Managing Policy Updates in Security-Typed Languages [*]

University of Maryland, Technical Report CS-TR-4793

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 exam-

[*]This is the full-length version of a paper that appeared in: N. Swamy, M. Hicks, S. Tse, and S.Zdancewic; Managing Policy Updates in Security-Typed Languages; In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW-2006)*; Venice, Italy; July 5-7, 2006

ple, 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 $\text{RX}_{\mathsf{core}}$, 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 $\text{RX}_{\mathsf{core}}$ 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 6 and future directions in Section 7.

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 $\text{RX}_{\mathsf{core}}$, 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 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 [21]) 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*—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$_{\text{core}}$, 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] $P.r \longleftarrow \{P_1, \ldots, P_n\}$ and $P_1.r_1 \longleftarrow P_2.r_2$. 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, $roledef(A.r \longleftarrow \{B\})$ 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 $\rho \longleftarrow X \in \Pi$, along with all elements of $[\![\rho']\!]^{\Pi}$ where $\rho \longleftarrow \rho' \in \Pi$. It is defined formally below.

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

[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	$::=$	$\rho \longleftarrow X \mid \rho_1 \longleftarrow \rho_2$
policy	Π	$::=$	$\{s_1, \ldots, s_n\}$

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

Figure 1. Syntax of RT_0 and a sample policy.

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 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$_{\text{core}}$ Programming Language

Rx$_{\text{core}}$ is a simple imperative language with security labels. Its syntax is shown at the top of Figure 2. Labels ℓ in Rx$_{\text{core}}$ 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:

$$\Pi \vdash \rho_1 \sqsubseteq \rho_2 \iff [\![\rho_2]\!]^{\Pi} \subseteq [\![\rho_1]\!]^{\Pi}$$

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

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

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

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

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

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

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

Figure 2. RX$_{\mathsf{core}}$ **syntax and typing.**

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$_{\mathsf{core}}$ contains a single base type (bool) subscripted with a security level (We add another base type when extending RX$_{\mathsf{core}}$ to RX.). There are two typing judgments for RX$_{\mathsf{core}}$, 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 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 similarly for other components of a context).

As in other security-typed languages, type checking in RX$_{\mathsf{core}}$ 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$_{\mathsf{core}}$ 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.

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

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$_{\mathsf{core}}$

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* [27, 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.

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	$::=$	$\ldots \mid \mathrm{pol}$
queries	q	$::=$	$L_1 \sqsubseteq L_2$
policy context	Q	$::=$	$\{q_1, \ldots, q_n\}$
update	δ	$::=$	$\texttt{add}\,() \mid \texttt{del}\,()$
updates	Δ	$::=$	$\delta s \mid \delta s, \Delta$
expressions	E	$::=$	$\ldots \mid \Delta$
statements	S	$::=$	$\ldots \mid \texttt{if}\ (q)\ S_1\ S_2$
			$\mid \texttt{update}\ E \mid \texttt{trans}_Q\ S$

Figure 3. Rx **syntax, based on** $\mathrm{Rx}_{\mathsf{core}}$**.**

3.1 Rx Syntax

The syntax of Rx is shown in Figure 3. It differs from $\mathrm{Rx}_{\mathsf{core}}$ 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 `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 `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 policy 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

[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.

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     : $\text{bool}_{(\mathit{Clinic}.\text{staff},\mathit{Clinic}.\text{staff})}$,
patSymptoms   : $\text{bool}_{(\mathit{Pat}.\text{healthRecords},\mathit{Pat}.\text{healthRecords})}$,
philRec       : $\text{bool}_{(\mathit{DrPhil}.\text{self},\mathit{DrPhil}.\text{self})}$
```

```
S1:  if($Pat$.healthRecords $\sqsubseteq$ $Clinic$.staff)
       clinicRec := patSymptoms;
S2:  if(leaveClinic)
       update(del($Pat$.doctors $\longleftarrow$ $Clinic$.staff));
S3:  update(add($Clinic$.staff $\longleftarrow$ {$DrPhil$}));
S4:  if($Clinic$.staff $\sqsubseteq$ $DrPhil$.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 violated by the update to policy. This problem of unintentional flows motivates the support of a non-standard transactional model [18, 26] to our language RX. The semantics of a transaction $\texttt{trans}_\Phi$ 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 $\mathcal{E}, S \leadsto \mathcal{E}', S'$ for rollback. The rules for standard constructs (assignment, addition, sequences etc.) are not shown.

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

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

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

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

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

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

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

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

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

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

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

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

The rules (E-TR1), (E-TR2), (E-TR3) and (E-TR4) are for the execution of transaction statement $\mathtt{trans}_Q\ S$. (E-TR1) takes a new snapshot Ψ' of the current memory store M and the current statement $\mathtt{trans}_Q\ S$ in the execution context $\mathcal{E}$, only if the current snapshot is empty. (E-TR2) is 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 $\mathtt{trans}_Q\ S$ (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* $\{A.r \longleftarrow B.r, B.r \longleftarrow \{B\}\}$*:*

```
trans$_{\{A.r \sqsubseteq B.r\}}$
  if(A.r $\sqsubseteq$ B.r) {
   update(del(A.r $\longleftarrow$ 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` $\longleftarrow$ 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 ($\{B.r \longleftarrow \{B\}\}$), 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 $\{B.r \longleftarrow \{B\}\}$.

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$_{\mathsf{core}}$ in Figure 2. However, the typing context Ω now contains a *static snapshot* Φ for type checking transactions:

typing context	Ω	$::=$	$(\Gamma; \mathrm{pc}; Q; \Phi)$
static snapshot	Φ	$::=$	$\cdot \mid (\mathrm{pc}, Q)$

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 $\Delta =$ `add` $(A.r \longleftarrow X)$, `del` $(B.r \longleftarrow Y)$ then the type of Δ is $\mathrm{pol}_{(C_\Pi(A.r), I_\Pi(A.r)) \sqcup (C_\Pi(B.r), I_\Pi(B.r))}$.

Note that we do not give a subsumption rule for pol_ℓ. This restriction on pol_ℓ can be understood by thinking of a policy statement s as a reference to component of policy. Since policy updates can mutate policy it is not sound to use subtyping for policy statements. Further intuition for the invariance of this type under subtyping is given in Section 3.5.

Typing policy queries The rule (T-IFQ) type checks policy query statement `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.

Example 4. *An instantiation of the typing rule (T-IFQ) for policy queries for Example 1, assuming the policy query appears within a* `trans`$_Q$ *statement, for an appropriate* Q*.* (Here we elide the else branch and abbreviate

Clinic.insuranceCos and *Pat*.insurers to save space.)

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

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

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

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.

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 [27]. 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 5. *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

$$\begin{array}{lcl} \Gamma & = & \texttt{patAcceptsTreatment} : \mathrm{bool}_{(C_\Pi(\mathit{Pat}.\mathrm{drs}), I_\Pi(\mathit{Pat}.\mathrm{drs}))} \\ \mathrm{pc} & = & (C_\Pi(\mathit{Pat}.\mathrm{drs}), I_\Pi(\mathit{Pat}.\mathrm{drs})) \end{array}$$

The variable `patAcceptsTreatment` determines whether *Pat*'s role *Pat*.drs should be updated. The label on its type, $(C_\Pi(\mathit{Pat}.\mathrm{drs}),\ I_\Pi(\mathit{Pat}.\mathrm{drs}))$, indicates that information flows from this variable to the definition of the role *Pat*.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 $\{\mathit{Clinic}.\mathrm{ins} \sqsubseteq \mathit{Pat}.\mathrm{ins}\}$, 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:

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

where

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

If Q were $\emptyset$, it would not be sufficient to prove the first premise according to the label ordering rules in Figure 5. 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

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

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

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-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 ($Q \vdash \mathit{lab}(Q') \sqsubseteq \mathrm{pc}'$) 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 $\mathit{lab}(Q') = (C_\Pi(\mathit{DrBob}.\mathrm{ins}), I_\Pi(\mathit{DrBob}.\mathrm{ins})) \sqcup (C_\Pi(\mathit{Pat}.\mathrm{ins}), I_\Pi(\mathit{Pat}.\mathrm{ins}))$.

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 than $\bot$. 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 6. *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 $[\![C_\Pi(B.r)]\!]$ does not 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 $[\![\mathit{Pat.doctors}]\!]^\Pi = \{\mathit{DrAlice}, \mathit{DrBob}, \mathit{DrSue}\}$. 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 $I_\Pi(\rho)$.).

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

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 $A.r \longleftarrow B.r$ 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 do not 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 7. *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$_{\{C_\Pi(A.r)\sqsubseteq C_\Pi(B.r),I_\Pi(A.r)\sqsubseteq I_\Pi(B.r)\}}$
   if($C_\Pi(A.r) \sqsubseteq C_\Pi(B.r)$)
      if($I_\Pi(A.r) \sqsubseteq I_\Pi(B.r)$)
         $x$ := add ($A.r \longleftarrow C$) ;
trans$_{\{\}}$
      update(del ($A.r \longleftarrow B.r$) );
      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 $A.r \longleftarrow B.r$ 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 $\mathit{lab}(C_\Pi(\rho)) = \mathit{lab}(I_\Pi(\rho)) = \mathit{lab}(\rho)$. 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 $\mathit{lab}(C_\Pi(\rho)) = \mathit{lab}(\rho)$ to be sound, the metapolicy must also satisfy the following condition.

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

An identical condition must also hold true for $I_\Pi(\rho)$.

In this section we have identified properties that must be fulfilled by a metapolicy for the noninterference theorem to hold. We have, however, left a specific choice of metapolicy undetermined. In Appendix C, we show that it is possible to construct a non-trivial metapolicy (a metapolicy that is not the constant function $\bot$) that satisfies the soundness conditions identified here. We devise this metapolicy in the context of the full RT_0 language — that is, we include the linking and intersection delegation constructs that were elided from the main presentation. We call this particular metapolicy $C^{\mathrm{del}}(\cdot)$ because its definition depends on the delegation structure of a policy. Preliminary investigation suggests that delegation in the metapolicy language, while adding flexibility, forces well-formed metapolicies to be relatively coarse grained. We leave to future work a full investigation of this issue.

4 Noninterference

This section states a noninterference property for RX. The proof is developed in detail in Appendices A and B. 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 8. *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:*

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

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. The following lemma ensures that this connection is sound.

Lemma 9 (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 $\mathit{lab}(\Gamma(x))$ 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). The relation $\mathit{dclas} R, \Pi_1, \Pi_2$ is non-empty when the policy Π_2 results from declassifications of memory or policy with respect to the policy Π_1. Its definition depends on a classification (dependent on an observation level R) of roles in a policy Π into three sets $\rho_L R, \Pi, \rho_M R, \Pi, \rho_H R, \Pi$, which stand intuitively for low, medium and high-security roles respectively. Informally, $\mathit{dclas}(R, \Pi_1, \Pi_2)$ contains roles that are low security in Π_2 that were medium or high-security in Π_1 and those that are medium-security in Π_2 which were high-security in Π_1. A detailed description of the definition of $\mathit{dclas}(R, \Pi_1, \Pi_2)$ is given in Sections A.2 and A.3 of the Appendix.

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 in the definition of $(\mathcal{E}_1, \mathcal{E}_2, \alpha)|_R$.

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

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. [4] We write $\alpha \doteq \beta$ if α and β are equivalent up to stuttering.

Theorem 10 (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*

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

[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*.

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 :
$(\mathcal{E}_1, \mathcal{E}_2, \alpha)|_R = \begin{cases} \mathcal{E}_1|_R \text{ if } dclas(R, \mathcal{E}_1.\Pi, \mathcal{E}_2.\Pi) \neq \emptyset \\ \mathcal{E}_1|_R, (\mathcal{E}_2, \alpha)|_R \quad \text{otherwise} \end{cases}$
where

$$dclas(R, \Pi_1, \Pi_2) \equiv \begin{array}{l} (\rho_L(R, \Pi_2) \cap \rho_M(R, \Pi_1)) \cup \\ (\rho_L(R, \Pi_2) \cap \rho_H(R, \Pi_1)) \cup \\ (\rho_M(R, \Pi_2) \cap \rho_H(R, \Pi_1)) \end{array}$$

$$\rho_L(R, \Pi) \equiv \{\rho \mid \exists \rho_R \in R.\Pi \vdash C_\Pi(\rho) \sqsubseteq \rho_R \ \wedge\ \Pi \vdash \rho \sqsubseteq \rho_R\}$$

$$\rho_M(R, \Pi) \equiv \{\rho \mid \begin{array}{l} \exists \rho_R \in R.\Pi \vdash C_\Pi(\rho) \sqsubseteq \rho_R \ \wedge \\ \forall \rho_R \in R.\Pi \vdash \rho \not\sqsubseteq \rho_R \end{array} \}$$

$$\rho_H(R, \Pi) \equiv \{\rho \mid \forall \rho_R \in R.\Pi \vdash C_\Pi(\rho) \not\sqsubseteq \rho_R \ \wedge\ \Pi \vdash \rho \not\sqsubseteq \rho_R\}$$

Figure 6. Trace observability.

The proof presented in Appendices A and B 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-interfering 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 The Transaction Model

The transaction model introduces some subtleties that need to be considered when writing RX programs. We treat updates to policy much as I/O effects are treated in other languages that use software transactional memory [18, 8]. That is, updates to policy are not reverted when an inconsistency is detected; a rollback only restores the memory and control of the program. After the rollback is complete, execution resumes under the new policy. To illustrate the implications of these semantics, consider the program below that is an extension of the program ing Example 1, evaluating under the RT_0 policy in Figure 1

