Managing Policy Updates in Security-Typed Languages

Nikhil Swamy Michael Hicks
University of Maryland, College Park
{nswamy, mwh}@cs.umd.edu

Stephen Tse Steve Zdancewic
University of Pennsylvania
{stse, stevez}@cis.upenn.edu

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 allow 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 flows*. 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

Security-typed programming languages extend standard types with labels to 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. Compile-time type-checking ensures that the policies expressed by labels mentioned in types 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. New principals may enter the system, while existing principals may leave or change 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 flows permitted by neither the old nor the new policy. We call this channel of information leaks across updates a *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 [12]. 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; policy queries executed at runtime allow the program to observe the evolution of policy. Programmers can use database-style transactions to denote code that must use a single consistent policy, preventing unintended transitive flows. Policy updates that would violate this consistency cause the program to roll back to a consistent state.

Once we allow policies to change within a program, policies themselves can become channels that carry sensitive information. To prevent these channels from leaking information to unauthorized principals, RX uses *metapolicies* that define which principals can view a particular role, and which principals trust a role's definition. To our knowledge, RX is the first programming language to formalize metapolicies to forbid illegal flows via policy updates. 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. Thus, only when the program is acting in a way trusted by that owner may the role be changed.

The RT policy language has 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 indirection: the contents 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 know, 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 by motivating the use of 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* [24] in which an information flow policy Π 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 $\sqsubseteq$. 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*—L and H 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 [14] 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: for instance, if according to Π, principal P_1 delegates to P_2, then P_2 may "act for" P_1. The ordering on labels is induced by this acts-for relation 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).

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 were not designed with policy changes in mind. 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 existing label models, we looked instead to the body of work on formal policy languages for which administrative models already exist. *Role-based* policy languages [17, 2, 6, 12] 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 [12]. Using RT roles as labels has a number of attractive administrative features:

1. *Ownership*: An RT role is 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 $Manager \sqsubseteq Alice$; i.e., 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 directly, 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 some 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_{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 [12]; 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 the function $owner(\rho)$ to extract the owner of ρ (so that $owner(P.r) = P$).

Policy statements s have two forms[1] $P.r \leftarrow \{P_1, \ldots, P_n\}$ 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 function $roledef(s)$ 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 according to a policy Π by the function $[\![\cdot]\!]_\Pi$. Intuitively, $[\![\rho]\!]_\Pi$ includes all elements of X where $\rho \leftarrow X \in \Pi$, along with all elements of $[\![\rho']\!]_\Pi$ where $\rho \leftarrow \rho' \in \Pi$. It is defined formally below.

$$
\begin{array}{lcl}
[\![\rho]\!]_\Pi & = & S_\Pi(\rho, \Pi) \\
S_{\Pi_0}(\rho, \emptyset) & = & \emptyset \\
S_{\Pi_0}(\rho, \{\rho \leftarrow X\} \cup \Pi) & = & X \cup S_{\Pi_0}(\rho, \Pi) \\
S_{\Pi_0}(\rho, \{\rho \leftarrow \rho'\} \cup \Pi) & = & [\![\rho']\!]_{\Pi_0 \setminus \{\rho \leftarrow \rho'\}} \cup S_{\Pi_0}(\rho, \Pi) \\
S_{\Pi_0}(\rho, \{s\} \cup \Pi) & = & S_{\Pi_0}(\rho, \Pi) \quad \text{if } roledef(s) \neq \rho
\end{array}
$$

An example of an RT_0 policy Π is given in Figure 1, 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 1, 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

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

principal	P		
principal sets	X	::=	$\{P_1, \ldots, P_n\}$
role	ρ	::=	$P.r$
policy stmt	s	::=	$\rho \leftarrow X \mid \rho_1 \leftarrow \rho_2$
policy	Π	::=	$\{s_1, \ldots, s_n\}$

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 1. Syntax of RT_0 and a sample policy.

of insurance companies accepted by the clinic. Finally, the last definition owned by $DrPhil$ includes only himself.

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

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

2.4 The Rx_{core} Programming Language

Rx_{core} is a simple imperative language with security labels. Its syntax is shown at the top of Figure 2. Labels ℓ in Rx_{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 is a lower security level than a role with fewer members, since strictly more principals can read data labeled by it. Extending this ordering to compound labels is straightforward by interpreting the join operator as set intersection.

Rx_{core} contains a single base type (bool) subscripted with a security level (We add another base type when extending Rx_{core} to Rx.). There are two typing judgments for Rx_{core}, shown at the bottom of Figure 2. Expression typings $\Omega \vdash E : \tau$ state that in context Ω the expression E has type τ. 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 types, and pc is simply a label ℓ that is used to bound the effect of writing to memory, to prevent indirect information flows [19]. We discuss Q below. In the

$$
\begin{array}{lrcl}
\text{atomic labels} & L & ::= & \rho \\
\text{compound labels} & \ell & ::= & L \mid \ell \sqcup \ell \\
\text{types} & t & ::= & \text{bool} \\
\text{security types} & \tau & ::= & t_\ell \\
\text{policy context} & Q & ::= & \Pi \\
\text{typing context} & \Omega & ::= & (\Gamma, \text{pc}, Q) \\
\text{expressions} & E & ::= & \texttt{true} \mid \texttt{false} \mid x \mid E_1 \oplus E_2 \\
\text{statements} & S & ::= & \texttt{skip} \mid x := E \mid S_1; S_2 \\
 & & & \mid \texttt{while}\ (E)\ S \mid \texttt{if}\ (E)\ S_1\ S_2
\end{array}
$$

