{\textsf{core}}$ Programming Language

Rx$_{\textsf{core}}$ is a simple imperative language with security labels. Its syntax is shown at the top of Figure 3. Labels ℓ in Rx$_{\textsf{core}}$ are either atomic labels L or the join of two labels according to the lattice ordering. An atomic label is merely a role ρ. Labels are ordered according to the judgment $\Pi \vdash \ell_1 \sqsubseteq \ell_2$, where Π is an RT_0 policy as described above. For atomic labels, this ordering is according to the semantics of roles as sets:

$$\Pi \vdash \rho_1 \sqsubseteq \rho_2 \iff [\![\rho_2]\!]_\Pi \subseteq [\![\rho_1]\!]_\Pi$$

Note that the label ordering relation ($\sqsubseteq$) is the *reverse* of the subset relation ($\subseteq$) over role membership. That is, a role that has a larger set of members lower security than a role with fewer members, since strictly more principals can read

atomic labels	L	$::=$	ρ
compound labels	ℓ	$::=$	$L \mid \ell \sqcup \ell$
policy context	Q	$::=$	Π
typing context	Ω	$::=$	(Γ, pc, Q)
expressions	E	$::=$	$i \mid \mathtt{x} \mid E_1 + E_2$
statements	S	$::=$	$\mathtt{skip} \mid \mathtt{x} := E \mid S_1; S_2$
			$\mid \mathtt{while}\ (E)\ S \mid \mathtt{if}\ (E)\ S_1\ S_2$

$$\Omega \vdash i : \ell \qquad \Omega \vdash \mathtt{x} : \Omega.\Gamma(\mathtt{x}) \qquad \Omega \vdash \mathtt{skip}$$

$$\frac{\Omega \vdash S_1 \quad \Omega \vdash S_2}{\Omega \vdash S_1; S_2} \qquad \frac{\Omega \vdash E_1 : \ell_1 \quad \Omega \vdash E_2 : \ell_2}{\Omega \vdash E_1 + E_2 : \ell_1 \sqcup \ell_2}$$

$$\frac{\Omega \vdash E : \ell' \quad \Omega[\mathrm{pc} = \Omega.\mathrm{pc} \sqcup \ell'] \vdash S_i \quad i \in \{1,2\}}{\Omega \vdash \mathtt{if}\ (E)\ S_1\ S_2}$$

$$\frac{\Omega \vdash E : \ell' \quad \Omega[\mathrm{pc} = \Omega.\mathrm{pc} \sqcup \ell'] \vdash S}{\Omega \vdash \mathtt{while}\ (E)\ S}$$

$$\frac{\Omega \vdash \mathtt{x} : \ell \quad \Omega \vdash E : \ell_1 \quad \Omega.Q \vdash \ell_1 \sqsubseteq \ell \quad \Omega.Q \vdash \Omega.\mathrm{pc} \sqsubseteq \ell}{\Omega \vdash \mathtt{x} := E}$$

Figure 3. $\textsc{Rx}_{\textsf{core}}$ **syntax and typing.**

data labeled by it. Extending this ordering to compound labels is straightforward by interpreting the join operator as set intersection.

There are two typing judgments for $\textsc{Rx}_{\textsf{core}}$, shown at the bottom of Figure 3. Expression typings $\Omega \vdash E : \ell$ state that in context Ω the expression E has a security level ℓ. Statement typings $\Omega \vdash S$ state that statement S is well formed with respect to the context Ω. The context Ω has three elements: the *environment* Γ, the *program counter label* pc and the *policy context* Q. Here Γ is a map from variables to their security labels, and pc is simply a label ℓ that is used to bound the effect of writing to memory, to prevent indirect information flows [21]. We discuss Q below. In the typing rules we project the elements of the Ω tuple via the dot notation; for example, $\Omega.\mathrm{pc}$ is the pc component of Ω. We write $\Omega[\mathrm{pc} = \mathrm{pc}']$ to represent the context that is identical to Ω except the pc component is replaced with the value pc', and likewise for other components of a context.

As in other security-typed languages, type checking in $\textsc{Rx}_{\textsf{core}}$ is equivalent to security checking: if program S type checks, when executed it will not leak information in violation of its policy. The policy context Q is a compile-time approximation of the actual policy Π at run time with which S will be executed. In $\textsc{Rx}_{\textsf{core}}$ and most security-typed languages, Q and Π are synonymous. That is, in these languages, it is assumed that the policy to be applied to the *entire execution* of S is known when S is compiled. We distinguish between policy context Q and policy Π now in anticipation of the full $\textsc{Rx}$ in Section 3, for which policies Π will evolve over time. Other than this difference, the typing rules in Figure 3 are standard [24].

To illustrate how the typing judgments of $\textsc{Rx}_0$ prevent illegal information flows, consider the rule for the if-statement. The branches of the statement S_1 and S_2 carry information about the value of the expression E. Therefore, the rule checks each branch in a context where the effect lower-bound pc is strengthened to be no less than the security level of E. This has an impact on how assignment statements are checked. Notice the last premise of the rule for the assignment statement. This premise states that the label of the location $\mathtt{x}$ being written to must not be less than the effect lower-bound. This ensures that if the assignment were to occur within a branch of an if-statement, then information about the expression that appears in the guard of the if-statement is not leaked to a low-security variable.

3 Rx: Adding Policy Updates to $\textsc{Rx}_0$

This section presents the remaining features of the full language $\textsc{Rx}$, which include (1) *policy queries* by which programs can examine the current policy during execution, and (2) *policy updates*, by which programs can add or delete statements from the current policy. The type system ensures none of these operations will leak confidential information, as proven in the next section. In addition, because policy updates are a potentially dangerous operation—adding a new delegation statement effectively declassifies information [11]—$\textsc{Rx}$ adapts the integrity constraints from previous work on *robust declassification* [31, 17]. Intuitively, the owner of a role ρ must trust the integrity of the decision to update policy statements that define ρ. Interestingly, changes to policy become a potential conduit for illegal information flow. As such, we use *metapolicies* [12] for protecting the confidentiality and integrity of roles.

3.1 Rx Syntax

The syntax of $\textsc{Rx}$ is shown in Figure 4. It differs from $\textsc{Rx}_{\textsf{core}}$ in several ways. Atomic labels, L, now include abstract operators $C(\rho)$ and $I(\rho)$ to represent metapolicies that define the confidentiality and integrity of roles. Like roles themselves, metapolicies are interpreted as sets of principals. Full labels, ℓ, are now joins of pairs consisting of a confidentiality component and an integrity component, which restricts where policy updates may occur.

Policy queries, q, are used in the statement $\mathtt{if}\ (q)\ S_1\ S_2$ to branch to S_1 or S_2 depending on whether the query $L_1 \sqsubseteq L_2$ holds according to the current dynamic policy Π. Policy contexts Q used for type checking the program now consist of a set of queries $\{q_1, \ldots, q_n\}$ that represent the knowledge gained about the run time policy through policy queries.

The statement $\mathtt{update}\ \Delta$ is used to change the current

$$\begin{array}{llcl}
\text{atomic labels} & L & ::= & \rho \mid C(\rho) \mid I(\rho) \\
\text{compound labels} & \ell & ::= & (L_C, L_I) \mid \ell \sqcup \ell \\
\text{queries} & q & ::= & L_1 \sqsubseteq L_2 \\
\text{policy context} & Q & ::= & \{q_1, \ldots, q_n\} \\
\text{update} & \delta & ::= & \texttt{add}\, s \mid \texttt{del}\, s \\
\text{updates} & \Delta & ::= & \delta \mid \delta, \Delta \\
\text{statements} & S & ::= & \ldots \\
 & & \mid & \texttt{if}\ (q)\ S_1\ S_2 \\
 & & \mid & \texttt{update}\ \Delta \\
 & & \mid & \texttt{trans}_Q\ S
\end{array}$$

Figure 4. Rx **syntax, extending** Rx_{core}**.**

policy by adding or removing a collection of policy statements $\Delta = \delta_1, \ldots, \delta_n$.