```
trans$_Q$
   if(patAcceptsTreatment)
     if($Clinic$.insuranceCos $\sqsubseteq$ $Pat$.insurers)
       x := 1 ;
       update(add $Pat$.doctors $\longleftarrow$ $Clinic$.staff);
   update(del DrBob.insurance $\longleftarrow$ $\{BCBS\}$)
```

The second update statement in this program violates the label ordering relation established by the policy query. The rollback that results causes the assignment to x to be reverted, but the effect of both policy update statements remain. The transaction resumes execution under a policy that includes (Pat.drs $\longleftarrow$ $DrBob$.staff) despite $DrBob$ not accepting payment from Pat's insurance company. This programming model requires therefore that a transaction be written so as to be re-entrant with respect to the state of policy. In this case, the programmer can install a *compensation* [25] to ensure that upon resumption of the transaction, the policy is in a consistent state according to the application semantics.

Since execution is deterministic, an alternative semantics in which effects to memory *and policy* are reverted when a transaction is rolled back is not tenable. This will cause the transaction to roll back repeatedly forever. Under the semantics chosen in this paper, a program that contained a series of mutually incompatible update statements can still fail to terminate because a transaction is rolled back repeatedly. This is illustrated by the following example, which does not terminate when executing under an initial policy in which the definitions of A.r and B.r are empty.