$$\Omega \vdash \texttt{true} : \text{bool}_\ell \qquad \Omega \vdash \texttt{false} : \text{bool}_\ell \qquad \Omega \vdash x : \Omega.\Gamma(x)$$

$$\frac{\Omega \vdash E_1 : \text{bool}_{\ell_1} \quad \Omega \vdash E_2 : \text{bool}_{\ell_2}}{\Omega \vdash E_1 \oplus E_2 : \text{bool}_{\ell_1 \sqcup \ell_2}} \qquad \frac{\Omega \vdash S_1 \quad \Omega \vdash S_2}{\Omega \vdash S_1; S_2}$$

$$\Omega \vdash \texttt{skip} \qquad \frac{\Omega \vdash E : \text{bool}_\ell \quad \Omega[\text{pc} = \Omega.\text{pc} \sqcup \ell] \vdash S}{\Omega \vdash \texttt{while}\ (E)\ S}$$

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

$$\frac{\Omega.\Gamma(x) = t_\ell \quad \Omega \vdash E : t_\ell \quad \Omega.Q \vdash \Omega.\text{pc} \sqsubseteq \ell}{\Omega \vdash x := E}$$

$$\frac{\Omega \vdash E : \text{bool}_{\ell'} \quad \Omega.Q \vdash \ell' \sqsubseteq \ell}{\Omega \vdash E : \text{bool}_\ell}$$

Figure 2. RX_{core} **syntax and typing.**

typing rules we project the elements of the Ω tuple via the dot notation; for example, $\Omega.\text{pc}$ is the pc component of Ω. We write $\Omega[\text{pc} = \text{pc}']$ to represent the context that is identical to Ω except the pc component is replaced with the value pc' (and similarly for other components of a context).

As in other security-typed languages, type checking in RX_{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 RX_{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 RX in Section 3, for which policies Π will evolve over time. Other than this difference, the typing rules in Figure 2 are standard [24].

To illustrate how the typing judgments of RX_0 prevent illegal information flows, consider typing the following program in an environment where x is a high-security location and y a low-security location.

$$\texttt{if}\ (x)\ \ (y := \texttt{true})\ \ (y := \texttt{false})$$

In this program, although the contents of x are not directly assigned to y, the value stored in x is successfully copied into y. This is because the branches of the if-statement carry information about the contents of the high-security location x. To prevent such flows, the rule for if-statements checks each branch in a context where the effect lower-bound pc is strengthened to be no less than the security level of x. When typing the branches, the last premise of the rule for assignment requires the label of y to be no less than the effect lower-bound. In our example, since y is a low-security location, this premise is not satisfied and the program fails to type-check.

3 RX: Adding Policy Updates to RX_{core}

This section presents the remaining features of the full language 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—increasing the membership of a role effectively declassifies information [9]—RX adapts the integrity constraints from previous work on *robust declassification* [26, 15]. 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* [10] for protecting the confidentiality and integrity of roles.

3.1 RX Syntax

The syntax of RX is shown in Figure 3. It differs from RX_{core} in several ways. Atomic labels, L, now include abstract operators $C_\Pi(\rho)$ and $I_\Pi(\rho)$ 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 $\texttt{if}\ (q)\ S_1\ S_2$ 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.

Expressions E are augmented to include collections Δ of policy mutation statements δs. The type language is extended to include the type pol_ℓ which stands for the type of policy mutation statements at security level ℓ, where ℓ is defined by a metapolicy. The statement $\texttt{update}\ E$ is used to change the current policy by adding or deleting a collection of policy statements $\{s_1, \ldots, s_n\}$ where each s_i results from the evaluation of E to $\Delta = \delta_1 s_1, \ldots, \delta_n s_n$.

Finally, the statement $\texttt{trans}_Q\ S$ creates a transaction with policy context Q. Policy updates in S that violate pol-

atomic labels	L	::=	$\rho \mid C_\Pi(\rho) \mid I_\Pi(\rho)$
compound labels	ℓ	::=	$(L_C, L_I) \mid \ell \sqcup \ell$
types	t	::=	$\ldots \mid \text{pol}$
queries	q	::=	$L_1 \sqsubseteq L_2$
policy context	Q	::=	$\{q_1, \ldots, q_n\}$
update	δ	::=	`add` $\mid$ `del`
updates	Δ	::=	$\delta s \mid \delta s, \Delta$
expressions	E	::=	$\ldots \mid \Delta$
statements	S	::=	$\ldots \mid$ `if` $(q)\ S_1\ S_2$
			$\mid$ `update` $E \mid$ `trans` $_Q\ S$

Figure 3. RX syntax, based on RX_{core}.

icy assumptions stated in Q cause all memory effects of the S to be rolled back. This ensures that modifications to memory by S are consistent with respect to a single policy.