Finally, the statement $\texttt{trans}_Q\ S$ creates a transaction with policy context invariant Q. Such a transaction ensures that, although the policy Π may be updated within S, modifications to memory by the statement S are consistent with respect to a single policy.

We present the intuitive idea behind these new constructs by example, followed by the formal dynamic and static semantics, and conclude with a discussion of metapolicies.

3.2 Motivating Examples

Example 1. *A fragment of a program that might be used to create the sample health care policy in Figure 2:*

```
if(patAcceptsTreatment)
  if(Clinic.insuranceCos ⊑ Pat.insurers)
    update(add(Pat.doctors ← Clinic.staff))        □
```

In the example, the variable `patAcceptsTreatment` indicates that *Pat* has agreed to be treated at the *Clinic*. As a result, the program will update *Pat*'s policy to include the *Clinic*'s staff in her authorized list of doctors, but only after ensuring that the *Clinic* accepts payment from her insurance provider.[2]

This example illustrates the purpose of the policy context Q. The policy update statement executes only if the runtime policy satisfies the label ordering relation that appears in the second if-statement. This indicates that it is safe to assume this label ordering when type-checking the update statement since it will always be true when the statement executes. The policy context Q is used to accumulate the result of label ordering queries that appear in enclosing scopes and is used to statically prove label orderings in the absence of the runtime policy Π.

[2]This example is a bit artificial: in practice, one would also need to check that *Pat*.insurers is not empty (i.e. she has *some* insurance); such a check could easily be added. Also, this check fails if *Pat*.insurers contains some principal not in *Clinic*.insuranceCos. Handling the condition correctly would require intersection roles that we have omitted for simplicity in this paper.

While straightforward, this program has a number of potential information leaks. Suppose that `patAcceptsTreatment` is private to only *Pat* and staff at the *Clinic*, but that the contents of *Pat*.doctors is public. Then an adversary could learn the secret value of `patAcceptsTreatment` by observing *Pat*.doctors. This occurs because policy is essentially another kind of data, which suggests we must protect it in the same way as we protect variables. There is a similar dependency between between the contents of *Clinic*.insuranceCos and *Pat*.insurers and the contents of *Pat*.doctors. The change to the latter may indirectly reveal information to an adversary about the former (i.e., that the members of *Pat*.insurers are included in *Clinic*.insuranceCos). To address both cases, we define *metapolicy label* of role ρ to be $lab(\rho)$, and use this label to protect policy information.

Protecting policy information involves both confidentiality and integrity concerns. In particular, the dependency between the variable `patAcceptsTreatment` and the update to role *Pat*.doctors implies that the contents of `patAcceptsTreatment` should be *trusted* by *Pat*; otherwise, a malicious adversary could modify this variable and affect an unauthorized change to *Pat*'s policy. Therefore, Rx labels have the form (L_C, L_I), where L_C describes the confidentiality level and L_I describes the integrity level. As a result, we must define both confidentiality and integrity of roles as well, with $lab(\rho) = (C(\rho), I(\rho))$. Here the metapolicies $C(\rho)$ and $I(\rho)$ may depend on the owner of the role ρ and delegation information in the current policy. Section 3.5 will discuss possible choices of metapolicy.

Example 2. *A program that* leaks information *across updates to the policy in Figure 2.* This transitive flow of information is illegal, motivating Rx's transactional semantics.

Assume `clinicRec` is labeled *Clinic*.staff, and `patSymptoms` labeled *Pat*.healthRecords, and `philRec` labeled *DrPhil*.self.

```
S1:  if(Pat.healthRecords ⊑ Clinic.staff)
       clinicRec := patSymptoms;
S2:  if(leaveClinic)
       update(del(Pat.doctors ← Clinic.staff));
S3:  update(add(Clinic.staff ← {DrPhil}));
S4:  if(Clinic.staff ⊑ DrPhil.self)
       philRec := clinicRec                          □
```

Here, `patSymptoms` contains data confidential to the role *Pat*.healthRecords. Line `S1` copies this data into the *Clinic* records, which is permitted by the policy in Figure 2. If the patient decides to leave the clinic, represented by the variable `leaveClinic` in line `S2`, the policy is updated to remove the *Clinic*.staff from *Pat*.doctors. Subsequently, *DrPhil* joins the clinic and is therefore added as

$$\frac{\mathcal{E}.\Psi = \cdot \quad \Psi' = (\mathcal{E}.M, \texttt{trans}_Q\, S)}{\mathcal{E}, \texttt{trans}_Q\, S \longrightarrow \mathcal{E}[\Psi = \Psi'], \texttt{trans}_Q\, S} \ \text{(E-TR1)}$$

$$\frac{\mathcal{E}.\Psi \neq \cdot \quad \mathcal{E}, S \longrightarrow \mathcal{E}', S'}{\mathcal{E}, \texttt{trans}_Q\, S \longrightarrow \mathcal{E}', \texttt{trans}_Q\, S'} \ \text{(E-TR2)}$$

$$\frac{\mathcal{E}.\Psi \neq \cdot}{\mathcal{E}, \texttt{trans}_Q\, \texttt{skip} \longrightarrow \mathcal{E}[\Psi = .], \texttt{skip}} \ \text{(E-TR3)}$$

$$\frac{\mathcal{E}.\Psi \neq \cdot \quad \mathcal{E}, S \rightsquigarrow \mathcal{E}', S'}{\mathcal{E}, \texttt{trans}_Q\, S \longrightarrow \mathcal{E}', S'} \ \text{(E-TR4)}$$

$$\frac{\begin{array}{c}\Pi' = \mathcal{E}.\Pi \cup \{s \mid \texttt{add}\ s \in \Delta\} \setminus \{s \mid \texttt{del}\ s \in \Delta\} \\ \mathcal{E}.\Psi = (M', \texttt{trans}_Q\, S) \quad \forall q \in Q.(\Pi \vdash q) \Leftrightarrow (\Pi' \vdash q)\end{array}}{\mathcal{E}, \texttt{update}\ \Delta \longrightarrow \mathcal{E}[\Pi = \Pi'], \texttt{skip}} \ \text{(E-UP)}$$

$$\frac{\begin{array}{c}\Pi' = \mathcal{E}.\Pi \cup \{s \mid \texttt{add}\ s \in \Delta\} \setminus \{s \mid \texttt{del}\ s \in \Delta\} \\ \mathcal{E}.\Psi = (M', \texttt{trans}_Q\, S) \quad \exists q \in Q.(\Pi \vdash q) \not\Leftrightarrow (\Pi' \vdash q)\end{array}}{\mathcal{E}, \texttt{update}\ \Delta \rightsquigarrow \mathcal{E}[M = M'][\Pi = \Pi'], \texttt{trans}_Q\, S} \ \text{(R-UP)}$$

$$\frac{\mathcal{E}.\Pi \vdash q \Rightarrow j = 1 \quad \mathcal{E}.\Pi \not\vdash q \Rightarrow j = 2}{\mathcal{E}, \texttt{if}\ (q)\ S_1\ S_2 \longrightarrow \mathcal{E}, S_j} \ \text{(E-IFQ)}$$

$$\frac{\mathcal{E}, S_1 \rightsquigarrow \mathcal{E}', S}{\mathcal{E}, S_1; S_2 \rightsquigarrow \mathcal{E}', S} \ \text{(R-SEQ)}$$

Figure 5. Rx **execution (**$\mathcal{E}, S \longrightarrow \mathcal{E}', S'$**) and rollback (**$\mathcal{E}, S \rightsquigarrow \mathcal{E}', S'$**).**

part of *Clinic*.staff. If this policy update succeeds, then the program can copy data from the `clinicRec` variable into `philRec`, which can be labeled by role *DrPhil*.self. Consequently, *DrPhil* is able to view the `patSymptoms` even though this information flow is permitted by neither the original nor the new policy. This is an example of a unintended *transitive flow*. Other unintentional flows due to the inconsistent update of policy are also possible.