```
trans$_Q$
  update add $A$.r $\longleftarrow$ $B$.r;
  if($A$.r $\sqsubseteq$ $B.r$)
      update(del $A$.r $\longleftarrow$ $B$.r);
```

These issues indicate that using transactions to manage policy updates requires particular care on behalf of the programmer. One could argue that since we have assumed that the contents of policy update is part of the program text, it is possible to devise a static analysis that guarantees that an update does not result in an inconsistent policy. The inlined policy update statements are however an artifact of our desire to keep the presentation simple. The analysis we have presented here only places restrictions on the identity of the role being updated; the definition of the role is free to change arbitrarily. This allows for a straightforward extension to our full-language which supports first-class policy update statements and runtime principals. In that language, a reasonable static analysis for policy consistency is not possible and we have to resort to runtime policy consistency checks.

6 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, 28, 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 the benefit of making explicit the concatenation of non-declassifying subtraces. However, their attacker model does not consider the state of policy as a channel of information.

7 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, Will Winsborough, 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 nondisclosure 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] V. Simonet. FlowCaml in a nutshell. In G. Hutton, editor, *APPSEM-II*, pages 152–165, Mar. 2003.

[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] W. Weimer and G. C. Necula. Finding and preventing runtime error handling mistakes. In *OOPSLA '04*, pages 419–431, New York, NY, USA, 2004. ACM Press.

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

[27] 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.

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

A Representing Multiple Executions in Rx^2

A.1 Rx^2 Syntax

To prove noninterference for Rx we use Pottier and Simonet's proof technique [16], whereby a pair of *executions* of an Rx program is represented within the syntax of the language itself. We call this language Rx^2 and its syntax is defined as an extension of the syntax of Rx. We make two superficial changes to the syntax of Rx to make simplify the presentation of the the proof. First, transaction statements are annotated with a set of invariants Q, as well as a label ℓ, where ℓ represents the pc label used to check the transaction. The second modification is to exclude policy statements Δ from syntactic class of expressions. In Rx^2 policy statements Δ can only appear in the context of a policy update statement $\texttt{update}\ \Delta$.[5]

$$
\begin{array}{llll}
\text{Rx statements} & S & ::= & \ldots \mid \texttt{trans}_\Phi\ S \\
\text{expressions} & \mathsf{E} & ::= & v \mid \mathtt{x} \mid \mathsf{E}_1 \oplus \mathsf{E}_2 \mid \langle E_1 \parallel E_2 \rangle \\
\text{statements} & \mathsf{S} & ::= & \texttt{skip} \mid \mathtt{x} := \mathsf{E} \mid \mathsf{S}; S \\
 & & \mid & \texttt{while}\ (E)\ S \mid \texttt{if}\ (\mathsf{E})\ S_1\ S_2 \\
 & & \mid & \texttt{if}\ (q)\ S_1\ S_2 \mid \texttt{trans}_\Phi\ \mathsf{S} \\
 & & \mid & \texttt{update}\ \Delta \mid \langle S_1 \parallel S_2 \rangle
\end{array}
$$

We adopt the convention that syntactic elements that belong to Rx^2 are denoted in sans serif font – e.g $\mathsf{E}, \mathsf{S}, \mathsf{v}$ etc. while those in Rx are denoted by their serif counterparts E, S, v etc.

The significant extension here is that an expression E can be constituted from a pair of expressions E_1 and E_2 using the form $\langle E_1 \parallel E_2 \rangle$ – the expression on the left-side of the bracket, E_1, represents an expression in the first execution, and the expression on the right, E_2, represents an expression in the second execution. Similarly, statements S in Rx^2 can be pairs of Rx statements in bracketed form; i.e. $\langle S_1 \parallel S_2 \rangle$.

[5]The inclusion of Δ in the class of expressions poses no fundamental problems. Informally, the key to including Δ as an expression hinges on not including a subsumption rule for pol_ℓ types. As discussed in Section 3.5, this ensures that an expression of type $\mathrm{pol}_{C_\Pi(A.r), I_\Pi(A.r)}$ is a policy update statement that modifies the definition of precisely $A.r$. Thus the type pol_ℓ adequately indicates which roles are being updated. The proof of noninterference detailed here appeals only to the identity of the role being updated, and not the actual contents of the update.

An important point to note here is that bracketed statements and expressions cannot be nested within one another. The intention is for bracket terms to represent only *pairs* of executions — a bracket within a bracket does not make sense.

Finally, note that the syntax of $\textsc{Rx}^2$ does not permit bracketed statements from appearing in all positions. For instance, the while-statement does not contain brackets at all. The reason for this is that our proof of noninterference considers pairs of executions of a *single* Rx *program*. These executions make take place in contexts that can differ in their high-security components. The operational semantics of $\textsc{Rx}^2$ (in Section A.4) causes bracketed expressions and statements to be introduced as the program is transformed. It will become clear that the operational semantics does not arbitrarily introduce brackets. This precludes the need for brackets in all positions within a statement.

A.2 Static semantics of $\textsc{Rx}^2$

The static sementics of $\textsc{Rx}^2$ is given in terms of a typing judgment of the form

$$\Omega \vdash \mathsf{S}$$

This judgment is identical to the judgments $\Omega \vdash S$ presented in Section 3.4, except in its handling of bracket expressions and statements. A minor change in the (T-TR) rule is discussed below.

The purpose of $\textsc{Rx}^2$ is to represent pairs of executions that differ only in the high-security component of their configurations. The objective is to show that the effects of each execution that are observable by a low-security observer are identical. The semantics of the language are therefore given with respect to an observation level R embodied as a set of roles. To make the presentation of the rules simpler, we classify roles according into three classes according to the observation level, namely $\rho_L(R, \Pi), \rho_M(R, \Pi)$ and $\rho_H(R, \Pi)$. Such a simplification is standard — for instance, in [16] Pottier and Simonet make the assumption of a label hierarchy, with $L \sqsubseteq H$.

Role defintions of roles in $\rho_L(R, \Pi)$ are observable as are memory locations that are labeled with roles in $\rho_L(R, \Pi)$. We abbreviate $\rho_L(R, \Pi)$ as ρ_L where R and Π are clear from the context. One can think of ρ_L as being low-security roles. Formally,

$$\rho_L(R, \Pi) \equiv \{\rho \mid \exists \rho_R \in R.\Pi \vdash C_\Pi(\rho) \sqsubseteq \rho_R \ \wedge \ \Pi \vdash \rho \sqsubseteq \rho_R\}$$

The set $\rho_M(R, \Pi)$ consists of those roles ρ for which the definition of ρ is observable, but memory locations labeled ρ are not observable. We abbreviate this set ρ_M where the context permits. Formally,

$$\rho_M(R, \Pi) \equiv \{\rho \mid \begin{array}{l} \exists \rho_R \in R.\Pi \vdash C_\Pi(\rho) \sqsubseteq \rho_R \wedge \\ \forall \rho_R \in R.\Pi \vdash \rho \not\sqsubseteq \rho_R \end{array} \}$$

Finally, the set $\rho_H(R, \Pi)$ contains those roles ρ for which neither the definition of ρ nor memory locations labeled ρ are observable. We abbreviate $\rho_H(R, \Pi)$ as ρ_H where convenient. One can think of ρ_H as being high-security roles. Formally,

$$\rho_H(R, \Pi) \equiv \{\rho \mid \forall \rho_R \in R.\Pi \vdash C_\Pi(\rho) \not\sqsubseteq \rho_R \ \wedge \ \Pi \vdash \rho \not\sqsubseteq \rho_R\}$$

In the formal definitions of $\rho_L(R, \Pi)$ etc. above the policy Π is understood to refer to the runtime policy under which the program executes, while R is the fixed observation level of the low-security observer. Even though the runtime policy is allowed to change, our definition of noninterference allows us to treat the partitioning of the roles into these three sets as unchanging. This is achieved by means of the relation $dclasR, \Pi_1, \Pi_2$ introduced in Figure 6 and will be discussed further in Section A.3.

Although all three sets, ρ_L, ρ_M, ρ_H are not needed for the static semantics, the proof technique we use relies on the the definition of these three sets in the operational semantics for Rx. This usage should become clear in Section A.4.

The typing judgments for $\textsc{Rx}^2$ are identical to those presented for Rx. The additional rules are (T-EBR) and (T-SBR) which handle bracket expressions and statements respectively. In both cases, the rules require that the components of a bracket be checked in a context where the pc is strengthened to be at a high-security level. Since the bracketed terms represent the divergent parts of the two executions, this condition forces divergent parts of the computation to not contain any effects observable to the low-security observer.

The (T-TR) rule given here differs slightly from the one given for Rx. In this report, we consider transaction statements $\mathtt{trans}_\Phi\ S$ to be labeled with a program counter label as well as a set of invariants. This permits a simplification of the operational semantics of $\textsc{Rx}^2$ which relies on the program counter annotation on transactions. This annotation serves no purpose in the semantics of Rx and was therefore omitted in our presentation in the prior sections. The content of the rule however remains unchanged — that is, the program counter annotation that appears on the transaction must be the same as the pc that is used to type-check the transaction.

A.3 $\textsc{Rx}^2$ Configurations and Observability

$\textsc{Rx}^2$ programs represent the execution of two Rx programs that differ only in the high-security parts of their

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

$$\Omega ::= \Gamma; \mathrm{pc}; Q; \Phi$$

$\boxed{\Omega \vdash \mathsf{E} : \ell}$

$$\Omega \vdash i : \ell \quad \text{(T-LIT)} \qquad\qquad \frac{\Omega.\Gamma(x) = \ell}{\Omega \vdash x : \ell} \quad \text{(T-VAR)}$$

$$\frac{\begin{array}{c}\Omega \vdash E_1 : \ell_1 \\ \Omega \vdash E_2 : \ell_2\end{array}}{\Omega \vdash \mathsf{E}_1 \oplus \mathsf{E}_2 : \ell_1 \sqcup \ell_2} \quad \text{(T-PLUS)} \qquad \frac{\begin{array}{c}\rho \in \rho_M \cup \rho_H \\ \Omega \vdash E_1 : \ell \quad \Omega \vdash E_2 : \ell\end{array}}{\Omega \vdash \langle E_1 \parallel E_2 \rangle : \ell \sqcup \rho} \quad \text{(T-EBR)}$$

$\boxed{\Omega \vdash \mathsf{S}}$

$$\Omega \vdash \texttt{skip} \quad \text{(T-SKIP)} \qquad \frac{\begin{array}{c}\Omega \vdash \mathsf{S} \\ \Omega \vdash S\end{array}}{\Omega \vdash \mathsf{S}; S} \quad \text{(T-SEQ)} \qquad \frac{\begin{array}{cc}\Omega \vdash \mathsf{E} : \ell_1 & \Omega \vdash x : \ell \\ \Omega.Q \vdash \ell_1 \sqsubseteq \ell & \Omega.Q \vdash \Omega.\mathrm{pc} \sqsubseteq \ell\end{array}}{\Omega \vdash x := \mathsf{E}} \quad \text{(T-ASN)}$$