We first introduce the intuitive idea behind these new constructs by example. We then present the formal dynamic and static semantics. We 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 1:*

```
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]

The policy update statement executes only if the runtime policy Π satisfies the label ordering relation that appears in the second if-statement. Thus 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.

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 leak 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 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 the *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 $lab(\rho) = (C_\Pi(\rho), I_\Pi(\rho))$. Here the metapolicies $C_\Pi(\rho)$ and $I_\Pi(\rho)$ may depend on the owner of the role ρ and delegation information in the policy Π. Section 3.5 will discuss possible choices of metapolicy.

Example 2. *A program that leaks information across updates to the policy in Figure 1, motivating* RX*'s transactional semantics. Assume Γ as below:*

```
clinicRec   : bool(Clinic.staff, Clinic.staff),
patSymptoms : bool(Pat.healthRecords, Pat.healthRecords),
philRec     : bool(DrPhil.self, 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 members of the role Pat.healthRecords. Line `S1` copies this data into the $Clinic$ records, which is permitted by the policy in Figure 1. 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 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*.

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

[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.

$$\frac{\mathcal{E}.\Psi = \cdot \quad \Psi' = (\mathcal{E}.M, \texttt{trans}_Q\ S)}{\mathcal{E}, \texttt{trans}_Q\ S \longrightarrow \mathcal{E}[\Psi = \Psi'], \texttt{trans}_Q\ S}\ \text{(E-TR1)} \qquad \frac{\mathcal{E}.\Psi \neq \cdot \quad \mathcal{E}, S \longrightarrow \mathcal{E}', S'}{\mathcal{E}, \texttt{trans}_Q\ S \longrightarrow \mathcal{E}', \texttt{trans}_Q\ S'}\ \text{(E-TR2)}$$

$$\frac{\mathcal{E}.\Psi \neq \cdot}{\mathcal{E}, \texttt{trans}_Q\ \texttt{skip} \longrightarrow \mathcal{E}[\Psi = .], \texttt{skip}}\ \text{(E-TR3)} \qquad \frac{\mathcal{E}.\Psi \neq \cdot \quad \mathcal{E}, S \rightsquigarrow \mathcal{E}', S'}{\mathcal{E}, \texttt{trans}_Q\ S \longrightarrow \mathcal{E}', S'}\ \text{(E-TR4)} \qquad \frac{\mathcal{E}, S_1 \rightsquigarrow \mathcal{E}', S}{\mathcal{E}, S_1; S_2 \rightsquigarrow \mathcal{E}', S}\ \text{(R-SEQ)}$$

$$\frac{\begin{array}{c}\Pi' = \mathcal{E}.\Pi \cup \{s \mid \texttt{add}\ s \in \Delta\} \setminus \{s \mid \texttt{del}\ s \in \Delta\} \\ \mathcal{E}.\Psi = (M', \texttt{trans}_Q\ S) \quad \forall q \in Q.(\Pi \vdash q) \Leftrightarrow (\Pi' \vdash q)\end{array}}{\mathcal{E}, \texttt{update}\ \Delta \longrightarrow \mathcal{E}[\Pi = \Pi'], \texttt{skip}}\ \text{(E-UP1)} \qquad \frac{\mathcal{E}, E \longrightarrow \mathcal{E}, E'}{\mathcal{E}, \texttt{update}\ E \longrightarrow \mathcal{E}, \texttt{update}\ E'}\ \text{(E-UP2)}$$

$$\frac{\begin{array}{c}\Pi' = \mathcal{E}.\Pi \cup \{s \mid \texttt{add}\ s \in \Delta\} \setminus \{s \mid \texttt{del}\ s \in \Delta\} \\ \mathcal{E}.\Psi = (M', \texttt{trans}_Q\ S) \quad \exists q \in Q.(\Pi \vdash q) \not\Leftrightarrow (\Pi' \vdash q)\end{array}}{\mathcal{E}, \texttt{update}\ \Delta \rightsquigarrow \mathcal{E}[M = M'][\Pi = \Pi'], \texttt{trans}_Q\ S}\ \text{(R-UP)} \qquad \frac{\mathcal{E}.\Pi \vdash q \Rightarrow j = 1 \quad \mathcal{E}.\Pi \nvdash q \Rightarrow j = 2}{\mathcal{E}, \texttt{if}\ (q)\ S_1\ S_2 \longrightarrow \mathcal{E}, S_j}\ \text{(E-IFQ)}$$

Figure 4. RX execution ($\mathcal{E}, S \longrightarrow \mathcal{E}', S'$) and rollback ($\mathcal{E}, S \rightsquigarrow \mathcal{E}', S'$).

violated by the update to policy. This problem of unintentional flows motivates the support of a non-standard transactional model [18, 25] 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 contains all the necessary label ordering relations.

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 in `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 guarantees that no information leaks of this kind occur.

Our choice of transaction semantics is motivated by our belief that policy updates must take effect immediately. This behavior is particularly critical in the case of updates that revoke privileges. With this in mind, we have defined the semantics of transaction rollback to be such that the policy is updated immediately, and only the state of memory rolled back. Note that policy updates that occur in a transaction are treated like I/O operations in traditional transaction systems [18, 8]—only writes to memory are undone by a rollback, policy updates are left intact. Rolling back both the state of policy and memory is not feasible since this would guarantee non-termination through infinite rollback. As with traditional transaction systems, we could use compensations 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.