In the example, the unintended flow is caused because the label ordering relation (*Pat*.healthRecords $\sqsubseteq$ *Clinic*.staff) needed to justify the flow of information in the assignment of `S1` was violated by the update to policy. This problem of unintentional flows motivates the support of a transactional model [20, 30] to our language Rx. The semantics of a transaction $\texttt{trans}_Q\, S$ is such that if, during the execution of S, a policy update violates a label ordering relation necessary to show the absence of unintentional flows, then the *memory* of the program is reverted to the state it was prior to the start of the transaction. Execution of the statement S then resumes using the updated policy. The subscript Q represents the policy invariant that must be preserved across the policy updates.

Rolling back transactions, however, introduces yet another channel of information leaks. To see why, suppose that we enclose the program of Example 2 within a transaction. Since the policy update statement is `S2` violates the policy invariant that appears in `S1`, the transaction is rolled back, undoing the assignment to location `clinicRec`. Any principal P who can view the contents of `clinicRec` can therefore observe whether or not the transaction has been rolled back. If the confidentiality of `leaveClinic` is greater than `clinicRec`, then, by observing the rollback, the principal P will have gained information about `leaveClinic`. The static semantics of Rx—explained in detail below—check transactions to guarantee that no information leaks of this kind occur.

Note that such information leaks are not peculiar to our choice of transaction rollback as the enforcement mechanism for policy consistency. An alternative roll-forward semantics, in which policy updates that violated consistency were delayed until the transaction completed execution, would have similar issues [11].

For simplicity, in this paper we consider only policy updates that contain statically-known, literal policy statements (the s in `add` s or `del` s). However, the transaction semantics just described also naturally supports first-class, dynamically determined policy updates and queries, following in the style of run-time principals [26] or dynamic labels [33].

3.3 Rx Dynamic Semantics

The dynamic semantics of Rx is defined by the execution relation $\mathcal{E}, S \longrightarrow \mathcal{E}', S'$ where $\mathcal{E}$ is the current execution configuration and S is the current program statement. The execution takes a small step, resulting in a new configuration $\mathcal{E}'$ and a new statement S' to be executed next. The syntax for configurations and dynamic snapshots are:

$$\begin{array}{lcll}\text{eval configuration} & \mathcal{E} & ::= & (\Pi, M, \Psi) \\ \text{dynamic snapshot} & \Psi & ::= & \cdot \mid (M, S)\end{array}$$

An execution configuration consists of a policy Π; a memory store M mapping variables to values; and a possibly empty *dynamic snapshot* Ψ of memory M and program statement S used to implement transactional rollback.

The rules, shown in Figure 5, define two relations: $\mathcal{E}, S \longrightarrow \mathcal{E}', S'$ for normal execution, and $\mathcal{E}, S \rightsquigarrow \mathcal{E}', S'$ for rollback. The rules for standard constructs (assignment, addition, sequences etc.) are not shown.

The rules (E-TR1), (E-TR2), (E-TR3) and (E-TR4) are for the execution of transaction statement $\texttt{trans}_Q\, S$. The checks $\mathcal{E}.\Psi = \cdot$ and $\mathcal{E}.\Psi \neq \cdot$ determine if a snapshot Ψ is available. (E-TR1) takes a new snapshot Ψ' of the current memory store M and the current statement $\texttt{trans}_Q\, S$ in the

$$\frac{\mathrm{pc}' = \mathrm{pc} \sqcup \mathit{lab}(q) \quad q \in \Phi.Q \quad \Gamma; \mathrm{pc}'; Q \cup \{q\}; \Phi \vdash S_1 \quad \Gamma; \mathrm{pc}'; Q; \Phi \vdash S_2}{\Gamma; \mathrm{pc}; Q; \Phi \vdash \texttt{if}\ (q)\ S_1\ S_2} \text{ (T-IFQ)}$$

$$\frac{\Gamma; \mathrm{pc}; \emptyset; (\mathrm{pc}, Q') \vdash S}{\Gamma; \mathrm{pc}; \emptyset; \cdot \vdash \texttt{trans}_{Q'}\ S} \text{ (T-TR)} \qquad \frac{Q \vdash \mathrm{pc} \sqsubseteq \mathit{lab}(\Delta) \quad Q \vdash \mathrm{pc} \sqsubseteq \mathrm{pc}' \quad Q \vdash \mathit{lab}(Q') \sqsubseteq \mathrm{pc}'}{\Gamma; \mathrm{pc}; Q; (\mathrm{pc}', Q') \vdash \texttt{update}\ \Delta} \text{ (T-UP)}$$

$$Q \vdash \ell \sqsubseteq \ell \qquad \frac{Q \vdash \ell_1 \sqsubseteq \ell \quad Q \vdash \ell \sqsubseteq \ell_2}{Q \vdash \ell_1 \sqsubseteq \ell_2} \qquad \frac{L_1 \sqsubseteq L_2 \in Q}{Q \vdash L_1 \sqsubseteq L_2}$$

$$\frac{Q \vdash L_{C_1} \sqsubseteq L_{C_2} \quad Q \vdash L_{I_1} \sqsubseteq L_{I_2}}{Q \vdash (L_{C_1}, L_{I_1}) \sqsubseteq (L_{C_2}, L_{I_2})} \qquad \frac{Q \vdash (L_C, L_I) \sqsubseteq \ell \quad Q \vdash \ell' \sqsubseteq \ell}{Q \vdash (L_C, L_I) \sqcup \ell' \sqsubseteq \ell} \qquad \frac{Q \vdash \ell \sqsubseteq (L_C, L_I) \quad Q \vdash \ell \sqsubseteq \ell'}{Q \vdash \ell \sqsubseteq (L_C, L_I) \sqcup \ell'}$$

Figure 6. Rx **typing (**$\Gamma; \mathrm{pc}; Q; \Phi \vdash S$**) and label ordering (**$Q \vdash \ell_1 \sqsubseteq \ell_2$**).**

execution context $\mathcal{E}$, only if the current snapshot is empty.[3] (E-TR2) is a congruence rule for evaluation within a transaction and (E-TR3) effectively commits. (E-TR4) uses the rollback relation $\mathcal{E}, S \rightsquigarrow \mathcal{E}', S'$ triggered by failed updates to abort a transaction.

(E-UP) takes the current policy $\mathcal{E}.\Pi$ and computes the new policy Π' by adding or deleting policy statements according to Δ in the statement update Δ. However, the new policy Π' must be consistent with the query set Q which annotates the enclosing transaction statement $\texttt{trans}_Q\ S$, also stored in the snapshot Ψ.