$$\frac{\begin{array}{cc}\Omega \vdash \mathsf{E} : \ell' & \mathrm{pc}' = \Omega.\mathrm{pc} \sqcup \ell' \\ \Omega[\mathrm{pc} = \mathrm{pc}'] \vdash S_1 & \Omega[\mathrm{pc} = \mathrm{pc}'] \vdash S_2\end{array}}{\Omega \vdash \texttt{if}\ (\mathsf{E})\ S_1\ S_2} \quad \text{(T-IFE)} \qquad \frac{\begin{array}{c}\Omega \vdash E : \ell' \quad \mathrm{pc}' = \Omega.\mathrm{pc} \sqcup \ell' \\ \Omega[\mathrm{pc} = \mathrm{pc}'] \vdash S\end{array}}{\Omega \vdash \texttt{while}\ (E)\ S} \quad \text{(T-WHL)}$$

$$\frac{\begin{array}{cc}\mathrm{pc}' = \mathrm{pc} \sqcup lab(q) & q \in \Phi.Q \\ \Gamma; \mathrm{pc}'; Q \cup \{q\}; \Phi \vdash S_1 & \Gamma; \mathrm{pc}'; Q; \Phi \vdash S_2\end{array}}{\Gamma; \mathrm{pc}; Q; \Phi \vdash \texttt{if}\ (q)\ S_1\ S_2} \quad \text{(T-IFQ)} \qquad \frac{\begin{array}{c}\Phi.\mathrm{pc} = \mathrm{pc} \\ \Gamma; \mathrm{pc}; \emptyset; \Phi \vdash \mathsf{S}\end{array}}{\Gamma; \mathrm{pc}; \emptyset; . \vdash \texttt{trans}_\Phi\ \mathsf{S}} \quad \text{(T-TR)}$$

$$\frac{\begin{array}{c}Q \vdash \mathrm{pc} \sqsubseteq \mathrm{pc}' \\ Q \vdash \mathrm{pc} \sqsubseteq lab(\Delta) \quad Q \vdash lab(Q') \sqsubseteq \mathrm{pc}'\end{array}}{\Gamma; \mathrm{pc}; Q; (\mathrm{pc}', Q') \vdash \texttt{update}\ \Delta} \quad \text{(T-UP)} \qquad \frac{\begin{array}{cc}\rho \in \rho_M \cup \rho_H & \mathrm{pc}' = \Omega.\mathrm{pc} \sqcup \rho \\ \Omega[\mathrm{pc} = \mathrm{pc}'] \vdash S_1 & \Omega[\mathrm{pc} = \mathrm{pc}'] \vdash S_2\end{array}}{\Omega \vdash \langle S_1 \parallel S_2 \rangle} \quad \text{(T-SBR)}$$

Figure 7. Static Semantics of RX^2

configurations. As such, a configuration of an RX^2 program must represent a pair of RX configurations. In this section, we define the form of $\mathcal{C}$, the context in which an RX^2 program executes. We also extend the definition of trace observability given in Figure 6 to cover pairs of configurations.

The definition of $\mathcal{C}$ the configuration of an RX^2 program is shown in the top part of Figure 8. It extends the definition of $\mathcal{E}$, the configuration of an RX program given in Section 3.3.

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 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 $(\mathcal{E}_1, \mathcal{E}_2, \alpha)|_R$ 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 $C_\Pi(\rho)$, which must satisfy the condition described in Section 3.5. However, if a policy update results in a declassification then the trace is truncated (the first clause of the $(\mathcal{E}_1, \mathcal{E}_2, \alpha)|_R$ definition of the figure).

Given the definitions of ρ_L, ρ_M and ρ_H from Section A.2 we can now examine in detail the definition of $dclas(R, \Pi_1, \Pi_2)$ given in Figure 6 and repeated here:

$$dclas(R, \Pi_1, \Pi_2) \equiv \begin{array}{l}(\rho_L(R, \Pi_2) \cap \rho_M(R, \Pi_1)) \cup \\ (\rho_L(R, \Pi_2) \cap \rho_H(R, \Pi_1)) \cup \\ (\rho_M(R, \Pi_2) \cap \rho_H(R, \Pi_1))\end{array}$$

Let us examine the last term $((\rho_M(R, \Pi_2) \cap \rho_H(R, \Pi_1)))$ first. Recall that roles ρ in ρ_M are those for which the definition of ρ is observable, but a memory location x, with

Rx^2 **Configuration** :

policy stmt.	s	$::=$	$s \mid \langle s_1 \parallel s_2\rangle$
policy	Π	$::=$	$\{\mathsf{s}_1, \ldots, \mathsf{s}_n\}$
values	v	$::=$	$v \mid \langle v_1 \parallel v_2\rangle$
memory	M	$::=$	$\{(x_1, \mathsf{v}_1), \ldots, (x_n, \mathsf{v}_n)\}$
one snapshot	Ψ^*	$::=$	$. \mid (\mathsf{M}, \mathsf{S})$
snapshot	Ψ	$::=$	$\Psi^* \mid \langle \Psi^*_1 \parallel \Psi^*_2\rangle$
configuration	$\mathcal{C}$	$::=$	$\Pi; \mathsf{M}; \Psi$

Values $\lfloor \mathsf{v} \rfloor_i$

$\lfloor \mathsf{v} \rfloor_0 \equiv \mathsf{v} \qquad \lfloor v \rfloor_i \equiv v \qquad \lfloor \langle v_1 \parallel v_2\rangle \rfloor_i \equiv v_i$

Policy $\lfloor \Pi \rfloor_i$

$\lfloor \Pi \rfloor_0 \equiv \Pi \qquad \lfloor \emptyset \rfloor_i = \emptyset \qquad \lfloor \{s\} \cup \Pi \rfloor_i = \{s\} \cup \lfloor \Pi \rfloor_i$

$\lfloor \{\langle s_1 \parallel s_2\rangle\} \cup \Pi \rfloor_i = \{s_i\} \cup \lfloor \Pi \rfloor_i$

Memory $\lfloor \mathsf{M} \rfloor_i$

$\lfloor \mathsf{M} \rfloor_0 = \mathsf{M} \qquad \lfloor \emptyset \rfloor_j = \emptyset \qquad \lfloor \{(x, i)\} \cup \mathsf{M} \rfloor_j = \{(x, i)\} \cup \lfloor \mathsf{M} \rfloor_j$

$\lfloor \{(x, \langle i_1 \parallel i_2\rangle)\} \cup \mathsf{M} \rfloor_j = \{(x, i_j)\} \cup \lfloor \mathsf{M} \rfloor_j$

Transaction snapshot $\lfloor \Psi \rfloor_i$

$\lfloor \Psi \rfloor_0 \equiv \Psi \qquad \lfloor \Psi^* \rfloor_i \equiv \Psi^* \qquad \lfloor \langle \Psi^*_1 \parallel \Psi^*_2\rangle \rfloor_i \equiv \Psi^*_i$

Configuration $\lfloor \mathcal{C} \rfloor_i$

$\lfloor (\Pi, \mathsf{M}, \Psi) \rfloor_i \equiv (\lfloor \Pi \rfloor_i, \lfloor \mathsf{M} \rfloor_i, \lfloor \Psi \rfloor_i)$

Figure 8. Rx^2 **configurations and projections.**

Policy $\Pi|_R$ **:**

$\Pi|_R \equiv \lfloor \Pi \rfloor_1|_R \oplus \lfloor \Pi \rfloor_2|_R \qquad$ where

$$\Pi_1 \oplus \Pi_2 \equiv \bigcup_{s \in \Pi_1 \cap \Pi_2} s \;\cup \bigcup_{s \in \Pi_1 \setminus \Pi_2} \langle s \parallel \cdot\rangle \;\cup \bigcup_{s \in \Pi_2 \setminus \Pi_1} \langle \cdot \parallel s\rangle$$

Memory $\mathsf{M}|_{R,\Pi}$ **:**

$\mathsf{M}|_{R,\Pi} \equiv \lfloor \mathsf{M} \rfloor_1|_{R,\lfloor \Pi \rfloor_1} \oplus \lfloor \mathsf{M} \rfloor_2|_{R,\lfloor \Pi \rfloor_2} \qquad$ where

$$M_1 \oplus M_2 \equiv \bigcup \begin{cases} \{x\} & M_1(x) = M_2(x) \\ \langle M_1(x) \parallel M_2(x)\rangle & M_1(x) \neq M_2(x) \\ \langle x \parallel \cdot\rangle & x \in \mathit{dom}(M_1) \setminus \mathit{dom}(M_2) \\ \langle \cdot \parallel x\rangle & x \in \mathit{dom}(M_2) \setminus \mathit{dom}(M_1) \end{cases}$$

Transaction snapshot $\Psi|_{R,\Pi}$**:**

$(\mathsf{M}, \mathsf{S})|_{R,\Pi} \equiv (\mathsf{M}|_{R,\Pi}, \mathsf{S})$

$\langle \Psi^*_1 \parallel \Psi^*_2\rangle|_{R,\Pi} \equiv \langle \Psi^*_1|_{R,\Pi} \parallel \Psi^*_2|_{R,\Pi}\rangle$

Configuration $\mathcal{C}|_R$**:**

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

Figure 9. Observability of Rx^2 **configurations.**

$\Gamma(x) = \rho$ is not observable. Neither the role definitions nor locations labeled with roles in ρ_H are observable. The intersection of the two sets of roles is non-empty when some role ρ for which neither policy nor memory was observable under the policy Π_1, has its definition only revealed under Π_2. Thus, this term captures declassifications that reveal information about policy only. Similarly, the term $\rho_L(R, \Pi_2) \cap \rho_M(R, \Pi_1)$ represents declassification of memory only — i.e. a role ρ for which only the definition was observable under Π_1, under Π_2 has its corresponding memory locations revealed also. Finally, $\rho_L(R, \Pi_2) \cap \rho_H(R, \Pi_1)$ represents declassifications that reveal both the contents of memory as well as policy. Obviously, $\mathit{dclas}(R, \Pi_1, \Pi_2)$ is parameterized with respect to an observation level R. Thus, declassifications that occur within a set ρ_M or within ρ_H do not reveal any information to the R-observer.

We say that an execution $\langle \mathcal{E}, S\rangle$ contains a declassification when $\mathcal{E}, S \longrightarrow^* \mathcal{E}_1, S_1 \longrightarrow \mathcal{E}_2, S_2$ and $\mathit{dclas}(R, \mathcal{E}_1.\Pi, \mathcal{E}_2.\Pi) \neq \emptyset$.

In Figure 9 we define the observability for configurations of Rx^2 programs. Recall that we are using Rx^2 to reason about pairs of program executions that are identical in the parts of their configuration observable to a low-security observer. We proceed by defined a well-formedness condition for $\mathcal{C}$ that requires precisely this condition — the low-observable parts of $\mathcal{C}$ should be identical for both executions. For this purpose, Figure 9 extends the definitions of observability in Figure 6 to include pairs of configurations.

The observability of $\mathcal{C}$ relies on the operator $\lfloor \cdot \rfloor_i$ which restricts the configuration component to the parts relevant to the ith execution. For instance $\lfloor \Pi \rfloor_i$ contains only the policy statements in Π that are relevant to the ith execution, where $i \in \{1, 2\}$; similarly $\lfloor \mathsf{M} \rfloor_i$ refers only to the contents of memory that are relevant to the ith execution. To define the observability of Π and M we use the operator $\oplus$ to combine the results of observability relation in each execution.

We extend Definition 8 to obtain a well-formedness condition for Rx^2 configurations.

Definition 11 (Well-formed $\mathcal{C}$)**.** *A configuration* $\mathcal{C} = (\Pi, \mathsf{M}, \Psi)$ *is well-formed with respect to a context* Ω *and a set of roles* R*, denoted* $\Omega \models_R \mathcal{C}$*, if and only if all of the following are true:*

$$\begin{array}{ll}
\mathit{dom}(\mathsf{M}) \equiv \mathit{dom}(\Omega.\Gamma) \;\wedge\; \forall x.\Omega \vdash \mathsf{M}(x) : \Omega.\Gamma(x) & (1) \\
\forall q \in \Omega.Q\ .\ \lfloor \Pi \rfloor_1 \vdash q \;\wedge\; \lfloor \Pi \rfloor_2 \vdash q & (2) \\
\textit{if for } i \in \{1,2\}\ \lfloor \Psi \rfloor_i = (\mathsf{M}', \mathtt{trans}_\Phi\ \mathsf{S}')\ \textit{then} & \\
\quad \Omega \vdash \mathtt{trans}_\Phi\ \mathsf{S}' & (3.1) \\
\quad \mathit{dom}(\mathsf{M}') = \mathit{dom}(\Omega.\Gamma) \;\wedge\; \forall x.\Omega \vdash \mathsf{M}'(x) : \Omega.\Gamma(x) & (3.2) \\
\quad \forall x. \lfloor \mathsf{M}(x) \rfloor_i \neq \lfloor \mathsf{M}'(x) \rfloor_i \Rightarrow & \\
\qquad \exists Q' \subseteq \Phi.Q.\forall q \in Q'. & \\
\qquad\quad \lfloor \Pi \rfloor_i \vdash q \;\wedge\; Q' \vdash \Omega.\mathrm{pc} \sqsubseteq \Omega.\Gamma(x) & (3.3) \\
\lfloor \mathcal{C} \rfloor_1|_R \equiv \lfloor \mathcal{C} \rfloor_2|_R & (4)
\end{array}$$

It should be clear that Definition 11 is closely related to Definition 8. The most significant difference is the addition of clause (4) which requires that the pair of Rx configurations embedded in $\mathcal{C}$ are observationally equivalent.

Clauses (1) and (3.2) have been strengthened to ensure that values stored in memory can be typed according to the labels assigned to the location. The need for this can be seen by examining the type-rules (T-LIT) and (T-EBR). (T-LIT) is used to type non-bracket values v, and clearly these values can be assigned any label whatsoever. In particular, a value v stored in location x can always be given the label $\Gamma(x)$. However, a pair of values $\langle v_1 \parallel v_2 \rangle$ must be typed using the rule (T-EBR), which forces the label assigned to this expression to be in $\rho_M \cup \rho_H$. The intuition here is that memory locations whose contents differ in each of the pair of configurations must be high-security locations. While clause (4) ensures observational equivalence, clauses (1) and (3.2) ensure that the Rx^2 representation of a pair of configurations does not gratuitously introduce bracket expressions of the form $\langle v \parallel v \rangle$ into low-security locations in memory.

The final difference is a strengthening of clause (3.3). Recall the form of clause (3.3) in Definition 8, reproduced here:

$$\forall x. M(x) \neq M'(x) \Rightarrow \Pi \vdash \Omega.\mathrm{pc} \sqsubseteq \Omega.\Gamma(x)$$

This clause enforces the condition that all memory effects of a transaction are bounded below by the pc label that was used to check the transaction. This condition allows us to show that the effects to memory that take place when a transaction is rolled-back does not leak any information. This requirement is strengthened in Definition 11 so that, not only are the effects bounded below by the pc, but that this is provable using the static judgment $Q \vdash \ell_1 \sqsubseteq \ell_2$. From the soundness of the static label ordering judgment (Lemma 9), we that clause (3.3) in Definition 11 is a strictly stronger condition than that shown in Definition 11. This stronger form permits a more direct proof of subject reduction. We show that this stronger form of the clause is preserved during execution.

A.4 Operational Semantics of Rx^2

The operational semantics of Rx^2 are presented in Figure 4. The judgment $\mathsf{S} \ /_i\ \mathcal{C} \longrightarrow \mathsf{S}' \ /_i\ \mathcal{C}'$ defines a set of rewriting rules for the program S executing in a context $\mathcal{C}$. The index $i \in \{0, 1, 2\}$ indicates the execution in which the step occurs. When $i = 0$ the the step of computation occurs in both executions; when $i = 1$ the step occurs in the left-side of the bracken; and when $i = 2$ the step is taken in the right side of the bracket.

The semantics mimics the behavior of the semantics of Rx on two configurations. In doing so, care must be taken to ensure that the effects of one execution are limited to the configuration of that execution only. For this reason, we define the helper functions updloc_i, updpsi_i, updpi_i, $\mathit{rollback}_i$ to ensure that the result of effect-ful steps of computation are confined to the appropriate part of the configuration. Similarly, operations that read parts of the configuration must do so from the part appropriate to the execution. For instance, in the (E-VAR) rule, the contents of the memory location $\mathsf{M}(x)$ is projected using $\lfloor \cdot \rfloor_i$ before it is read.

The rules in Figure A.4 augment those in Figure 4. The lifting rules have no computational content. They only serve to allow computation to proceed in cases where the steps taken by the two executions are not identical. For instance, the (IFV) rule of Figure 4 only applies to if-statements where an Rx value v appears in the guard. The rule (L-IFE) applies in the case where a bracketed value v appears in the guard. In this case, the brackets around the expression in the guard are "lifted" to include the if-statement itself. As a result, each subexecution can now take a step. Note that if the intention of the Rx^2 semantics is only to simulate a pair of Rx executions, then it is always permissible to introduce lift brackets. Designing the semantics such that brackets are introduced only where essential is key to obtaining a proof.

The following lemma states that these semantics are sound.

Lemma 12. *Given an* Rx *program S and two configurations $\mathcal{E}_1$ and $\mathcal{E}_2$ such that $\mathcal{E}_1 |_R = \mathcal{E}_2 |_R$, then there exists an Rx^2 configuration $\mathcal{C}$ such that $\lfloor \mathcal{C} \rfloor_i = \mathcal{E}_i$. Furthermore, if there are finite declassification free sub-executions $(\mathcal{E}_1, S) \longrightarrow^* (\mathcal{E}_1', S')$ and $(\mathcal{E}_2, S) \longrightarrow^* (\mathcal{E}_2', S'')$ then there there exists a finite sub-execution $S \ /_0\ \mathcal{C} \longrightarrow^* \mathsf{S} \ /_0\ \mathcal{C}'$ such that $\lfloor \mathcal{C}' \rfloor_i = \mathcal{E}_i'$ for $i \in \{1, 2\}$, and vice-versa.*

The proof of this statement is, for the most part, entirely straightforward. Two rules in Figure 4 need special care — (IFQ-0) and (TR0). Both these rules inspect the roles that appear in the program text, either in queries or in the annotations that are on trasnactions. Since the semantics of Rx^2 is only for the purposes of a proof, this is reasonable. The semantics of Rx contain no such checks. We describe here the purpose of these rules, and then show how an alternative formulation of the static semantics of Rx^2 could eliminate these rules.

The soundness of the premise of (IFQ-0) requires that a query q that refers only to observable roles ($\mathrm{roles}(q) \subseteq \rho_L \cup \rho_M$) has the same result when evaluated under policies that are observationally equivalent. Given that this condition holds, (IFQ-0) states that since the result of the query is the same, both executions take the same branch of the if-statement and therefore no brackets need to be introduced. In contrast, (L-IFQ) is a lifting rule that states that if a query contains a role that is not observable then, since the result of the query can be different in each execution, a bracketed statement is introduced. Lemma 13 states that the conditions required of a choice of metapolicy are sufficient to guarantee this requirement. Later, we also state and prove Lemma 20 for a specific choice of metapolicy based on the delegation relation.

$$\frac{\lfloor \mathcal{C}.\mathsf{M}(x)\rfloor_i = \mathsf{v}}{x \ /_i\ \mathcal{C} \longrightarrow \mathsf{v} \ /_i\ \mathcal{C}}\ \text{(E-VAR)} \qquad \frac{\mathsf{E}_1 \ /_i\ \mathcal{C} \longrightarrow \mathsf{E}_1' \ /_i\ \mathcal{C}}{\mathsf{E}_1 \oplus \mathsf{E}_2 \ /_i\ \mathcal{C} \longrightarrow \mathsf{E}_1' \oplus \mathsf{E}_2 \ /_i\ \mathcal{C}}\ \text{(E-ADL)} \qquad \frac{\mathsf{E}_2 \ /_i\ \mathcal{C} \longrightarrow \mathsf{E}_2' \ /_i\ \mathcal{C}}{\mathsf{v} \oplus \mathsf{E}_2 \ /_i\ \mathcal{C} \longrightarrow \mathsf{v} \oplus \mathsf{E}_2' \ /_i\ \mathcal{C}}\ \text{(E-ADR)}$$

$$\frac{v = v_1 [\![\oplus]\!] v_2}{v_1 \oplus v_2 \ /_i\ \mathcal{C} \longrightarrow v \ /_i\ \mathcal{C}}\ \text{(E-ADV)} \qquad \frac{\mathsf{S} \ /_i\ \mathcal{C} \longrightarrow \mathsf{S}' \ /_i\ \mathcal{C}'}{\mathsf{S}; S \ /_i\ \mathcal{C} \longrightarrow \mathsf{S}'; S \ /_i\ \mathcal{C}'}\ \text{(E-SEQ)} \qquad \mathtt{skip}; S \ /_i\ \mathcal{C} \longrightarrow S \ /_i\ \mathcal{C}\ \text{(E-SKP)}$$

$$\frac{\mathsf{E} \ /_i\ \mathcal{C} \longrightarrow \mathsf{E}' \ /_i\ \mathcal{C}}{x := \mathsf{E} \ /_i\ \mathcal{C} \longrightarrow x := \mathsf{E}' \ /_i\ \mathcal{C}}\ \text{(E-ASE)} \qquad \frac{\mathcal{C}' = \mathcal{C}[\mathsf{M} = \mathit{updloc}_i(x, \mathsf{v}, \mathcal{C}.\mathsf{M})]}{x := \mathsf{v} \ /_i\ \mathcal{C} \longrightarrow \mathtt{skip} \ /_i\ \mathcal{C}'}\ \text{(E-ASV)}$$

$$\frac{E \ /_i\ \mathcal{C} \longrightarrow \mathsf{E} \ /_i\ \mathcal{C}}{\begin{array}{c}\mathtt{if}\ (E)\ S_1\ S_2 \ /_i\ \mathcal{C} \longrightarrow \\ \mathtt{if}\ (\mathsf{E})\ S_1\ S_2 \ /_i\ \mathcal{C}\end{array}}\ \text{(E-IFE)} \qquad \frac{\begin{array}{c}(v \neq 0 \Rightarrow j = 1) \\ (v = 0 \Rightarrow j = 2)\end{array}}{\begin{array}{c}\mathtt{if}\ (v)\ S_1\ S_2 \ /_i\ \mathcal{C} \longrightarrow \\ S_j \ /_i\ \mathcal{C}\end{array}}\ \text{(E-IFV)} \qquad \begin{array}{c}\mathtt{while}\ (E)\ S \ /_i\ \mathcal{C} \longrightarrow \\ \mathtt{if}\ (E)\ \{\ S; \mathtt{while}\ (E)\ S\ \}\ \mathtt{skip} \ /_i\ \mathcal{C}\end{array}\ \text{(E-WHL)}$$

$$\frac{\begin{array}{c}\mathrm{roles}(q) \subseteq \rho_L \cup \rho_M \\ \mathcal{C}.\Pi \vdash q \Rightarrow j = 1 \quad \mathcal{C}.\Pi \not\vdash q \Rightarrow j = 2\end{array}}{\mathtt{if}\ (q)\ S_1\ S_2 \ /_0\ \mathcal{C} \longrightarrow S_j \ /_0\ \mathcal{C}}\ \text{(E-IFQ-0)} \qquad \frac{\begin{array}{c}i \in \{1, 2\} \\ \lfloor \mathcal{C}.\Pi \rfloor_i \vdash q \Rightarrow j = 1 \quad \lfloor \mathcal{C}.\Pi \rfloor_i \not\vdash q \Rightarrow j = 2\end{array}}{\mathtt{if}\ (q)\ S_1\ S_2 \ /_i\ \mathcal{C} \longrightarrow S_j \ /_i\ \mathcal{C}}\ \text{(E-IFQ-i)}$$

$$\frac{\begin{array}{c}\exists \rho \in \rho_L . \lfloor \mathcal{C}.\Pi \rfloor_i \vdash \ell \sqsubseteq \rho \\ \mathcal{C}.\Psi = \cdot \quad \Psi^* = (\mathcal{C}.\mathsf{M}, \mathtt{trans}_{(\ell,Q)}\ S)\end{array}}{\mathtt{trans}_{(\ell,Q)}\ S \ /_0\ \mathcal{C} \longrightarrow \mathtt{trans}_{(\ell,Q)}\ S \ /_0\ \mathcal{C}[\Psi = \Psi^*]}\ \text{(E-TR1-0)} \qquad \frac{\begin{array}{c}i \in \{1, 2\} \qquad \lfloor \mathcal{C}.\Psi \rfloor_i = \cdot \\ \Psi' = \mathit{updpsi}_i(\mathcal{C}.\Psi, (\mathsf{M}, \mathtt{trans}_\Phi\ S))\end{array}}{\mathtt{trans}_\Phi\ S \ /_i\ \mathcal{C} \longrightarrow \mathtt{trans}_\Phi\ S \ /_i\ \mathcal{C}[\Psi = \Psi']}\ \text{(E-TR1-i)}$$

$$\frac{\lfloor \mathcal{C}.\Psi \rfloor_i \neq \cdot \quad \mathsf{S} \ /_i\ \mathcal{C} \longrightarrow \mathsf{S}' \ /_i\ \mathcal{C}'}{\mathtt{trans}_\Phi\ \mathsf{S} \ /_i\ \mathcal{C} \longrightarrow \mathtt{trans}_\Phi\ \mathsf{S}' \ /_i\ \mathcal{C}'}\ \text{(E-TR2)} \qquad \frac{\lfloor \mathcal{C}.\Psi \rfloor_i \neq \cdot \quad \Psi' = \mathit{updpsi}_i(\mathcal{C}.\Psi, \cdot)}{\mathtt{trans}_\Phi\ \mathtt{skip} \ /_i\ \mathcal{C} \longrightarrow \mathtt{skip} \ /_i\ \mathcal{C}[\Psi = \Psi']}\ \text{(E-TR3)}$$

$$\frac{\begin{array}{cc}\Pi' = \mathit{updpi}_i(\mathcal{C}.\Pi, \Delta) & \lfloor \mathcal{C}.\Psi \rfloor_i = (\mathsf{M}', \mathtt{trans}_{(\ell,Q)}\ S) \\ i = 0 \Rightarrow j = 1 & i \neq 0 \Rightarrow i = j \\ \mathit{dclas}^2(R, \mathcal{C}.\Pi, \Pi') = \emptyset & \forall q \in Q . (\lfloor \Pi \rfloor_j \vdash q) \Leftrightarrow (\lfloor \Pi' \rfloor_j \vdash q)\end{array}}{\mathtt{update}\ \Delta \ /_i\ \mathcal{C} \longrightarrow \mathtt{skip} \ /_i\ \mathcal{C}[\Pi = \Pi']}\ \text{(E-UP)}$$

$$\frac{\begin{array}{c}\Pi' = \mathit{updpi}_i(\mathcal{C}.\Pi, \Delta) \qquad \lfloor \mathcal{C}.\Psi \rfloor_i = (\mathsf{M}, \mathtt{trans}_{(\ell,Q)}\ S) \\ i = 0 \Rightarrow j = 1 \quad i \neq 0 \Rightarrow i = j \quad \mathsf{M}' = \mathit{rollback}_i(\mathcal{C}.\mathsf{M}, \mathsf{M}) \\ \mathit{dclas}^2(R, \mathcal{C}.\Pi, \Pi') = \emptyset \quad \exists q \in Q . (\lfloor \Pi \rfloor_j \vdash q) \not\Leftrightarrow (\lfloor \Pi' \rfloor_j \vdash q)\end{array}}{\mathtt{update}\ \Delta \ /_i\ \mathcal{C} \rightsquigarrow \mathtt{trans}_{(\ell,Q)}\ S \ /_i\ \mathcal{C}[\mathsf{M} = \mathsf{M}'][\Pi = \Pi']}\ \text{(R-UP)}$$

$$\frac{\mathsf{S} \ /_i\ \mathcal{C} \rightsquigarrow \mathsf{S}' \ /_i\ \mathcal{C}'}{\mathsf{S}; S \ /_i\ \mathcal{C} \rightsquigarrow \mathsf{S}' \ /_i\ \mathcal{C}'}\ \text{(R-SEQ)} \qquad \frac{\mathsf{S} \ /_i\ \mathcal{C} \rightsquigarrow \mathsf{S}' \ /_i\ \mathcal{C}'}{\mathtt{trans}_\Phi\ \mathsf{S} \ /_i\ \mathcal{C} \longrightarrow \mathsf{S}' \ /_i\ \mathcal{C}'}\ \text{(E-TR4)} \qquad \frac{\begin{array}{c}S_i \ /_i\ \mathcal{C} \longrightarrow S_i' \ /_i\ \mathcal{C}' \\ \{i, j\} = \{1, 2\} \quad S_j' = S_j\end{array}}{\langle S_1 \parallel S_2 \rangle \ /_0\ \mathcal{C} \longrightarrow \langle S_1' \parallel S_2' \rangle \ /_i\ \mathcal{C}'}\ \text{(E-BRK)}$$

where

$$\Pi \vdash q \iff \lfloor \Pi \rfloor_1 \vdash q \ \wedge\ \lfloor \Pi \rfloor_2 \vdash q \qquad \mathit{dclas}^2(R, \Pi_1, \Pi_2) \equiv \mathit{dclas}(R, \lfloor \Pi_1 \rfloor_1, \lfloor \Pi_2 \rfloor_1) \cup \mathit{dclas}(R, \lfloor \Pi_1 \rfloor_2, \lfloor \Pi_2 \rfloor_2)$$