In situations where it is not essential for the update to take effect immediately, it might be desirable to choose a roll-forward semantics in which policy updates that violated consistency were delayed until the transaction completed execution. We explored such a semantics in a previous paper [9]. One of the contributions of this paper is showing that transactions can be rolled back in a secure manner.

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 is:

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

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.[3]

The rules, shown in Figure 4, 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 $\texttt{trans}_Q\ S$. (E-TR1) takes a new snapshot Ψ' of the current memory store M and the current statement $\texttt{trans}_Q\ S$ in the execution context $\mathcal{E}$, only if the current snapshot is empty. (E-TR2) is

[3] As discussed in Section 3.4, we support only non-nested transactions, for simplicity. So, no stack of snapshots is needed.

$$lab(L_1 \sqsubseteq L_2) \;=\; lab(L_1) \sqcup lab(L_2) \qquad lab(C_\Pi(\rho)) \;=\; lab(I_\Pi(\rho)) \;=\; lab(\rho) \;=\; (C_\Pi(\rho), I_\Pi(\rho))$$

$$Q \vdash C_\Pi(\rho) \sqsubseteq \rho \qquad Q \vdash I_\Pi(\rho) \sqsubseteq \rho \qquad Q \vdash \ell \sqsubseteq \ell \qquad \frac{Q \vdash \ell_1 \sqsubseteq \ell \quad Q \vdash \ell \sqsubseteq \ell_2}{Q \vdash \ell_1 \sqsubseteq \ell_2} \qquad \frac{L_1 \sqsubseteq L_2 \in Q}{Q \vdash L_1 \sqsubseteq L_2}$$

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

$$\frac{\ell = lab(roledef(s))}{\Omega \vdash \delta s : \mathrm{pol}_\ell}\ \text{(T-POL1)} \qquad \frac{\Omega \vdash \delta s : \mathrm{pol}_\ell \quad \Omega \vdash \Delta : \mathrm{pol}_{\ell'}}{\Omega \vdash \delta s, \Delta : \mathrm{pol}_{\ell \sqcup \ell'}}\ \text{(T-POL2)} \qquad \frac{\mathrm{pc}' = \mathrm{pc} \sqcup lab(q) \quad q \in \Phi.Q \quad \Gamma; \mathrm{pc}'; Q \cup \{q\}; \Phi \vdash S_1 \quad \Gamma; \mathrm{pc}'; Q; \Phi \vdash S_2}{\Gamma; \mathrm{pc}; Q; \Phi \vdash \texttt{if}\ (q)\ S_1\ S_2}\ \text{(T-IFQ)}$$

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

Figure 5. Rx **metapolicy labels (**$lab(\cdot)$**), label ordering (**$Q \vdash \ell_1 \sqsubseteq \ell_2$**) and typing (**$\Omega \vdash E : \tau,\ \Omega \vdash S$**).**

a congruence rule for evaluation within a transaction and (E-TR3) discards the snapshot when a transaction completes. (E-TR4) and (R-SEQ) use the rollback relation $\mathcal{E}, S \rightsquigarrow \mathcal{E}', S'$ triggered by failed updates to abort a transaction.

(E-UP1) takes the current policy $\mathcal{E}.\Pi$ and computes the new policy Π' by adding or deleting policy statements according to Δ_s, which is the result of evaluating each E that appears in Δ according to the rule (E-UP2). We omit the standard definition of the execution relation for expressions $\mathcal{E}, E \longrightarrow \mathcal{E}, E'$. However, the new policy Π' must be consistent with the query set Q which annotates the enclosing transaction statement $\texttt{trans}_Q\ S$ (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 policy context 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).

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):