Formally, the *policy consistency condition* is:

$$\forall q \in Q. \quad (\Pi \vdash q) \Leftrightarrow (\Pi' \vdash q)$$

This consistency condition says that the satisfiability of every query q in the query set Q is the same for the old policy and for the new policy. This condition is sufficient to guarantee that every information flow witnessed during the execution of the transaction under the old policy is also consistent with the new policy. If the consistency condition fails, (R-UP) is triggered instead, rolling back using (R-SEQ) to discard the second statement of any sequence statement $S_1; S_2$, and completing the abort using (E-TR4).

If the consistency condition fails, (E-TR4) together with (R-UP) is triggered instead, rolling back using (R-SEQ) by discarding the second statement of the sequence statement $S_1; S_2$. Note that policy updates that occur in a transaction are treated like I/O operations in traditional transaction systems [20, 10]—only writes to memory are undone by a rollback, policy updates are left intact. This means that pending updates in a transaction may not have been performed before the rollback operation. As with traditional transaction systems, we could use compensations [29] to allow programmers to undo some updates if necessary. Another consequence of not rolling back policy updates is that it is possible for badly written programs to enter livelock—for instance, a transaction that performs mutually incompatible policy updates can cause the rollback mechanism to enter an infinite loop.

Finally, (E-IFQ) for the policy query statement chooses the appropriate branch to take according to the judgment $\mathcal{E}.\Pi \vdash q$; that is, whether or not the query q holds in the current policy Π. This judgment is defined as follows (note the contravariance):

$$\Pi \vdash L_1 \sqsubseteq L_2 \iff [\![L_2]\!]_\Pi \subseteq [\![L_1]\!]_\Pi$$

Based on the definition of L, the interpretation of $[\![L]\!]_\Pi$ depends on the semantics of roles $[\![\rho]\!]_\Pi$ (Section 2.3), and the semantics of a role's confidentiality and integrity labels, $C(\rho)$ and $I(\rho)$, respectively, which are also treated as sets of principals.

Example 3. A program that, executes under the policy $\{A.r \leftarrow B.r, B.r \leftarrow \{B\}\}$, rolls back:

```
trans{A.r⊑B.r}
  if(A.r ⊑ B.r) {
   update(del(A.r ← B.r)); S }
```
□

Execution of this program begins with the (E-TR1) rule which takes a snapshot of the memory and program and records it in Ψ. Notice that the subscript $\{A.r \sqsubseteq B.r\}$ on the transaction statement is a set that includes the lone policy query that occurs in the body of the transaction. (E-TR2) now applies and with the program taking a small step using (E-IFQ). Since the role $A.r$ delegates to $B.r$, the policy entails the query ($\mathcal{E}.\Pi \vdash q \Rightarrow j = 1$), the then-branch of the statement is taken. We now have a sequence of the statements with the first being an update statement update(del(A.r $\leftarrow$ B.r)), all enclosed in a transaction statement from the first line.

In attempting to apply the (E-TR2) rule again, the first statement in sequence must take a step under the normal execution relation $\longrightarrow$ (according to the standard rule for evaluating sequences, which is omitted here). In this case

[3] We support only non-nested transactions for simplicity, as discussed in Section 3.4, so no stack of snapshots is needed.

the policy consistency condition is violated by the update since, under the new policy ($\{B.r \leftarrow \{B\}\}$), the policy query ($A.r \sqsubseteq B.r$) is not satisfied, unlike under the old policy. Therefore, the first statement of the sequence can only take a step under the rollback relation $\rightsquigarrow$. Then, we use (E-TR4) with (R-SEQ) preceded by (R-UP) in the premise. The conclusion of (R-SEQ) serves to discard the statement S that succeeds the update statement. The result is that the program and memory is reverted to its original state and the policy is now $\{B.r \leftarrow \{B\}\}$.

3.4 Rx Static Semantics

The static semantics of Rx is defined by the typing relation $\Omega \vdash S$ in Figure 6, just like the typing relation for Rx_0 in Figure 3. However, the typing context Ω now contains a *static snapshot* for type checking transactions:

$$\begin{array}{llll} \text{typing context} & \Omega & ::= & (\Gamma; \mathrm{pc}; Q; \Phi) \\ \text{policy context} & Q & ::= & \{L_1 \sqsubseteq L_{1'}, \ldots, L_n \sqsubseteq L_{n'}\} \\ \text{static snapshot} & \Phi & ::= & \cdot \mid (\mathrm{pc}, Q) \end{array}$$

Hence, we also write the typing judgment as $\Gamma; \mathrm{pc}; Q; \Phi \vdash S$. The type binding for variables Γ and the program counter pc are standard, and the policy context is already defined Figure 4 (but expanded here for easy reference). The snapshot Φ is used to approximate the assumptions of a transaction (to be explained below).

Label ordering and metapolicy labels Figure 6 (the third and the fourth rows) also specifies the label ordering relation $Q \vdash \ell_1 \sqsubseteq \ell_2$. In the third row of Figure 6, the three rules of label ordering are straightforward: the left and the middle rules say that the relation is reflexive and transitive, and the right rule makes use of the policy context Q when the labels L_1 and L_2 are atomic. In the fourth row of Figure 6, the left rule handles the compound label (L_C, L_I), and the middle and the right rules handle the join label $\ell \sqcup \ell'$.

We also need the following auxiliary function $lab(\cdot)$ to compute the metapolicy label of policy updates Δ and policy queries Q:

$$\begin{array}{rcl} lab(\Delta) & = & \bigsqcup_{\texttt{add } s, \texttt{del } s \in \Delta} lab(roledef(s)) \\ lab(Q) & = & \bigsqcup_{q \in Q} lab(q) \\ lab(L_1 \sqsubseteq L_2) & = & lab(L_1) \sqcup lab(L_2) \\ lab(C(\rho)) & = & lab(\rho) \\ lab(I(\rho)) & = & lab(\rho) \\ lab(\rho) & = & (C(\rho), I(\rho)) \end{array}$$

The function $lab(\cdot)$ uses the metapolicy $C(\cdot)$ and $I(\cdot)$ to construct a label for a role. A metapolicy label for queries $L_1 \sqsubseteq L_2$ is the join of all the metapolicy labels for roles contained in L_1 and L_2.

Typing policy queries The rule (T-IFQ) type checks policy query statement $\texttt{if}\ (q)\ S_1\ S_2$. The rule has three important aspects. First, notice that we check the true-branch S_1 using policy context $Q \cup \{q\}$, that is, the union of the current policy context and the query being asked. Second, both branches are checked using a restricted program counter label pc', which is defined as the join of the current pc label and the label of the query q according to the label set function $lab(q)$. This reflects the information gained by querying the policy, and is used to prevent leaks about a policy through assignments to variables. Finally, the premise $q \in \Phi.Q$ is used to ensure transaction consistency, which we will explain when we consider the typing rule for transactions below.