$$\mathrm{roles}(L_1 \sqsubseteq L_2) = \mathrm{roles}(L_1) \cup \mathrm{roles}(L_2) \qquad \mathrm{roles}(\rho) = \{\rho\} \qquad \mathrm{roles}(C_\Pi(\rho)) = \{\rho\} \qquad \mathrm{roles}(I_\Pi(\rho)) = \{\rho\}$$

$$\Pi \pm^i \Delta \equiv (\lfloor \Pi \rfloor_i \cup \{s \mid \mathtt{add}\ (s)\ \in \Delta\} \setminus \{s \mid \mathtt{del}\ (s)\ \in \Delta\})$$

$$\begin{array}{lcllcl}
\mathit{updloc}_0(x, \mathsf{v}, \mathsf{M}) & \equiv & \mathsf{M}[(x, \mathsf{v})] & \mathit{updpsi}_0(\Psi, \Psi^*) & \equiv & \Psi^* \\
\mathit{updloc}_1(x, \mathsf{v}, \mathsf{M}) & \equiv & \mathsf{M}[(x, \langle \mathsf{v} \parallel \lfloor \mathsf{M}(x) \rfloor_2 \rangle)] & \mathit{updpsi}_1(\Psi, \Psi^*) & \equiv & \langle \Psi^* \parallel \lfloor \Psi \rfloor_2 \rangle \\
\mathit{updloc}_2(x, \mathsf{v}, \mathsf{M}) & \equiv & \mathsf{M}[(x, \langle \lfloor \mathsf{M}(x) \rfloor_1 \parallel \mathsf{v} \rangle)] & \mathit{updpsi}_2(\Psi, \Psi^*) & \equiv & \langle \lfloor \Psi \rfloor_1 \parallel \Psi^* \rangle \\
\mathit{updpi}_0(\Pi, \Delta) & \equiv & (\Pi \pm^1 \Delta) \oplus (\Pi \pm^2 \Delta) & \mathit{rollback}_0(\mathsf{M}, \mathsf{M}') & \equiv & \mathsf{M}' \\
\mathit{updpi}_1(\Pi, \Delta) & \equiv & (\Pi \pm^1 \Delta) \oplus \lfloor \Pi \rfloor_2 & \mathit{rollback}_1(\mathsf{M}, \mathsf{M}') & \equiv & \lfloor \mathsf{M}' \rfloor_1 \oplus \lfloor \mathsf{M} \rfloor_2 \\
\mathit{updpi}_2(\Pi, \Delta) & \equiv & \lfloor \Pi \rfloor_1 \oplus (\Pi \pm^2 \Delta) & \mathit{rollback}_2(\mathsf{M}, \mathsf{M}') & \equiv & \lfloor \mathsf{M} \rfloor_1 \oplus \lfloor \mathsf{M}' \rfloor_2
\end{array}$$

Figure 10. Operational semantics of $\mathrm{R{\small X}}^2$ excluding lifting rules.

The rules (TR-0) and (L-TR) are similar to (IFQ-0) and (L-IFQ) in that, like (IFQ-0), (TR-0) prevents the introduction of brackets in the case where the transaction mentions only low-security policy queries and updates — the condition that the label ℓ mentioned on the transaction statement is "low-security" ($\ell \sqsubseteq \rho \quad \rho \in \rho_L$) ensures that all policy queries and update statement refer only to roles in $\rho_L \cup \rho_M$. Lemma 14 guarantees that in such a case, the effect of every non-declassifying update statement is the same in both executions. Thus, no brackets need to be introduced. The lifting rule (L-TR), like (L-IFQ), introduces a brackets if this condition is not met since the result of updates in the two executions need not be identical.

An alternative approach to introducing these role inspections in the operational semantics of $\textsc{Rx}^2$ would have been to change the form of the static semantics. We could have defined the static semantics of $\textsc{Rx}^2$ to be a compilation process where the input $\textsc{Rx}$ program is transformed to introduce bracket expressions where necessary. The form of such a judgment, and a rule in this judgment that handles policy query statements is given below:

$$\boxed{\Omega \vdash S :: \mathsf{S}}$$

$$\frac{\mathrm{roles}(q) \not\subseteq \rho_L \cup \rho_M \qquad \ldots}{\Omega \vdash \texttt{if}\ (q)\ S_1\ S_2 :: \texttt{if}\ (\langle q \parallel q\rangle)\ S_1\ S_2}$$

Given such a compilation process, we could eliminate (E-IFQ-0) and replace it with a lifting rule identical to (L-IFE). Although this method is equivalent, we chose not to take this approach since it requires a substantial conceptual change in the static semantics.

$$\langle \texttt{skip} \parallel \texttt{skip}\rangle; S \ /_0\ \mathcal{C} \longrightarrow S \ /_0\ \mathcal{C} \quad \text{(L-SKIP)}$$

$$\frac{\mathsf{v} = \langle \lfloor \mathsf{v}_1 \rfloor_1 + \lfloor \mathsf{v}_2 \rfloor_1 \parallel \lfloor \mathsf{v}_1 \rfloor_2 + \lfloor \mathsf{v}_2 \rfloor_2 \rangle}{\mathsf{v}_1 + \mathsf{v}_2 \ /_0\ \mathcal{C} \longrightarrow \mathsf{v} \ /_0\ \mathcal{C}} \quad \text{(L-ADD)}$$

$$\begin{array}{c}\texttt{if}\ (\langle E_1 \parallel E_2\rangle)\ S_1\ S_2 \ /_0\ \mathcal{C} \longrightarrow \\ \langle \texttt{if}\ (E_1)\ S_1\ S_2 \parallel \texttt{if}\ (E_2)\ S_1\ S_2\rangle \ /_0\ \mathcal{C}\end{array} \quad \text{(L-IFE)}$$

$$\frac{\mathrm{roles}(q) \not\subseteq \rho_L \cup \rho_M}{\begin{array}{c}\texttt{if}\ (q)\ S_1\ S_2 \ /_0\ \mathcal{C} \longrightarrow \\ \langle \texttt{if}\ (q)\ S_1\ S_2 \parallel \texttt{if}\ (q)\ S_1\ S_2\rangle \ /_0\ \mathcal{C}\end{array}} \quad \text{(L-IFQ)}$$

$$\frac{\begin{array}{c}\mathcal{C}.\Psi = \cdot \\ \exists i \in \{1,2\}.\exists \rho \in \rho_M \cup \rho_H.\lfloor \mathcal{C}.\Pi \rfloor_i \vdash \rho \sqsubseteq \ell\end{array}}{\begin{array}{c}\texttt{trans}_{(\ell,Q)}\ S \ /_0\ \mathcal{C} \longrightarrow \\ \langle \texttt{trans}_{(\ell,Q)}\ S \parallel \texttt{trans}_{(\ell,Q)}\ S\rangle \ /_0\ \mathcal{C}\end{array}} \quad \text{(L-TR)}$$

Figure 11. Lifting rules for $\textsc{Rx}^2$.

Lemma 13. *Given a pair of policies Π_1 and Π_2, and observation level R and a metapolicy $C_\Pi(\cdot)$ such that (1) $\Pi_1|_R \equiv \Pi_2|_R$; (2) $C_\Pi(\cdot)$ satisfies the conditions (C1) and (C2) below:*

C1 $del_\Pi(\rho) = \{\rho_1, \ldots, \rho_n\} \Rightarrow \forall i.\ [\![C_\Pi(\rho)]\!]^\Pi \subseteq [\![C_\Pi(\rho_i)]\!]^\Pi$

C2 $\forall i.[\![C_\Pi(\rho)]\!]^\Pi \subseteq [\![C_\Pi(\rho_i)]\!]^\Pi$*, where $C_\Pi(\rho)$ or $I_\Pi(\rho)$ is a function of ρ_i.*

Then for any query q such that $\mathrm{roles}(q) \subseteq \rho_L(\Pi_i, R) \cup \rho_M(\Pi_i, R)$*, $\Pi_1 \vdash q \iff \Pi_2 \vdash q$.*

Proof. Queries q have the form $L_1 \sqsubseteq L_2$ where L_1 and L_2 are atomic labels of the form ρ, $C_\Pi(\rho)$ and $I_\Pi(\rho)$. It suffices to show that if $\rho \in \rho_L \cup \rho_M$ then $[\![\rho]\!]^{\Pi_1} = [\![\rho]\!]^{\Pi_2}$, $[\![C_\Pi(\rho)]\!]^{\Pi_1} = [\![C_\Pi(\rho)]\!]^{\Pi_2}$ and $[\![I_\Pi(\rho)]\!]^{\Pi_1} = [\![I_\Pi(\rho)]\!]^{\Pi_2}$.

We first consider $[\![\rho]\!]^{\Pi_1} = [\![\rho]\!]^{\Pi_2}$. Note the following condition for the observability of a role as given in Figure 6, for $i \in \{1,2\}$

$$\rho \in \mathit{Obs}(R, \Pi_i) \iff \exists \rho_R \in R.\Pi_i \vdash C_\Pi(\rho) \sqsubseteq \rho_R$$

This condition can be stated equivalently as

$$[\![\rho]\!]^{\Pi_i} \subseteq [\![C_\Pi(\rho)]\!]^{\Pi_i}$$

Now, $[\![\rho]\!]^{\Pi_i}$ is a function of $[\![\rho_1]\!]^{\Pi_i} \ldots [\![\rho_n]\!]^{\Pi_i}$, where $\{\rho_1, \ldots, \rho_n\} = del_{\Pi_i}(\rho)$. By condition (C1) we have that $\exists \rho_R \in R.[\![\subseteq]\!]^{\Pi_i}[\![C_\Pi(\rho_i)]\!]^\Pi$. Thus, each $\{\rho_1, \ldots, \rho_n\} \subseteq \mathit{Obs}(R, \Pi_1) = \mathit{Obs}(R, \Pi_2)$; or $\forall s.\mathit{roledef}(s) \in \{\rho, \rho_1, \ldots, \rho\} \Rightarrow s \in \Pi_1 \wedge s \in \Pi_2$. Hence $[\![\rho]\!]^{\Pi_1} = [\![\rho]\!]^{\Pi_2}$.

To derive $[\![C_\Pi(\rho)]\!]^{\Pi_1} = [\![C_\Pi(\rho)]\!]^{\Pi_2}$, we note in a manner similar to the previous case, that if $[\![C_\Pi(\rho)]\!]^{\Pi_i}$ depends on $\{\rho_1, \ldots, \rho_n\}$ then, by (C2), each of $\{\rho_1, \ldots, \rho_n\} \subseteq \mathit{Obs}(R, \Pi_1) = \mathit{Obs}(R, \Pi_2)$. Since the constraint on integrity are identical to confidentiality the equivalence of the interpretation of $I_\Pi(\rho)$ follows similarly. □

Lemma 14. *Let $S = \texttt{trans}_{(\ell,Q)}\ S'$ be an* $\textsc{Rx}$ *statement such that $\Omega \vdash S$ for some Ω; Given a security level R and two configurations $\mathcal{E}_1 = (\Pi_1, M_1)$ and $\mathcal{E}_2 = (\Pi_2, M_2)$ such that $\mathcal{E}_1|_R \equiv \mathcal{E}_2|_R$. If*

$$(L1) \quad \forall i \in \{1,2\}.\exists \rho \in \rho_L(R, \Pi_i).\Pi_i \vdash \ell \sqsubseteq \rho$$

then for any pair of finite executions $\langle \mathcal{E}_1, S\rangle$ and $\langle \mathcal{E}_2, S\rangle$ that are free of declassifications, the following three properties hold true.

(i) $\quad (\mathcal{E}_1, S) \longrightarrow^* (\mathcal{E}_1', \texttt{trans}_\Phi\ \texttt{update}\ \Delta; S'') \iff (\mathcal{E}_2, S) \longrightarrow^* (\mathcal{E}_2', \texttt{trans}_\Phi\ \texttt{update}\ \Delta; S'')$

(ii) $\quad (\mathcal{E}_1', \texttt{trans}_\Phi\ \texttt{update}\ \Delta; S'') \xrightarrow{\text{(E-UP)}} (\mathcal{E}_1'', \texttt{trans}_\Phi\ S'')$

$$\iff$$

$\quad (\mathcal{E}_2', \texttt{trans}_\Phi\ \texttt{update}\ \Delta; S'') \xrightarrow{\text{(E-UP)}} (\mathcal{E}_2'', \texttt{trans}_\Phi\ S'')$

(iii) $\quad \mathcal{E}_1'.\Pi|_R \equiv \mathcal{E}_2'.\Pi|_R \Rightarrow \mathcal{E}_1''.\Pi|_R \equiv \mathcal{E}_2''.\Pi|_R$

Proof. We proceed by first showing that the conditions hold for a program S that contains only a single `update` Δ statement. We then use induction to argue that these properties hold for programs with an arbitrary number of update statements.

First, we assume noninterference for the fragment of RX that does not contain any policy updates statements. This is permissible since noninterference of full RX follows from a modular proof of subject reduction of RX^2. The proof of noninterference for the update-free fragment of RX does not use the (E-UP) and (E-RB) rules. Similarly, the occurrences of the (TR0) and (L-TR) rules can be replaced with the (E-TR) rules frome the RX semantics. Thus, the proof of noninterference for the update-free fragment does not rely on this lemma.

The assumption of noninterference for update-free programs is manifested by the following statement: let $\mathcal{E}_1$ and $\mathcal{E}_2$ be program configurations such that $\mathcal{E}_1|_R \equiv \mathcal{E}_2|_R$ and let $S = \ldots; S'; \ldots$ be an update-free program such that $\Omega \vdash S$. Then, the derivation of $\Omega \vdash S$ contains a sub-derivation $\Omega' \vdash S'$. Suppose $\exists\rho \in \rho_L(R, \mathcal{E}_i.\Pi).\Omega'.Q \vdash \Omega'.\text{pc} \sqsubseteq \rho$ and $\Omega \models \mathcal{E}_1$ and $\Omega \models \mathcal{E}_2$. Then, for all finite executions $\langle\mathcal{E}_1, S\rangle$ and $\langle\mathcal{E}_2, S\rangle$ we have

$$\mathcal{E}_1, S \longrightarrow \mathcal{E}_1', S'; S'' \iff \mathcal{E}_2, S \longrightarrow \mathcal{E}_2', S'; S''$$

Next, we show, by analysis of the structure of the typing judgment $\Omega \vdash S$, that if $\Omega \vdash \texttt{trans}_{(\ell,Q)}\ S'$ contains a single sub-derivation $\Omega' \vdash \texttt{skip}; \texttt{update}\ \Delta$, then $\exists\rho \in \rho_L.\Omega'.\text{pc} \sqsubseteq \rho$. According to the typing judgment (T-TR) $\ell = \Omega.\text{pc}$. Now, the derivation $\Omega' \vdash \texttt{skip}; \texttt{update}\ \Delta$ must end with an application of (T-SEQ), with (T-SKIP) and (T-UP)($\Omega' \vdash \texttt{update}\ \Delta$) in the premise, where $\Omega'.\Phi = (\ell, Q)$. The first premise of (T-UP) asserts $\Omega.Q \vdash \Omega'.\text{pc} \sqsubseteq \ell$. Using the premise (L1), transitivity of the $\sqsubseteq$ relation and soundness of the static label ordering judgement, we can conclude $\exists\rho \in \rho_L.\Omega'.\text{pc} \sqsubseteq \rho$.

Using our assumption of noninterference for the update-free fragment, we can conclude that

$$\mathcal{E}_1, S \longrightarrow^* \mathcal{E}_1', \texttt{trans}_{(\ell,Q}\ \texttt{skip}; \texttt{update}\ \Delta; S' \iff$$