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

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

```
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 $Q = \{A.r \sqsubseteq B.r\}$ 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 q and the then-branch of the statement is taken. We now have a sequence of statements with the first being an update statement `update(del(A.r ← 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 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 relations $\Omega \vdash E : \tau$ and $\Omega \vdash S$ in Figure 5, just like the typing relation for Rx_{core} in Figure 2. However, the typing context Ω now contains a *static snapshot* Φ for type checking transactions:

$$\begin{array}{lrcl} \text{typing context} & \Omega & ::= & (\Gamma; \mathrm{pc}; Q; \Phi) \\ \text{static snapshot} & \Phi & ::= & \cdot \mid (\mathrm{pc}, Q) \end{array}$$

Hence, we also write the typing judgment as $\Gamma; \mathrm{pc}; Q; \Phi \vdash S$. The type binding for variables in Γ and the program

counter pc are standard, and the policy context is already defined Figure 3. The snapshot Φ is used to approximate the assumptions of a transaction (explained below).

Metapolicy labels The first row of Figure 5 defines the auxiliary function $lab(\cdot)$ to compute the metapolicy label of policy queries q. The function $lab(\cdot)$ uses the metapolicy $C_\Pi(\cdot)$ and $I_\Pi(\cdot)$ to construct a label for a role. The assertion $lab(C_\Pi(\rho)) = lab(I_\Pi(\rho)) = lab(\rho)$ is the *meta-metapolicy*. It states that the metapolicies $C_\Pi(\rho)$ and $I_\Pi(\rho)$ only carry information about ρ. 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.

Label ordering Figure 5 (the second and third rows) specifies the label ordering relation $Q \vdash \ell_1 \sqsubseteq \ell_2$. In the second row of Figure 5, the first two rules impose conditions on the metapolicy. The first rule states that all members of a role ρ are permitted observe the definition of ρ; the second rule states that all members of a role ρ trust the definition of ρ. We discuss these conditions in more detail in Section 3.5. The remaining three rules on this row are straightforward: the left and the middle rules say that the relation is reflexive and transitive, and the rightmost rule makes use of the policy context Q when the labels L_1 and L_2 are atomic. In the third row of Figure 5, the left rule handles the compound label (L_C, L_I), and the middle and the right rules handle the join label $\ell \sqcup \ell'$.

Typing policy mutation statements The rule (T-POL1) assigns a policy mutation statement δs the type pol_ℓ where ℓ is the metapolicy associated with the role defined by the s. For a collection of policy mutation expressions Δ (T-POL2) states that the label in the type of Δ is the join of the labels assigned to each policy mutator that appears in Δ. For instance, if $\Delta = \texttt{add}\ A.r \leftarrow X, \texttt{del}\ B.r \leftarrow Y$ then the type of Δ is $\mathrm{pol}_{(C_\Pi(A.r), I_\Pi(A.r)) \sqcup (C_\Pi(B.r), I_\Pi(B.r))}$.

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 an augmented policy context $Q \cup \{q\}$. Second, both branches are checked using an elevated 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.

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.

Supporting nested transactions (assuming inner transactions can roll back without causing outer ones to rollback too) would require a flow-sensitive static analysis. Such an approach would also increase the precision of the static semantics and permit more updates. To simplify the dynamic semantics and typing rules, (T-TR) must occur in an empty policy context, thus preventing nested transactions. 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 $(C_\Pi(\rho), I_\Pi(\rho))$, where the metapolicy $C_\Pi(\rho)$ is the set of principals who are permitted to view the members of ρ, and the metapolicy $I_\Pi(\rho)$ 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 $I_\Pi(\rho)$; this is a condition similar to robust declassification [26]. Moreover, according to the metapolicy, the change in a role definition ρ reveals information about the context to principals in $C_\Pi(\rho)$. The first two premises of (T-UP) (in a manner analogous to the rule for assignments in Figure 2) ensures that members of $C_\Pi(\rho)$ are permitted to gain information about the context. In particular, the pc must be no more confidential and no less trustworthy than the confidentiality and integrity levels of the role, thus ensuring that the role is not improperly updated, and that its update does not leak information. Note that to ensure that the owner of a role is permitted to modify its definition, any metapolicy $I_\Pi(\rho)$ must include the owner of the role.

Example 4. *An instantiation of the typing rule (T-UP) for policy updates in Figure 5. We abbreviate role names to save space.* 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

$$\Gamma = \texttt{patAcceptsTreatment} : \mathrm{bool}_{(C_\Pi(Pat.\mathrm{drs}), I_\Pi(Pat.\mathrm{drs}))}$$
$$\mathrm{pc} = (C_\Pi(Pat.\mathrm{drs}), I_\Pi(Pat.\mathrm{drs}))$$

The variable `patAcceptsTreatment` determines whether Pat's role $Pat.\mathrm{drs}$ should be updated. The label on its type, $(C_\Pi(Pat.\mathrm{drs}),\ I_\Pi(Pat.\mathrm{drs}))$, indicates that information flows from this variable to the definition of the role $Pat.\mathrm{drs}$. The pc at the start of the transaction will be added to the snapshot Φ by (T-TR). The instance of (T-UP) that checks the `update` statement appears within a derivation that includes (T-IFQ). (T-IFQ) checks the then-branch of the policy query statement by augmenting the policy context Q to include $\{Clinic.\mathrm{ins} \sqsubseteq Pat.\mathrm{ins}\}$, while the program counter is strengthened to pc' to reflect the security level of the query. The instantiation of the (T-UP) rule in the derivation is as follows:

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

where

$$\begin{array}{rcl} \mathrm{pc}' & = & \mathrm{pc} \sqcup (C_\Pi(Clinic.ins), I_\Pi(Clinic.ins)) \\ & & \quad \sqcup (C_\Pi(Pat.ins), I_\Pi(Pat.ins)) \\ lab(Pat.\mathrm{drs}) & = & (C_\Pi(Pat.drs), I_\Pi(Pat.drs)) \end{array}$$