Example 4. *An instantiation of the typing rule (T-IFQ) for policy queries for Example 1.* (Here we abbreviate *Clinic*.insuranceCos and *Pat*.insurers to save space.)

$$\frac{\begin{array}{c} \mathrm{pc}' = \mathrm{pc} \sqcup lab(\mathit{Clinic}.\mathrm{ins} \sqsubseteq \mathit{Pat}.\mathrm{ins}) \\ \mathit{Clinic}.\mathrm{ins} \sqsubseteq \mathit{Pat}.\mathrm{ins} \in \Phi.Q \\ \Gamma; \mathrm{pc}'; Q \cup \{\mathit{Clinic}.\mathrm{ins} \sqsubseteq \mathit{Pat}.\mathrm{ins}\}; \Phi \vdash \texttt{update} \ldots \end{array}}{\Gamma; \mathrm{pc}; Q; \Phi \vdash \texttt{if}\ (\mathit{Clinic}.\mathrm{ins} \sqsubseteq \mathit{Pat}.\mathrm{ins})\ \texttt{update} \ldots}$$

where $lab(\mathit{Clinic}.\mathrm{ins} \sqsubseteq \mathit{Pat}.\mathrm{ins})$ is

$$\begin{array}{ll} & (C(\mathit{Clinic}.\mathrm{ins}), I(\mathit{Clinic}.\mathrm{ins})) \\ \sqcup & (C(\mathit{Pat}.\mathrm{ins}), I(\mathit{Pat}.\mathrm{ins})) \end{array} \qquad \square$$

Typing transactions The snapshot Φ is used to ensure that every policy query q that appears in the body S of a transaction $\texttt{trans}_{Q'}\ S$ also appears in Q'. This is ensured by the (T-TR) rule, whose body S is checked in a Φ snapshot that mentions Q', and the (T-IFQ) rule, whose premise $q \in \Phi.Q$ ensures that every policy query is accounted for. The (T-TR) rule also includes the current program counter label pc in Φ. Doing this guarantees that the memory effects that occur when a transaction is rolled back do not leak information. We explain how this works when considering the (T-UP) rule below.

Notice that (T-TR) must occur in an empty policy context. We did this to prevent nested transactions, which simplifies the dynamic semantics and typing rules. Supporting nested transactions would require a flow-sensitive static analysis if we were to support partial rollback (in which inner transactions roll back but outer ones do not). Such an approach would also increase the precision of the static semantics and permit more updates. Ultimately, we want to extend Rx with procedures, which will increase the need for nested transactions; i.e., to allow transaction-containing procedures to compose.

Also notice that these rules effectively prevent policy queries from occurring outside a transaction. This is to prevent aberrant behavior in which an update occurring within a transaction has a conflict with non-transactional query outside the transaction; in this case, rolling back would not solve the problem, and the program would resume execution under the new policy while still not satisfying the non-transactional query.

Typing policy updates The (T-UP) rule defines the conditions under which policy may be safely modified. Recall that the metapolicy label of a role ρ is $(C(\rho), I(\rho))$, where the metapolicy $C(\rho)$ is the set of principals who are permitted to view the members of ρ, and the metapolicy $I(\rho)$ is the set of principals that trust ρ's definition. As motivated by the discussion of Example 1, we must be careful to only allow a program to update the definition of a role ρ when doing so is trusted by those in $I(\rho)$; this is a condition similar to robust declassification [31].

Moreover, the change in a role definition ρ should only reveal information to principals in $C(\rho)$. This condition is checked by the first premise of (T-UP) in a manner analogous to the rule for the assignment statement in Figure 3. In particular, the pc must be lower than both the integrity and confidentiality levels of the role, thus ensuring that it is not improperly updated, and that its update does not leak information. Note that to ensure that only the owner of a role is permitted to modify its definition, any metapolicy $I(\rho)$ must include the owner of the role.

Example 5. *An instantiation of the typing rule (T-UP) for policy updates in Figure 6.* Suppose we enclose Example 1 as statement S in a transaction $\texttt{trans}_{Q'}\ S$. Hence we wish to prove $\Gamma; \mathrm{pc}; Q; \cdot \vdash \texttt{trans}_{Q'}\ S$ with

$$\begin{array}{rcl}\Gamma & \equiv & \texttt{patAcceptsTreatment} : (C(\mathit{Pat}.\mathrm{drs}), I(\mathit{Pat}.\mathrm{drs})) \\ \mathrm{pc} & \equiv & (C(\mathit{Pat}.\mathrm{drs}), I(\mathit{Pat}.\mathrm{drs}))\end{array}$$

The label $(C(\mathit{Pat}.\mathrm{drs}),\ I(\mathit{Pat}.\mathrm{drs}))$ for the variable `patAcceptsTreatment` determines whether Pat's role $\mathit{Pat}.\mathrm{drs}$ should be updated. The pc here will be added to the snapshot Φ by (T-TR). Later, within (T-IFQ) derivation shown earlier, we type check the `update` and apply the (T-UP) rule as follows:

$$\frac{\begin{array}{c} Q \cup \{\mathit{Clinic}.\mathrm{ins} \sqsubseteq \mathit{Pat}.\mathrm{ins}\} \vdash \mathrm{pc}' \sqsubseteq \mathit{lab}(\mathit{Pat}.\mathrm{drs}) \\ Q \cup \{\mathit{Clinic}.\mathrm{ins} \sqsubseteq \mathit{Pat}.\mathrm{ins}\} \vdash \mathrm{pc}' \sqsubseteq \mathrm{pc} \\ Q \cup \{\mathit{Clinic}.\mathrm{ins} \sqsubseteq \mathit{Pat}.\mathrm{ins}\} \vdash \mathit{lab}(Q') \sqsubseteq \mathrm{pc}\end{array}}{\begin{array}{c}\Gamma; \mathrm{pc}'; Q \cup \{\mathit{Clinic}.\mathrm{ins} \sqsubseteq \mathit{Pat}.\mathrm{ins}\}; (\mathrm{pc}, Q') \vdash \\ \texttt{add}(\mathit{Pat}.\mathrm{drs} \leftarrow \mathit{Clinic}.\mathrm{staff})\end{array}}$$

where

$$\begin{array}{rcl}\mathrm{pc}' & \equiv & \mathrm{pc} \sqcup (C(\mathit{Clinic}.\mathrm{ins}), I(\mathit{Clinic}.\mathrm{ins})) \\ & & \sqcup\, (C(\mathit{Pat}.\mathrm{ins}), I(\mathit{Pat}.\mathrm{ins})) \\ \mathit{lab}(\mathit{Pat}.\mathrm{drs}) & \equiv & (C(\mathit{Pat}.\mathrm{drs}), I(\mathit{Pat}.\mathrm{drs}))\end{array}$$

If Q were $\emptyset$, it would not be sufficient to prove the first premise according to the label ordering rules in Figure 6. This is because $\{\mathit{Clinic}.\mathrm{ins} \sqsubseteq \mathit{Pat}.\mathrm{ins}\}$ alone has nothing to say about the relationship between the metapolicies of the various roles. It would be sufficient to choose

$$\begin{array}{rlcl} Q \equiv \{ & C(\mathit{Clinic}.\mathrm{ins}) & \sqsubseteq & C(\mathit{Pat}.\mathrm{drs}), \\ & C(\mathit{Pat}.\mathrm{ins}) & \sqsubseteq & C(\mathit{Pat}.\mathrm{drs}), \\ & I(\mathit{Clinic}.\mathrm{ins}) & \sqsubseteq & I(\mathit{Pat}.\mathrm{drs}), \\ & I(\mathit{Pat}.\mathrm{ins}) & \sqsubseteq & I(\mathit{Pat}.\mathrm{drs})\ \} \end{array}$$

Such a context Q could be established by preceding the code S in Example 1 with policy queries testing these assertions within the transaction. Rather than expect the programmer to write these, they could be straightforwardly inferred. To type these queries (and the one already in S) would require we choose $Q' \supseteq Q \cup \{\mathit{Clinic}.\mathrm{ins} \sqsubseteq \mathit{Pat}.\mathrm{ins}\}$. □

The decision of whether or not an update causes a rollback *depends* on the policy consistency condition ($\forall q \in Q.\ \Pi \vdash q = \Pi' \vdash q$) appearing in the operational rules (E-UP) and (R-UP) in Figure 5. We want to avoid leaking information about the queries through low-security data and low-security policy. The first case is handled by the second premise of the (T-UP) rule. It ensures that all memory effects in a transaction are bounded from below by pc label of the current context. As explained earlier (Section 3.1) for Example 2, this guarantees that the change to memory caused by the rollback of a transaction is observable only by principals who are also permitted to view the effects of the context in which the update occurs. In our example typing above, Q clearly satisfies this condition because it asserts that each component of the pc label is higher than each of the components in pc' that do not already include pc.