$$\mathcal{E}_2, S \longrightarrow^* \mathcal{E}_2', \texttt{trans}_{(\ell,Q)}\ \texttt{skip}; \texttt{update}\ \Delta; S'$$

from which property (i) follows immediately for programs with a single update statement.

We show properties (ii) and (iii) simulataneously, by noting first that the application of (T-UP) in the derivation $\Omega' \vdash \texttt{update}\ \Delta$ contains the premise $\Omega'.Q \vdash \Omega.PC \sqsubseteq lab(\Delta)$. This implies that for all s such that $\texttt{add}dels \in \Delta$ that $roledefs \in \rho_L \cup \rho_M$. Next, we note that if $(\mathcal{E}_1', \texttt{trans}_\Phi\ \texttt{update}\ \Delta; S'') \overset{\text{(E-UP)}}{\longrightarrow} (\mathcal{E}_1'', \texttt{trans}_\Phi\ S'')$ then $dclas(R, \mathcal{E}_1'.\Pi, \mathcal{E}_1''.\Pi) = \emptyset$.

To show (iii), Since revocations ($\texttt{del}\ (s)$) are irrelevant, it suffices to show that if $\Pi_1|_R \equiv \Pi_2|_R$, $roledefs \in \rho_L \cup \rho_M$, and $dclas(R, \Pi_1, \Pi_1 \cup s) = \emptyset$ then $\Pi_1 \cup s|_R \equiv \Pi_2 \cup s|_R$. But this follows immediately from the definition of $dclas$ since s can only add a delegation relation from roles in ρ_L to ρ_M. Since roles in $\rho_L \cup \rho_M$ are already observable in $\Pi_2|_R$ we have $\Pi_i|_R \equiv (\Pi_i \cup s)|_R$ and $(\Pi_1 \cup s)|_R \equiv (\Pi_2 \cup s)|_R$.

Let $\Pi \cdot \Delta$ denote the policy that results from transforming the policy Π according the the $\texttt{add}\ (s)$ and $\texttt{del}\ (s)$ statements that appear in Δ. To show (ii), i.e., that $(\mathcal{E}_1', \texttt{trans}_\Phi\ \texttt{update}\ \Delta)$ takes a step under (E-UP) if and only if $(\mathcal{E}_2', \texttt{trans}_\Phi\ \texttt{update}\ \Delta)$ takes a step under (E-UP) too, if suffices to show that $\forall q \in Q.(\mathcal{E}_1'.\Pi \cdot \Delta) \vdash q \iff (\mathcal{E}_2'.\Pi \cdot \Delta) \vdash q$. However, from the third premise of (T-UP) ($\Omega'.Q \vdash lab(()Q) \sqsubseteq \ell$) to deduce that $\forall q \in Q.\text{roles}(q) \subseteq \rho_L \cup \rho_M$. From (iii) we conclude that $\mathcal{E}_1'.\Pi \cdot \Delta|_R \equiv \mathcal{E}_1'.\Pi \cdot \Delta|_R$ and using Lemma 13 we prove the necessary.

Finally, we note that this result is easily extended to programs with more that one update statement. This can be seen by noting that after update statements the policies remain observationally equivalent. If an update statement succeeds in updating the policy from Π to Π' (makes a transition using (E-UP)), an equivalent execution of the transaction is obtained by executing the entire transaction under Π' and removing the update statement to result in a program with one less update statement. The same basic argument holds true when an update statement takes a step using (E-RB), except since mutually incompatible update statements can result in non-termination we cannot simply remove update statements. However, the statement of the Lemma explicitly requires that both executions terminate. Thus, when considering (E-RB) for terminating transactions, we can safely ignore update statements that are re-encountered when the transaction resumes. □

B Proving Noninterference

Our objective is to show subject reduction for RX^2, from which noninterference for RX follows directly. This is because our definition of well-formedness for RX^2 configurations $\mathcal{C}$ includes a clause that states that the pair of RX confiurations embedded within $\mathcal{C}$, (i.e. $\lfloor\mathcal{C}\rfloor_1$ and $\lfloor\mathcal{C}\rfloor_2$) are observationally equivalent. A proof of subject reduction would guarantee that observational equivalence of these configurations is preserved, which is precisely the property we desire for noninterference.

In preparation for the statement and proof of subject reduction, we need to introduce an additional judgment.

B.1 An Auxiliary Judgment

Our proof of subject reduction will proceed by induction on the structure of the derivation based on the operational rules in Figure A.4. However, some technicalities in the form of the operational rule (E-TR2) and the type rule (T-TR) requires special attention. In this section we present an auxiliary judgment $\Omega \vdash_{\mathcal{C},i,R} \mathsf{S}$ which permits us to formulate an induction hypothesis sufficiently strong to show subject reduction.

We consider (E-TR2) first. (E-TR2) is a congruence rule in which a statement S enclosed within a transaction as $\mathtt{trans}_\Phi\ \mathsf{S}$ takes a step in the premise independently of the transaction declaration. Suppose $\mathsf{S} \equiv \mathtt{if}\ (q)\ S_1\ S_2$, then due to the structure of the type rule (T-IFQ), S is not well-typed — all policy query statements must appear within a transaction. Thus, even though we might have $\Omega \vdash \mathtt{trans}_\Phi\ \mathsf{S}$, $\Omega \vdash \mathsf{S}$ does not hold, and so any inductive proof of subject reduction for Rx^2 faces the problem that the inductive hypothesis of a well-typed program does not hold in the premise of (E-TR2).

An alternative formulation of the operational rules would have been to eliminate the congruence rule (E-TR2), an replace it instead with a collection of rules to handle $\mathtt{trans}_\Phi\ \mathsf{E}_1 + \mathsf{E}_2, \mathtt{trans}_\Phi\ x := \mathsf{E}, \mathtt{trans}_\Phi\ \mathtt{if}\ (S)\ E_1\ E_2 e$, etc. separately. This proliferation of rules is clearly undesirable as it obscures the true structure of the semantics.

The second issue to contend with here is the structure of the type rule (T-TR). The conclusion of (T-TR) requires that a statement $\mathtt{trans}_\Phi\ \mathsf{S}$ be typed in a context Ω, where $\Omega.Q = \emptyset$. The need for this condition is illustrated by the following example.

$$\begin{array}{l}
\mathtt{trans}_{(\ell,\{A.r \sqsubseteq B.r\})} \\
\quad x := y; \\
\quad \mathtt{update\ del}\ (A.r \longleftarrow B.r) \\
\Omega.\Gamma \equiv \{(x, B.r), (y, A.r)\} \\
\Omega.Q \equiv \{A.r \sqsubseteq B.r\}
\end{array}$$

$$\frac{\begin{array}{c}\Phi.\mathrm{pc} = \mathrm{pc} \qquad Q \subseteq \Phi.Q \\ \Gamma; \mathrm{pc}; Q; \Phi \vdash \mathsf{S}\end{array}}{\Gamma; \mathrm{pc}; Q; . \vdash \mathtt{trans}_\Phi\ \mathsf{S}} \text{ (T-TR-BAD)}$$

If (T-TR) were replaced with (T-TR-BAD), then the program above would typecheck since the assigment $x := y$ is permitted when $Q \vdash A.r \sqsubseteq B.r$. If we consider an unchanged operational semantics and suppose the update statement $\mathtt{del}\ (A.r \longleftarrow B.r)$ resulted in a policy that violated the ordering $A.r \sqsubseteq B.r$, then the transaction would be rolled-back. This would undo the effect of the assignment $x := y$ but on resumption of the transaction, $x := y$ would be re-executed even though the new policy does not permit the flow. The condition in (T-TR) which requires that the policy context $\Omega.Q$ be empty prevents such a bad situation from occuring.

When combined with (E-TR2), this requirement creates a problem when stating an induction hypothesis for subject reduction. Consider the program $\mathsf{S} \equiv \mathtt{trans}_\Phi\ \mathtt{if}\ (q)\ S_1\ S_2$, $\Omega \vdash \mathsf{S}$, which takes a step using (E-TR2) with (E-IFQ) in the premise to $\mathsf{S}' = \mathtt{trans}_\Phi\ S_1$. The derivation of $\Omega \vdash \mathsf{S}$, contains a subderivation $\Omega[Q = \{q\}][\ldots] \vdash S_1$, since we may statically assume q to be true when S_1 is executed. However, any derivation of $\Omega' \vdash \mathsf{S}'$ must end with an application of (T-TR) which requires $\Omega'.Q = \emptyset$.

$\boxed{\Omega \vdash_{\mathcal{C},i,R} \mathsf{S}}$

$$\frac{\begin{array}{c}\Omega \models_R \mathcal{C} \\ \lfloor \mathcal{C}.\Psi \rfloor_i = \cdot \qquad \Omega \vdash \mathsf{S}\end{array}}{\Omega \vdash_{\mathcal{C},i,R} \mathsf{S}} \text{ (T-A1)}$$

$$\frac{\begin{array}{c}\Omega \models_R \mathcal{C} \qquad\qquad \lfloor \mathcal{C}.\Psi \rfloor_i = (\mathsf{M}, \mathtt{trans}_{\Phi'}\ \mathsf{S}_1) \\ (\mathsf{S} = \mathtt{trans}_{\Phi'}\ \mathsf{S}_2 \Rightarrow \mathsf{S}' = \mathsf{S}_2) \\ (\mathsf{S} \neq \mathtt{trans}_{\Phi'}\ \mathsf{S}_2 \Rightarrow \mathsf{S}' = \mathsf{S}) \\ Q' \subseteq \Phi.Q \qquad\qquad \forall q \in Q. \lfloor \mathcal{C}.\Pi \rfloor_i \vdash q \\ \Omega.\mathrm{pc} = \Phi.\mathrm{pc} \qquad \Omega.\Gamma; \Omega.\mathrm{pc}; Q = Q'; \Phi = \Phi' \vdash \mathsf{S}'\end{array}}{\Omega \vdash_{\mathcal{C},i,R} \mathsf{S}} \text{ (T-A2)}$$

$$\frac{\begin{array}{c}\mathcal{C}.\Psi = \langle \Psi_1^* \parallel \Psi_2^* \rangle \\ \Omega \models_R \mathcal{C} \qquad\qquad \rho \in \rho_M(R, \mathcal{C}.\Pi) \cup \rho_H(R, \mathcal{C}.\Pi) \\ \mathrm{pc}' = \Omega.\mathrm{pc} \sqcup \rho \qquad \forall i \in \{1,2\}. \Omega[\mathrm{pc} = \mathrm{pc}'] \vdash_{\mathcal{C},i,R} S_i\end{array}}{\Omega \vdash_{\mathcal{C},0,R} \langle S_1 \parallel S_2 \rangle} \text{ (T-A3)}$$

$$\frac{\Omega \vdash_{\mathcal{C},i,R} \mathtt{trans}_\Phi\ \mathsf{S} \qquad \Omega \vdash \mathsf{S}'}{\Omega \vdash_{\mathcal{C},i,R} \mathtt{trans}_\Phi\ \mathsf{S}; \mathsf{S}'} \text{ (T-A4)}$$

Figure 12. Augmented typing judgment.

To address these two idiosyncracies of (E-TR2) and (T-TR) we formulate an auxiliary judgment $\Omega \vdash_{\mathcal{C},i,R} \mathsf{S}$ in Figure B.1 which takes into account the state of the program configuration $\mathcal{C}$ when typing a program S. Intuitively, given a configuration pair $\mathcal{C}, \mathsf{S}$, where $\mathcal{C}.\Psi = (\mathsf{M}, \mathtt{trans}_\Phi\ S)$ then we can treat S as being enclosed within a transaction declaration as $\mathtt{trans}_\Phi\ \mathsf{S}$. This is reasonable since the policy update rules inspect $\mathcal{C}.\Psi$ to determine whether or not S is enclosed within a transaction. We have to take care in the second rule to ensure that we do not introduce nested transactions; hence the second predicate. This solves the problem presented by the structure of the congruence rule (E-TR2). Additionally, as long as the program $\mathtt{trans}_\Phi\ S$ that appears in the snapshot can be typed in a context $\Omega[Q = \emptyset]$ (as required by the well-formedness condition $\Omega \models_R \mathcal{C}$), then it is permissible to type $\mathtt{trans}_\Phi\ S$ in a context Ω where all queries $\Omega.Q$ are compatible with Φ and the current policy $\mathcal{C}.\Pi$. This is reasonable since the program that executes after a transaction is rolled-back (the one in the snapshot) makes no static assumption about the policy. Thus, the problem illustrated by the example program cannot occur.

This judgment is sound with respect to the typing judgment $\Omega \vdash S$ according to the following lemma.

Lemma 15. *Given a program S, a context Ω, a configuration $\mathcal{C}$, an observation level R and an index $i \in \{0,1,2\}$, $\Omega \vdash_{\mathcal{C},i,R} \mathsf{S}$, if and only if, $\exists \mathsf{S}_0, \mathcal{C}'. \Omega \vdash \mathsf{S}_0$ and $\mathsf{S}_0 /_i\, \mathcal{C}' \longrightarrow^* \mathsf{S}_1 /_i\, \mathcal{C}$ and either $\mathsf{S}_1 = \mathsf{S}$, or the derivation* *of* $\mathsf{S}_1 \;/_i\; \mathcal{C} \longrightarrow \mathsf{S}_2 \;/_i\; \mathcal{C}'$ *contains a small-step using* $\longrightarrow$ *or* $\rightsquigarrow$ *starting from* $\mathsf{S} \;/_i\; \mathcal{C}$.