If Q were $\emptyset$, it would not be sufficient to prove the first premise according to the label ordering rules in Figure 5. This is because $\{Clinic.\mathrm{ins} \sqsubseteq Pat.\mathrm{ins}\}$ alone has nothing to say about the relationship between the metapolicies of the various roles. It would be sufficient to choose

$$\begin{array}{rlcl} Q = \{ & C_\Pi(Clinic.ins) & \sqsubseteq & C_\Pi(Pat.drs), \\ & C_\Pi(Pat.ins) & \sqsubseteq & C_\Pi(Pat.drs), \\ & I_\Pi(Clinic.ins) & \sqsubseteq & I_\Pi(Pat.drs), \\ & I_\Pi(Pat.ins) & \sqsubseteq & I_\Pi(Pat.drs)\ \} \end{array}$$

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-check these queries (and the one already in S) would require choosing the transaction's $Q' \supseteq Q \cup \{Clinic.\mathrm{ins} \sqsubseteq Pat.\mathrm{ins}\}$. □

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-UP1) and (R-UP) in Figure 4. We want to avoid leaking information about the queries through low-security data and low-security policy. The first case is handled by the third premise of the (T-UP) rule. It ensures that all memory effects in a transaction are bounded below by the 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 last premise ($Q \vdash lab(Q') \sqsubseteq \mathrm{pc}'$) of (T-UP), which requires that all the queries 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'. So, policy information is not leaked into memory via rollback. In our example typing, this third premise is clearly satisfied because $lab(Q') = (C_\Pi(DrBob.ins), I_\Pi(DrBob.ins)) \sqcup (C_\Pi(Pat.ins), I_\Pi(Pat.ins))$.

3.5 Requirements of a Metapolicy

RX uses metapolicies $C_\Pi(\rho)$ and $I_\Pi(\rho)$ 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. $[\![C_\Pi(\rho)]\!] = \{P_1, \ldots, P_n\}$ for some principals P_i, and similarly for $I_\Pi(\rho)$. Here we discuss possible interpretations of $C_\Pi(\rho)$ and $I_\Pi(\rho)$. We define sufficient conditions for metapolicy interpretations that enables proving noninterference.

A simple interpretation for role confidentiality is $[\![C_\Pi(\rho)]\!] = \bot$. 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, which may be too limiting.

An attempt to permit updates to occur in contexts dependent on secret data would have to define $[\![C_\Pi(\rho)]\!]$ to be more restrictive that $\bot$. An *anonymity* policy might, for instance, allow a principal to learn of its own membership but not that of others [7]. That is, not all members of ρ can compute the interpretation $[\![\rho]\!]_\Pi$. However, such a metapolicy is overly restrictive in that many simple programs will fail to type-check, as illustrated by the following example.

Example 5. *Consider checking the following program in a context* $\Gamma = x : \mathrm{bool}_{(B.r, B.r)}, y : \mathrm{bool}_{(A.r, A.r)}$*:*

$$\texttt{if}(A.\mathrm{r} \sqsubseteq B.\mathrm{r})\ x := y$$

Since the query carries information about the roles $A.r$ and $B.r$, (T-IFQ) checks the then-branch in a context with

$\mathrm{pc} = (C_\Pi(A.r), I_\Pi(A.r)) \sqcup (C_\Pi(B.r), I_\Pi(B.r))$ and $Q = \{A.r \sqsubseteq B.r\}$. To justify the flow of information from y to x the rule for assignments requires $A.r \sqsubseteq B.r$, the evidence for which is provided by Q. The mutation of location x that results from this assignment is observable by all members of $B.r$. Therefore the rule for assignments must also show $\mathrm{pc} \sqsubseteq (B.r, B.r)$, so that information about the query is not leaked to unauthorized principals. If the metapolicy is such that $[\![C_\Pi(B.r)]\!]$ doesn't include $[\![B.r]\!]_\Pi$, then $\mathrm{pc} \sqsubseteq (B.r, B.r)$ cannot be satisfied and the program fails to type-check.

Intuitively, by observing the write to location x, all members of $B.r$ gain information about $[\![B.r]\!]_\Pi$. To be able to write programs in which information flows across security levels (from low-security to high-security), we must ensure that the policy conditions that are necessary to justify the flow of information into a particular memory location are not more confidential than the contents of that location. This requirement is expressed formally in Figure 5 as $Q \vdash C_\Pi(\rho) \sqsubseteq \rho$. A similar argument explains the need for $Q \vdash I_\Pi(\rho) \sqsubseteq \rho$. □

Though intuitive, allowing $C_\Pi(\rho)$ to include only the members (and the owner of ρ) is not sufficient. A policy that includes *delegations* permits information to flow between roles that are related by delegation. These flows could possibly reveal secret information. To see why, consider the example from Figure 1. In the example, the definition of the role Pat.doctors is 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 a choice of metapolicy where $[\![C_\Pi(\rho)]\!]$ includes only the members of ρ and the role's owner, we permit $DrSue$ to view the interpretation of Pat.doctors although she is not permitted to view the interpretation of $Clinic$.staff. However, any change in the definition of $Clinic$.staff (say, if $DrAlice$ is removed) is reflected in the interpretation of Pat.doctors. Hence, even though $DrSue$ is not a member of $Clinic$.staff, she can observe the effect of changes to that role. 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. We formally state this constraint on the confidentiality metapolicy below (A similar constraint must hold for the integrity metapolicy $I_\Pi(\rho)$.).

$$\forall \Pi. \forall \rho, \rho'. (\exists s. roledef(s) = \rho' \wedge [\![\rho]\!]_\Pi \neq [\![\rho]\!]_{\Pi \cup \{s\}}) \Rightarrow [\![C_\Pi(\rho)]\!] \subseteq [\![C_\Pi(\rho')]\!]$$

Informally, this constraint reads: "if the interpretation of the role ρ depends on the definition of ρ', then the metapolicy for ρ must be at least as restrictive as the metapolicy for ρ'." Intuitively, ρ depends on ρ' if ρ delegates transitively to ρ'. Note that an interpretation that satisfies this condition 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, which is the approach we adopt here. While simple, this permits 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 $I_\Pi(A.r)$ and $I_\Pi(B.r)$. We leave exploration of this issue to future work.

We don't extend the subtyping relation $\sqsubseteq$ given in Figure 2 to pol_ℓ types. The following example illustrates what might go wrong if we allowed covariant subtyping for pol_ℓ as we do for bool_ℓ.

Example 6. *Assume the existence of a covariant subtyping rule for* pol_ℓ *and consider the program below checked in a context with* $\Gamma = x : \mathrm{pol}_{(C_\Pi(B.r), I_\Pi(B.r))}$.