The second case of a leak via policy is handled by the third premise ($Q \vdash \mathit{lab}(Q') \sqsubseteq \mathrm{pc}'$) of (T-UP), which requires that all the invariants mentioned in Q' are at a lower security level than the program counter label at the start of the transaction. This ensures that the effects to memory that occur as a result of rollback are at a higher security level than all the policy queries. Therefore, the principals that can observe the effects to memory as a result of rollback are also sufficiently privileged to view the definitions of roles mentioned in Q'. Thus policy information is not leaked into memory as a result of rollback. In our example typing, this third premise is clearly satisfied because $\mathit{lab}(Q') = (C(\mathit{DrBob}.\mathrm{ins}), I(\mathit{DrBob}.\mathrm{ins})) \sqcup (C(\mathit{Pat}.\mathrm{ins}), I(\mathit{Pat}.\mathrm{ins}))$.

3.5 Metapolicy for Role Labeling

RX uses metapolicies $C(\rho)$ and $I(\rho)$ to protect the confidentiality and integrity, respectively, of a role ρ. Because metapolicies are labels, they must be interpreted as sets of principals; i.e. $[\![C(\rho)]\!]_\Pi = \{P_1, \ldots, P_n\}$ for some principals P_i, and similarly for $I(\rho)$. Here we discuss possible interpretations of $C(\rho)$ and $I(\rho)$ in turn, and ultimately define a sufficient condition for metapolicy interpretations that enables proving noninterference. The particular features of RT_0 policies for defining sets have a heavy influence on what determines a reasonable interpretation.

A simple interpretation for role confidentiality is $[\![C(\rho)]\!]_\Pi = \bot$. Here, $\bot$ denotes the set of all principals, so that under this metapolicy every principal can know the contents of all roles. While simple, this metapolicy requires policy update decisions to be independent of secret data, as shown in Example 1.

To define $C(\rho)$ to be more restrictive, RT_0 semantics requires at the least that $[\![C(\rho)]\!]_\Pi \supseteq \{owner(\rho)\}$. That is, we must permit the owner to view the definition of the role. In addition, a principal P should be able to learn of its membership in a role ρ. Were this not so, there would be no observable effect if P was removed from the role, and so its initial membership would be fairly useless. Thus an intuitive choice would to have $[\![C(\rho)]\!]_\Pi = [\![\rho]\!]_\Pi \cup \{owner(\rho)\}$. That is, members of ρ are permitted to view its definition, which would include themselves and the other members of the role. A stronger *anonymity* policy might allow a principal to learn of its own membership but not that of others [9]. This is not expressible in terms of $C(\rho)$, but even if it were, it is not clear how it could be enforced. In particular, if some variable x were labeled with such a ρ, and a program acting on the authority of principal P were to modify this variable, other principals in ρ could observe the effect of this change and conclude P was a member of ρ.

Though intuitive, allowing $C(\rho)$ to include all the members of ρ is not enough; it could still reveal information. To see why, consider the example from Figure 2. In the example, the definition of the role Pat.doctors given by a membership statement including $DrSue$ and a delegation to $Clinic$.staff; the interpretation of the role is given by $[\![Pat.doctors]\!]_\Pi = \{DrAlice, DrBob, DrSue\}$. Under this choice of metapolicy, we permit $DrSue$ to view the definition of Pat.doctors. By doing so, even though $DrSue$ is not a member of $Clinic$.staff, she can ascertain the membership of $Clinic$.staff, since she knows that $DrAlice$ and $DrBob$ are in Pat.doctors. Realizing that the definition of Pat.doctors *depends* on the definition of $Clinic$.staff makes it clear that it is not reasonable to treat the policy statements defining $Clinic$.staff as being more confidential than the those defining Pat.doctors.

To remedy this problem, we can require the interpretation of $C(\rho)$ to satisfy the following condition. Given a relation $del_\Pi(\rho)$ that specifies the set of roles ρ_i to which ρ delegates (either directly or transitively), we must have both the condition $[\![\rho]\!]_\Pi \subseteq [\![C(\rho)]\!]_\Pi$ and:

$$del_\Pi(\rho) = \{\rho_1, \ldots, \rho_n\} \Rightarrow \forall i.\ [\![C(\rho)]\!]_\Pi \subseteq [\![C(\rho_i)]\!]_\Pi$$

Note that an interpretation that satisfies these conditions must also be robust under policy updates. A simple way to ensure this is to allow the semantics of role confidentiality to change with the update. While simple, this would permit members of one role to view another role by delegating to it. To prevent this we could require that for an update to add a delegation statement $A.r \leftarrow B.r$ the integrity of the pc must be trusted by both $I(A.r)$ and $I(B.r)$. We leave exploration of this issue to future work.

Similar choices are possible for the definition of the role integrity metapolicy $I(\rho)$. The simplest choice is to have $[\![I(\rho)]\!]_\Pi = \bot$, which means that the policy is trusted by all principals, and thus policy updates can occur only in contexts trusted by all principals. In general, the condition on the interpretation for integrity mirrors the condition on confidentiality, replacing $C(\rho)$ with $I(\rho)$ in the equation above. To see how this makes sense, consider the policy in Figure 2 once again. The delegation Pat.doctors $\leftarrow$ $Clinic$.staff indicates that Pat trusts $Clinic$ to partially manage the definition of Pat.doctors through the definition of $Clinic$.staff. As such, a reasonable choice for metapolicy $I(\rho)$ would be that ρ is trusted by $owner(\rho)$ and the owners of all roles that transitively delegate to ρ.

A further condition on metapolicies $C(\rho)$ and $I(\rho)$ is induced by our definition of $lab(C(\rho))$ and $lab(I(\rho))$. This is necessary since the result of a policy query of the form $C(\rho) \sqsubseteq L$ carries information about the *confidentiality* of ρ as opposed to information about ρ itself. In Section 3.4, we defined $lab(C(\rho)) \equiv lab(\rho)$ and $lab(I(\rho)) \equiv lab(\rho)$. This means that the confidentiality of the metapolicy governing a role is the same as the confidentiality of the role itself. This choice avoids the need for higher-order metapolicies.

To appreciate the implications of this choice we can view $C(\cdot)$ and $I(\cdot)$ as functions that map roles to sets of principals. The result of this function might depend on its input ρ, and possibly on the definition of some other roles $\{\rho_1, \ldots, \rho_n\}$ that appear in the policy. In such a case, since $C(\rho)$ carries information about ρ and $\rho_1, \ldots, \rho_n$, the label of $C(\rho)$ should be as below: $(\sqcup_i C(\rho_i)) \sqcup C(\rho)$. To be able to claim that $lab(C(\rho)) \equiv lab(\rho)$ the metapolicy must satisfy the following condition:

$$\forall i. [\![C(\rho)]\!]_\Pi \subseteq [\![C(\rho_i)]\!]_\Pi$$

An identical condition must also hold true for $I(\rho)$. The specific metapolicies described previously in terms of the delegation relation $del_\Pi(\rho)$ satisfy this constraint.