```
trans{C_Π(A.r)⊑C_Π(B.r), I_Π(A.r)⊑I_Π(B.r)}
   if(C_Π(A.r) ⊑ C_Π(B.r))
        if(I_Π(A.r) ⊑ I_Π(B.r))
             x := add A.r ← C;
trans{}
      update(del A.r ← B.r);
      update(x)
```

The type of the policy statement in the assignment is $\mathrm{pol}_{(C_\Pi(A.r), I_\Pi(A.r))}$. The policy queries provide the necessary evidence for the covariant subtyping judgment for pol_ℓ to permit the assignment to x. A separate transaction deletes the delegation $A.r \leftarrow B.r$ from the policy. Since the interpretation of the metapolicies $C_\Pi(\cdot)$ and $I_\Pi(\cdot)$ depend in general on the the state of the policy Π and in particular the delegations between roles in Π, the deletion of a delegation in the second transaction can violate the assumptions of the first transaction. This has the effect of destroying the evidence for subtyping necessary to check the assignment to x. The final update statement updates the role $A.r$. Even though at runtime the effect of this update is observable by all members of $C_\Pi(A.r)$, the type of x indicates that the update is observable only by members of $B.r$. □

Treating pol_ℓ as invariant is one way of ensuring updates that use first-class policy statements do not leak information even in the the presence of non-monotonic updates to policy. An alternative might be to permit subtyping for pol_ℓ while imposing constraints on how policy is allowed to evolve. We leave examining this alternative to future work.

A further condition on metapolicies $C_\Pi(\rho)$ and $I_\Pi(\rho)$ is induced by our definition in Figure 5 of meta-metapolicy through $lab(C_\Pi(\rho)) = lab(I_\Pi(\rho)) = lab(\rho)$. The metapolicy $C_\Pi(\cdot)$ is a function that maps a role to a set of principals. The interpretation 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 Π. In such a case, since $C_\Pi(\rho)$ carries information about ρ and $\rho_1, \ldots, \rho_n$, the label of $C_\Pi(\rho)$ should be $(\sqcup_i C_\Pi(\rho_i)) \sqcup C_\Pi(\rho)$. Thus, for our definition of $lab(C_\Pi(\rho)) = lab(\rho)$ to be sound, the metapolicy must also satisfy the following condition.

$$\forall \Pi. \forall \rho, \rho'. (\exists s. roledef(s) = \rho' \wedge [\![C_\Pi(\rho)]\!] \neq [\![C_{\Pi \cup \{s\}}(\rho)]\!]) \Rightarrow [\![C_\Pi(\rho)]\!] \subseteq [\![C_\Pi(\rho')]\!]$$

An identical condition must also hold true for $I_\Pi(\rho)$. For a more complete treatment of metapolicy including possibly explicit higher-order metapolicies, see our technical report [21].

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 (memory and policy) with which the program executes. Updates to policy intentionally alter the security behavior of the program, possibly revealing previously secret information [9]. 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 during the program's execution are consistent with the policy at that time. 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 or termination 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 7. *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:*

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

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. The following lemma ensures that this connection is sound.

Lemma 8 (Static Label Ordering Soundness)**.** *For all contexts Ω and programs S, if the derivation of $\Omega \vdash S$ contains a sub-derivation $\Omega' \vdash S'$, then the following holds true for all policies Π:*

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

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) trace and $\mathcal{E}, \alpha$ to denote the concatenation of a single configuration and a trace.

We define the attacker's observation level as a set of roles R. We assume the existence of a type environment Γ. The restriction of a trace α to observation level R is written $\alpha\,|_R$, and is defined in Figure 6. As long as the policy remains unchanged, a restricted trace consists of a restriction to each configuration element of the trace (the "otherwise" clause of the **Trace** definition of the figure). In doing so, we restrict the view of memory according to the policy Π and the $\Omega.\Gamma$ used to type check the initial program. Here $lab(\Gamma(x))$ refers to the security label associated with the contents of the location x. We restrict the policy according to the metapolicy $C_\Pi(\rho)$, which must satisfy the condition described in Section 3.5. However, if a policy update results in a declassification with respect to the observer's roles R then the trace is truncated (the first clause of the **Trace** definition of the figure). This truncation is justified since declassifications due to policy update are intentional releases of information. For a formal definition of the predicate $declassify(\cdot, \cdot, \cdot)$ refer to the full version of our paper [21]. 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) does not require the trace to be truncated.

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 $(\Pi, M, \Psi)\,|_R^\psi = (\Pi\,|_R, M|_{R,\Pi}, \Psi|_{R,\Pi})$.

The definition of trace observability implies that computation steps are only observable if they have an effect on an

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

Policy :
$\Pi|_R = \Pi\|_{Obs(R,\Pi)}$
$\emptyset\|_R = \emptyset \quad (\{s\} \cup \Pi')\|_R = \begin{cases} \{s\} \cup (\Pi'\|_R) & roledef(s) \in R \\ \Pi'\|_R & \text{otherwise} \end{cases}$

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

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

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

Trace :
$(\mathcal{E}_1, \mathcal{E}_2, \alpha)|_R = \begin{cases} \mathcal{E}_1|_R \text{ if } declassify(R, \mathcal{E}_2.\Pi, \mathcal{E}_1.\Pi) \\ \mathcal{E}_1|_R, (\mathcal{E}_2, \alpha)|_R \qquad \text{otherwise} \end{cases}$

Figure 6. Trace observability.