4 Noninterference

This section proves a *noninterference* property for Rx. Informally speaking, we show that if an Rx program S is well-formed according to the static semantics, then the effects of executing that program visible to a low-security observer are independent of the high-security parts of the configuration elements M and Π (i.e., memory and policy) with which the program executes. Updates to policy intentionally alter the security behavior of the program, possibly revealing previously secret information [11]. Therefore, rather than providing an end-to-end security guarantee with respect to a single policy, we prove that information flows observable by a principal at a given point in time are consistent with the current policy. Since our formulation of policy and data integrity is conceptually identical to our formulation of confidentiality, this property of noninterference also yields a preservation property for the integrity of policy and data. We do not consider timing channels.

The statement of noninterference relies on the notion of a well-formed configuration. We write $\Omega \models \mathcal{E}$ to mean that the execution context is consistent with the static assumptions made while type-checking the program.

Definition 6. *A configuration $\mathcal{E} = (\Pi, M, \Psi)$ is well-formed with respect to a context Ω, denoted $\Omega \models \mathcal{E}$, if and only if all of the following are true:*

$$\begin{array}{lr}
\mathit{dom}(M) \equiv \mathit{dom}(\Omega.\Gamma) & (1)\\
\forall q \in \Omega.Q \;.\; \Pi \vdash q & (2)\\
\mathit{if}\ \Psi = (M', S')\ \mathit{then} & \\
\quad \Omega \vdash S' & (3.1)\\
\quad \mathit{dom}(M') = \mathit{dom}(\Omega.\Gamma) & (3.2)\\
\quad \forall x. M(x) \neq M'(x) \Rightarrow \Pi \vdash \Omega.\mathrm{pc} \sqsubseteq \Omega.\Gamma(x) & (3.3)
\end{array}$$

The clauses in the definition above are mostly straightforward. Clause (2) connects the static approximation Q used during type checking to the runtime policy Π. To ensure this connection is sound, we prove the following lemma.

Lemma 7 (Label Ordering Soundness)**.** *For all contexts Ω and programs S, if the derivation of $\Omega \vdash S$ contains a subderivation $\Omega' \vdash S$, then the following holds true for all policies Π:*

$$(\forall q \in \Omega'.Q.\Pi \vdash q) \Rightarrow (\forall \ell_1, \ell_2.Q \vdash \ell_1 \sqsubseteq \ell_2 \Rightarrow \Pi \vdash \ell_1 \sqsubseteq \ell_2)$$

Clause (3.3) states that all effects on memory exhibited during a transaction are bounded above the pc lower-bound used to statically check the transaction.

We prove noninterference by relating execution traces of well-formed configurations, restricted to an attacker's level of observation. An *execution* of a configuration $(\mathcal{E}_0, S_0)$ (where $\mathcal{E}_0 = (\Pi, M, \Psi)$) is written $\langle \mathcal{E}_0, S_0 \rangle$ and denotes a (possibly infinite) sequence of configurations $\mathcal{E}_0, \ldots, \mathcal{E}_n, \ldots$ and programs $S_0, \ldots, S_n, \ldots$ such that $(\mathcal{E}_i, S_i) \longrightarrow (\mathcal{E}_{i+1}, S_{i+1})$. The sequence of configurations $\mathcal{E}_0, \ldots, \mathcal{E}_n, \ldots$ is called the *trace* and is written $\mathrm{Tr}(\langle \mathcal{E}_0, S_0 \rangle)$. We write α to denote a (possibly empty) sequence of configurations, and $\mathcal{E}, \alpha$ to denote the concatenation of a single configuration and a sequence.

We define the attacker's observation level as a set of roles R. The restriction of a trace α to observation level R is written $\alpha|_R$, and is defined in Figure 7. As long as the policy

Role :
$\mathit{Obs}(R, \Pi) \equiv \{\rho \mid \exists \rho' \in R.\ \Pi \vdash C(\rho) \sqsubseteq \rho'\}$

Policy :
$\Pi|_R \equiv \Pi\|_{\mathit{Obs}(R,\Pi)}$

$\emptyset\|_R \equiv \emptyset \qquad (\{s\} \cup \Pi')\|_R \equiv \begin{cases} \{s\} \cup (\Pi'\|_R) & \mathit{roledef}(s) \in R \\ \Pi'\|_R & \text{otherwise} \end{cases}$

Memory :
$M|_{R,\Pi} \equiv \{(x, M(x)) \mid \exists \rho \in R.\ \Pi \vdash \Gamma(x) \sqsubseteq \rho\}$

Transaction snapshot :
$\cdot|_{R,\Pi} \equiv \cdot \qquad (M, S)|_{R,\Pi} = (M|_{R,\Pi}, S)$

Configuration :
$(\Pi, M, \Psi)|_R \equiv (\Pi|_R, M|_{R,\Pi}, \cdot)$

Trace :
$(\mathcal{E}_1, \mathcal{E}_2, \alpha)|_R \equiv \begin{cases} \mathcal{E}_1|_R \text{ if } \mathit{Obs}(R, \mathcal{E}_2.\Pi) \not\subseteq \mathit{Obs}(R, \mathcal{E}_1.\Pi) \\ \mathcal{E}_1|_R, (\mathcal{E}_2, \alpha)|_R \quad \text{otherwise} \end{cases}$

Figure 7. Trace observability.

remains unchanged, a restricted trace consists of a restriction to each configuration element of the trace (the "otherwise" clause of the $(\mathcal{E}_1, \mathcal{E}_2, \alpha)|_R$ definition of the figure). In doing so, we restrict the view of memory according the policy Π and the $\Omega.\Gamma$ used to type check the initial program. We restrict the policy according to the metapolicy $C(\rho)$, which must satisfy the condition described in Section 3.5. However, if a policy update results in a declassification that causes the set of observable role definitions to *increase* then the trace is truncated (the first clause of the $(\mathcal{E}_1, \mathcal{E}_2, \alpha)|_R$ definition of the figure). Note that declassifications to observers at an unrelated observation level do not cause the trace to be truncated. Similarly, a policy update that causes a reduction in the privilege of an observer at level R (a revocation) has no impact on trace truncation. This truncation is justified since declassifications due to policy update are intentional releases of information. These ideas are captured by the two clauses of $(\mathcal{E}_1, \mathcal{E}_2, \alpha)|_R$.

Note that we make no attempt to restrict the observability of a program configuration while the program executes within a transaction. This makes it reasonable to exclude the snapshot Ψ when defining the observability of a configuration. However, for our statement of non-interference, it is useful to identify configurations while taking into account the transaction context, so we define $(\Pi, M, \Psi)|_R^{\psi} \equiv (\Pi|_R, M|_{R,\Pi}, \Psi|_{R,\Pi})$.

The definition of trace observability implies that computation steps are only observable if they have an effect on an observable part of memory or policy. This entails that we identify traces only up to stuttering. We write $\alpha \doteq \beta$ if α and β are equivalent up to stuttering.

We can now state the noninterference theorem.

Theorem 8 (Noninterference)**.** *Suppose for an* RX *program S and a pair of configurations $\mathcal{E}_0$ and $\mathcal{E}_1$, there exists a context Ω such that $\Omega \vdash S$, $\Omega \models \mathcal{E}_0$ and $\Omega \models \mathcal{E}_1$. Then, for any set of roles R, whenever both $\langle \mathcal{E}_0, S\rangle$ and $\langle \mathcal{E}_1, S\rangle$ terminate, we have*

$$\mathcal{E}_0 \mid_R^\psi \equiv \mathcal{E}_1 \mid_R^\psi \Rightarrow \text{Tr}(\langle \mathcal{E}_0, S\rangle) \mid_R \doteq \text{Tr}(\langle \mathcal{E}_1, S\rangle) \mid_R$$

The proof (in our technical report [25]) uses Pottier and Simonet's proof technique [18] which extends the language to represent pairs of executions that differ only in the high-security parts of their configurations. Because we may truncate traces for which there is a declassification visible at level R, to obtain an end-to-end security guarantee we can apply noninterference piecewise to each non-declassifying sub-trace. Thus we can claim that (1) the execution is noninterfering until the policy is updated; (2) the act of updating the policy itself does not leak information; and (3) after the policy has been updated all subsequent flows are consistent with the new policy.