observable part of memory or policy. This entails that we identify traces only up to stuttering.[4] We write $\alpha \doteq \beta$ if α and β are equivalent up to stuttering.

Theorem 9 (Noninterference)**.** *Suppose that 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*

$$\mathcal{E}_0 \mid^{\psi}_{R} = \mathcal{E}_1 \mid^{\psi}_{R} \Rightarrow \mathrm{Tr}(\langle \mathcal{E}_0, S\rangle) \mid_R \doteq \mathrm{Tr}(\langle \mathcal{E}_1, S\rangle) \mid_R$$

The proof (in our technical report [21]) uses Pottier and Simonet's proof technique [16] 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 non-interferring 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 [17], SPKI [6]. RX policies are based on those from RT framework by Li, Mitchell and Winsborough [12], which is similar to SPKI/SDSI [11]. The RX transaction semantics is inspired by software transactional memory [20].

There has been much prior work on language-based enforcement of information-flow policies [19]. 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 [14] supports dynamic queries of the security lattice and includes features for using both dynamic principals and dynamic labels [22, 27, 23]. Jif 2.0 also allows delegations between principals to change at run time, but does not prevent information leaks through policy updates.

The predecessor [9] 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), by using roles and metapolicies to form an administrative model (the term *metapolicy* is due to Hosmer [10]) and by introducing transactions to ensure policy consistency.

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], and lexically-scoped flow policies [13]. RX is designed to support unrestricted changes to policy during execution. Since RX supports first-class policy mutation statements the content of an update statement is not fully known statically. The intent is to support even more general models of policy update statements by following techniques of dynamic labels and run-time principals.

When policy 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. Our approach to obtain an end-to-end security guarantee by piecing together non-declassifying subtraces yields a property similar to the *nondisclosure* property proposed by Almeida Matos and Boudol [13]. Their approach of using a labeled transition semantics has

[4] Sequence α_1 is equivalent up to stuttering to α_2 if $\alpha'_1 = \alpha'_2$, where α'_i is obtained from α_i by removing all consecutively repeated elements from α_i. For example, the sequence $aabbbc$ is equivalent up to stuttering to $abbccc$ since the result of removing consecutively repeated elements from each sequence is abc.

the benefit of making explicit the concatenation of non-declassifying subtraces. However, their attacker model does not consider the state of policy to be a channel of information.

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.

It is for large distributed systems characterized by mutual distrust that the need for a principled approach to security is most pressing. To become a relevant technology for this kind of setting, security-typed languages must be able to cope with highly dynamic environments in which policy evolution is the norm rather than the exception. 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. 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 designing a language that can provide strong guarantees of security for complex, realistic applications.

Acknowledgments We thank Jeff Foster, Boniface Hicks, Polyvios Pratikakis, Saurabh Srivastava, and the anonymous reviewers for their comments. Funding for this research was provided in part by NSF grants CCF-0346989, CCF-0524036, CCF-0524305, CNS-0346939 and CCR-0311204.

References

[1] A. Banerjee and D. A. Naumann. Using access control for secure information flow in a Java-like language. In *CSFW*, June 2003.

[2] M. Y. Becker. Cassandra: flexible trust management and its application to electronic health records. Technical Report UCAM-CL-TR-648, University of Cambridge, Computer Laboratory, 2005.

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

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

[5] S. Chong and A. C. Myers. Security policies for downgrading. In *CCS*, 2004.

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

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

[8] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP*, 2005.

[9] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic Updating of Information-Flow Policies. In *FCS*, 2005.

[10] H. H. Hosmer. Metapolicies 1. *SIGSAC Review*, 10(2-3):18–43, 1992.

[11] N. Li and J. C. Mitchell. Understanding SPKI/SDSI using first-order logic. In *CSFW*, 2003.

[12] 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.

[13] A. Matos and G. Boudol. On declassification and the non-disclosure policy. In *CSFW*, 2005.

[14] 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.

[15] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing Robust Declassification. In *CSFW*, 2004.

[16] F. Pottier and V. Simonet. Information flow inference for ML. *TOPLAS*, 25(1), Jan. 2003.

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

[18] M. F. Ringenburg and D. Grossman. AtomCaml: First-Class Atomicity via Rollback. In *ICFP*, 2005.

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

[20] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, 1995.

[21] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. Technical Report CS-TR-4793, University of Maryland, 2006. `www.cs.umd.edu/~nswamy/rx/tr.pdf`.

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

[23] S. Tse and S. Zdancewic. Designing a security-typed language with certificate-based declassification. In *ESOP*, Lecture Notes in Computer Science, 2005.

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

[25] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In *ECOOP*, 2004.

[26] 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.

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