5 Related Work

There is a large body of work on policy specification languages, including owned policies [4] and role-based languages like Cassandra [2], RBAC [19], SPKI [7, 8], and owned policies [4]. RX policies are based on those from RT framework by Li, Mitchell and Winsborough [14], which is similar to SPKI/SDSI [13]. The RX transaction semantics is inspired by software transactional memory [22], an idea which has gained popularity as an alternative to pessimistic lock-based synchronization for implementing atomic blocks [20, 30, 10].

There has been much prior work on language-based enforcement of information-flow policies [21]. The majority of that research has assumed that the security lattice and other policy components are known at compile-time and remain fixed for the duration of the program execution.

In some information flow languages the policy remains fixed but may be discovered at run time by using dynamic queries. Banerjee and Naumann [1] permit information-flow policies to be mixed with stack-inspection style dynamic access control checks. The Jif programming language [16] supports dynamic queries of the security lattice and includes features for using both dynamic principals and dynamic labels [26, 33, 27]. The latest version of Jif also allows delegations between principals to change at run time; however, Jif has no transactions or formalisms to show the absence of information leaks across updates.

The predecessor [11] of this paper showed that unrestricted updates to the security lattice could violate soundness in languages supporting dynamic policy queries, and proposed delaying updates until soundness could be ensured, as determined by a run-time examination of the program. RX builds on this work by reasoning about fine-grained policy updates within a program (in our prior work they were out-of-band), and by using roles and metapolicies to form an administrative model (the term *metapolicy* is due to Hosmer [12]).

There has been recent interest in studying *temporal policies* which are permitted to change in predefined ways during execution. Recent work on *flow locks* by Broberg and Sands [3] can encode many recently-proposed temporal policies, including declassification policies [5], erasure policies [6], and lexically-scoped flow policies [15]. RX is designed to support unrestricted changes to policy during execution. While RX currently supports updates Δ using literal statements s, the intent of transactional rollback is to support partially-unknown changes, following techniques of dynamic labels and run-time principals. When policies updates cause declassifications, our noninterference guarantee is similar to the *noninterference until conditions* property provided by Chong and Myers [5]. Both our definitions of noninterference consider only declassification-free subtraces of the execution. Our noninterference guarantee however permits certain classes of declassifications to occur without necessitating a truncation of the trace.

6 Conclusions

This paper has presented RX, a security-typed language that supports dynamic updates to role-based information-flow policies. The main contributions of this work are: (1) The novel use of role-based policies to provide a natural administrative model for managing policies in long-running programs. (2) A language design that allows programmatic addition and deletion of the policy statements that define roles along with a transaction mechanism that ensures that policies are applied consistently. (3) The novel use of metapolicies for preventing illegal flows of information through changes to policy. (4) A static type system and accompanying proof that the type system enforces a form of noninterference.

Although we have not studied the issue here, we expect that the transactional approach will scale better to systems with concurrent threads, each of which might try to update the global information-flow policy concurrently. The transactional model is also likely to be useful when policy updates are asynchronous, or in a distributed environment. The techniques presented in this paper provide some of the groundwork for achieving our long-term objective of handling these idioms.

References

[1] A. Banerjee and D. A. Naumann. Using access control for secure information flow in a Java-like language. In *Computer Security Foundations Workshop*. IEEE Computer Society Press, June 2003.

[2] M. Y. Becker. Cassandra: flexible trust management and its application to electronic health records, Oct. 2005.

[3] N. Broberg and D. Sands. Flow Locks: Towards a Core Calculus for Dynamic Flow Policies. In *European Symposium on Programming*, 2006.

[4] H. Chen and S. Chong. Owned Policies for Information Security. In *Computer Security Foundations Workshop*, 2004.

[5] S. Chong and A. C. Myers. Security policies for downgrading. In *Proceedings of the ACM conference on Computer and communications security*, pages 198–209, New York, NY, USA, 2004. ACM Press.

[6] S. Chong and A. C. Myers. Language-based information erasure. In *CSFW*, pages 241–254, 2005.

[7] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. Simple public key certificates. Internet Draft (work in progress), July 1999. Available at `http://world.std.com/~cme/spki.txt`.

[8] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. IETF RFC 2693, Sept. 1999.

[9] J. Y. Halpern and K. R. O'Neill. Anonymity and information hiding in multiagent systems. *J. Computer Security*, 13(3):483–514, 2005.

[10] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. Available at `http://research.microsoft.com/Users/simonpj/papers/stm/index.htm`.

[11] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic Updating of Information-Flow Policies. In *Foundations of Computer Security*, 2005.

[12] H. H. Hosmer. Metapolicies i. *SIGSAC Review*, 10(2-3):18–43, 1992.

[13] N. Li and J. C. Mitchell. Understanding SPKI/SDSI using first-order logic. In *Proceedings of the Computer Security Foundations Workshop*, 2003.

[14] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a Role-Based Trust-Management Framework. In *IEEE Symposium on Security and Privacy*, 2002.

[15] A. Matos and G. Boudol. On declassification and the nondisclosure policy. In *Computer Security Foundations Workshop*, 2005.

[16] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java + information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[17] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing Robust Declassification. In *Proc. of 17th IEEE Computer Security Foundations Workshop*, pages 172–186, Asilomar, CA, June 2004. IEEE Computer Society Press.

[18] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, Jan. 2003.

[19] Role based access control. `http://csrc.nist.gov/rbac/`, 2006.

[20] M. F. Ringenburg and D. Grossman. AtomCaml: First-Class Atomicity via Rollback. In *ACM International Conference on Functional Programming*, 2005.

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

[22] N. Shavit and D. Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[23] V. Simonet. FlowCaml in a nutshell. In G. Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, Mar. 2003.

[24] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 355–364, San Diego, California, Jan. 1998.

[25] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. Technical report, Computer Science Department, University of Maryland, Feb 2006. `www.cs.umd.edu/~nswamy/rx/tr.pdf`.

[26] S. Tse and S. Zdancewic. Run-time Principals in Information-flow Type Systems. In *IEEE 2004 Symposium on Security and Privacy*. IEEE Computer Society Press, May 2004.

[27] S. Tse and S. Zdancewic. Designing a security-typed language with certificate-based declassification. In *Proc. of the 10th European Symposium on Programming*, Lecture Notes in Computer Science, 2005.

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

[29] W. Weimer and G. C. Necula. Finding and preventing runtime error handling mistakes. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 419–431, New York, NY, USA, 2004. ACM Press.

[30] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In *Proceedings of theEuropean Conference on Object-Oriented Programming*, June 2004.

[31] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. of 14th IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Canada, June 2001.

[32] L. Zheng, S. Chong, S. Zdancewic, and A. C. Myers. Building Secure Distributed Systems Using Replication and Partitioning. In *IEEE 2003 Symposium on Security and Privacy*. IEEE Computer Society Press, 2003.

[33] L. Zheng and A. C. Myers. Dynamic Security Labels and Noninterference. In *Formal Aspects in Security and Trust*, 2004.