B.2 Interpretation of the effect lower bound

In this section we present two lemmas that simplify the proof of subject reduction that follows. These lemmas, in essence, guarantee that observational equivalence of a pair of configurations is maintained when an $\mathrm{R}\mathrm{X}^2$ program takes a single step. Given these lemmas, the proof of subject reduction in the next section is concerned primarily with showing that the program that results after a single step of execution of a well typed program is also well typed.

Lemma 16. *Suppose for a program* S *and a configuration* $\mathcal{C}$ *such that* $\Omega \vdash_{\mathcal{C},i,R} \mathsf{S}$, $\mathsf{S} \;/_i\; \mathcal{C} \longrightarrow \mathsf{S}' \;/_i\; \mathcal{C}'$ *or* $\mathsf{S} \;/_i\; \mathcal{C} \rightsquigarrow \mathsf{S}' \;/_i\; \mathcal{C}'$. *Then, if* $\forall \rho \in R.\mathcal{C}.\Pi \not\vdash \Omega.\mathrm{pc} \sqsubseteq \rho_i$ *then* $\mathcal{C}|_R = \mathcal{C}'|_R$.

Proof. We proceed by induction on the structure of the derivation $\mathsf{S} \;/_i\; \mathcal{C} \longrightarrow \mathsf{S}' \;/_i\; \mathcal{C}'$. Clearly, if $\mathcal{C} = \mathcal{C}'$ then the lemma holds trivially. The only observable components of a configuration $\mathcal{C}$ are $\mathcal{C}.\mathsf{M}$ and $\mathcal{C}.\Pi$. We consider memory and policy effects separately.

Let us first consider the memory effects of a program. All updates to memory occur through the function $\mathit{updloc}_i(x, \mathsf{v}, \mathsf{M})$ and $\mathit{rollback}_i(\mathsf{M}, \mathsf{M}')$. Applications of these functions appear in the premises of (E-ASV) and (R-UP). If the derivation of $\mathsf{S} \;/_i\; \mathcal{C} \longrightarrow \mathsf{S}' \;/_i\; \mathcal{C}'$ ends with an application of (E-ASV) rule, then $\Omega \vdash_{\mathcal{C},i,R} \mathsf{S}$ contains an application (T-ASN) for $\Omega' \vdash x := \mathsf{v}$, where $\forall \Pi.\Pi \vdash \Omega.\mathrm{pc} \sqsubseteq \Omega'.\mathrm{pc}$. The final premise of (T-ASN) requires that for x, the location being written to, we have that $\Omega'.Q \vdash \Omega'.\mathrm{pc} \sqsubseteq \Omega'.\Gamma(x)$.

Suppose, for contradiction, that $\mathcal{C}.\mathsf{M} \;|_{R,\mathcal{C}.\Pi} \neq \mathcal{C}'.\mathsf{M} \;|_{R,\mathcal{C}'.\Pi}$, then, since $\mathcal{C}.\Pi = \mathcal{C}'.\Pi$, for the change to be observable, we must have that $x \in \mathit{dom}(\mathcal{C}.M \;|_{R,\mathcal{C}.\Pi}) = \mathit{dom}(\mathcal{C}'.M \;|_{R,\mathcal{C}'.\Pi})$. By the definition of memory observability, this is possible if and only if $\exists \rho \in R$ such that $\mathcal{C}.\Pi \vdash \Omega.\Gamma(x) \sqsubseteq \rho$. However, by assumption and transitivity of the label ordering $\sqsubseteq$, and Lemma 9 (soundness of the static decision procedure for label ordering) we have that $\forall \rho \in R.\mathcal{C}.\Pi \vdash \Omega.\Gamma(x) \sqsubseteq \rho$; i.e. a contradiction.

An effect on memory as a result of the (R-UP) rule is combined with a policy effect. Suppose the derivation of $\mathsf{S} \;/_i\; \mathcal{C} \rightsquigarrow \mathsf{S}' \;/_i\; \mathcal{C}'$ concludes with an application of the (R-UP) rule. This implies that the derivation of $\Omega \vdash_{\mathcal{C},i,R} \mathsf{S}$ must contain a subderivation that contains an application of (T-UP), for $\Omega' \vdash \texttt{update}\ \Delta$, where $\forall \Pi.\Pi \vdash \Omega.\mathrm{pc} \sqsubseteq \Omega'.\mathrm{pc}$. At the conclusion of a step of evaluation using (R-UP), we have that $\mathcal{C}'.\mathsf{M} \equiv \mathcal{C}.\Psi.\mathsf{M}$. From clause (3.3) of the Definition 11 ($\Omega \models_R \mathcal{C}$), we have that $\mathcal{C}.\Pi \vdash \Omega.\mathrm{pc} \sqsubseteq \Omega.\Gamma(x)$ for all changed locations in the transaction x. Suppose, for contradiction, that after the policy is updated $\lfloor \mathcal{C}'.M \rfloor_1$ is not observationally equivalent to $\lfloor \mathcal{C}'.M \rfloor_2$, at observation level R. By the definition of memory observability in Figure 6, this can only occur if for one of the locations x modified during the transaction, we have $\exists \rho \in \rho_L.\lfloor \mathcal{C}'.\Pi \rfloor_i \vdash \Gamma(x) \sqsubseteq \rho$. However, by hypothesis with have that $\lfloor \mathcal{C}.\Pi \rfloor_i \vdash \Omega.\mathrm{pc} \not\sqsubseteq \rho$. Thus we have that $\Gamma(x) \in \rho_M(R, \lfloor \mathcal{C}.\Pi \rfloor_i) \cup \rho_H(R, \lfloor \mathcal{C}.\Pi \rfloor_i)$, but $\Gamma(x) \in \rho_L(R, \lfloor \mathcal{C}.\Pi \rfloor_i)$. But this is precisely the condition under which $\mathit{dclas}(R, \lfloor \mathcal{C}.\Pi \rfloor_i, \lfloor CTX'.\Pi \rfloor_i) \neq \emptyset$; so, (R-UP) is not applicable. Thus, we have reached a contradiction. cannot take a step.

Finally we consider policy effects. Here we have that a single step of evaluation causes policy $\mathcal{C}.\Pi$ to differ from $\mathcal{C}'.\Pi$. The only function that causes a change to policy is $\mathit{updpi}(\Pi, \Pi')$ which appears in the premises of the (E-UP) and (R-UP) rules. Thus, the derivation of $\Omega \vdash_{\mathcal{C},i,R} \mathsf{S}$ contains an application of (T-UP) for the subderivation $\Omega' \vdash \texttt{update}\ \Delta$, $\forall \Pi.\Omega.\mathrm{pc} \sqsubseteq \Omega'.\mathrm{pc}$. The premise of (T-UP) guarantees that $\lfloor \mathcal{C}.\Pi \rfloor_i \vdash \Omega'.\mathrm{pc} \sqsubseteq \mathit{lab}(\Delta)$. Therefore, all changes are to roles ρ where $\Omega'.PC \sqsubseteq C_\Pi(\rho)$. Since there are no declassification to roles in ρ_L, ρ_M, we have that $\mathcal{C}.\Pi|_R = \mathcal{C}'.\Pi|_R$. ☐

Corollary 17. *If* $\langle S_1 \parallel S_2 \rangle \;/_0\; \mathcal{C} \longrightarrow \langle S_1' \parallel S_2' \rangle \;/_0\; \mathcal{C}'$ *and* $\Omega \vdash_{\mathcal{C},0,R} \langle S_1 \parallel S_2 \rangle$ *then* $\mathcal{C}|_R = \mathcal{C}'|_R$.

Proof. Follows immediately from Lemma 16 and premise of (T-SBR) and (A-T3) which requires S_1 and S_2 to be checked in Ω' where $\Omega'.\mathrm{pc} \in \rho_M \cup \rho_H$. ☐

Lemma 18. *If* $\mathsf{S} \;/_0\; \mathcal{C} \longrightarrow \mathsf{S}' \;/_0\; \mathcal{C}'$, *then if* $\Omega \vdash_{\mathcal{C},i,R} \mathsf{S}$, $\lfloor \mathcal{C}' \rfloor_1|_R = \lfloor \mathcal{C}' \rfloor_2|_R$.

Proof. Proceeds in similarly to the proof of Lemma 14, using the premise of (T-EBR) to show that v in (E-ASV) does not contain a bracket. ☐

B.3 Subject Reduction

Theorem 19 (Subject Reduction). *Given a program* S *and a configuration* $\mathcal{C}$ *such that* $\mathsf{S} \;/_i\; \mathcal{C} \longrightarrow \mathsf{S}' \;/_i\; \mathcal{C}'$ *or* $\mathsf{S} \;/_i\; \mathcal{C} \rightsquigarrow \mathsf{S}' \;/_i\; \mathcal{C}'$; *if there is a context* Ω *such that* $i \in \{1,2\} \Rightarrow \Omega.\mathrm{pc} \in \rho_M \cup \rho_H$ *and* $\Omega \vdash_{\mathcal{C},i,R} \mathsf{S}$; *then* $\Omega \vdash_{\mathcal{C}',i,R} \mathsf{S}'$. *Similarly, if* $\mathsf{E} \;/_i\; \mathcal{C} \longrightarrow \mathsf{E}' \;/_i\; \mathcal{C}$, *and* $\Omega \models_R \mathcal{C}$ *and* $\Omega \vdash \mathsf{E} : \ell$ *then* $\Omega \vdash \mathsf{E}' : \ell$.

Proof. By induction on the structure of the derivations $\mathsf{S} \;/_i\; \mathcal{C} \longrightarrow \mathsf{S}' \;/_i\; \mathcal{C}'$, $\mathsf{S} \;/_i\; \mathcal{C} \rightsquigarrow \mathsf{S}' \;/_i\; \mathcal{C}'$ and $\mathsf{E} \;/_i\; \mathcal{C} \longrightarrow \mathsf{E}' \;/_i\; \mathcal{C}$.

(E-VAR): If $i \in \{1,2\}$ then since $\lfloor \mathsf{M}(x) \rfloor_i$ is of the form of v (no brackets) then by replacing the occurence of (T-VAR) for x in $\Omega \vdash_{\mathcal{C},i,R} x$ with (T-LIT) for v, we can derive $\Omega \vdash_{\mathcal{C},i,R} \mathsf{v}$ is immediately applicable. If $i = 0$, then by clause (1) of $\Omega \models_R \mathcal{C}$, and since $\lfloor \mathsf{M}(x) \rfloor_0 = \mathsf{M}(x)$, we have that $\Omega \vdash \mathsf{v} : \Omega.\Gamma(x)$, and once again we can replace the occurence of (T-VAR) for x with the appropriate judgment for v.

(E-ADL), (E-ADR), (E-ADV): Trivial.

(E-SEQ): If the derivation of $\Omega \vdash_{\mathcal{C},i,R} \mathsf{S}; S$ ends with an application of (T-A1), and $\lfloor \mathcal{C}'.\Psi \rfloor_i = \cdot$ then (T-A1) is applicable again simply replacing $\Omega \vdash \mathsf{S}$ with $\Omega \vdash \mathsf{S}'$ from the inductive hypothesis. If $\lfloor \mathcal{C}'.\Psi \rfloor_i \neq \cdot$, then S must be of the form $\mathtt{trans}_\Phi\ \mathsf{S}'$. Thus (T-A4) is applicable, using $\Omega \vdash S$ from (T-A1) of the hypothesis.

If the derivation of $\Omega \vdash_{\mathcal{C},i,R} \mathsf{S}; S$ ends with (T-A2), then since no rollback has occured, $\lfloor \mathcal{C}'.\Pi \rfloor_i \vdash q \iff \lfloor \mathcal{C}.\Pi \rfloor_i$, the derivation of $\Omega \vdash_{\mathcal{C}',i,R} \mathsf{S}'; S$ can pick a Q' in the premise is at least as big as the Q' picked in the application of (T-A2) in the hypothesis.

The derivation of $\Omega \vdash_{\mathcal{C},i,R} \mathsf{S}; S$ cannot end with (T-A3).

If the derivation of $\Omega \vdash_{\mathcal{C},i,R} \mathsf{S}; S$ ends with (T-A4), then either (T-A1) or (T-A4) is applicable for $\Omega \vdash_{\mathcal{C}',i,R} \mathsf{S}'; S$ depending on whether $\lfloor \mathcal{C}'.\Psi \rfloor_i = \cdot$. In either case, the necessary judgment $\Omega \vdash S$ is provided by the hypothesis.

(E-SKP), (E-ASE): Trivial.

(E-ASV): If we can show $\Omega \models_R \mathcal{C}'$ then the rest is trivial. From the premise $\Omega \vdash_{\mathcal{C},i,R} x := \mathsf{v}$, we have that $\Omega \vdash \mathsf{v} : \Omega.\Gamma(x)$. This is sufficient to show clause 1. Since *updloc* does not change $\mathcal{C}.\Pi$ we have clause 2 trivially. To show clause (3.3), note that if $\lfloor \mathcal{C}.\Psi \rfloor_i = (\mathsf{M}', \mathtt{trans}_\Phi\ \mathsf{S})$ then the derivation of $\Omega \vdash_{\mathcal{C},i,R} x := v$ ends with an application of (T-A2). The premise of (T-A2) requires that $x := v$ be checked in a context Ω' where $\Omega'.\mathrm{pc} = \Omega.\mathrm{pc}$, using the (T-ASN) rule. But the premise of the (T-ASN) rule requires that $\Omega'.Q \vdash \Omega.\mathrm{pc} \sqsubseteq \Omega.\Gamma(x)$, while the premise of (T-A2) guarantees that $\Omega'.Q \subseteq \Phi.Q$ and $\forall q \in \Omega'.Q.\lfloor \mathcal{C}.\Pi \rfloor_i \vdash q$. This is precisely the requirement of clause (3.3). When $i \in \{1, 2\}$ clause (4) follows from Corollary 17. When $i = 0$ clause (4) follows from Lemma 18.

(E-IFE), (E-IFV), (E-WHL): Trivial.

(E-IFQ-0), (E-IFQ-i): For $i \in \{0, 1, 2\}$ the derivation of $\Omega \vdash_{\mathcal{C},i,R} \mathtt{if}\ (q)\ S_1\ S_2$ must end with an application of (T-A2), since the premise of (T-IFQ) requires that policy queries appear within transactions. If $j = 2$ we can derive $\Omega \vdash_{\mathcal{C},i,R} S_j$ using the same choice of Q' in the premise of (T-A2) as that used in the hypothesis. If $j = 1$, we can augment Q' with q since $\lfloor \mathcal{C}.\Pi \rfloor_i \vdash q$ and (T-IFQ) ensures that $q \in \Phi.Q$.

(E-TR1-0): Since $\mathcal{C}.\Psi = \cdot$ the derivation of $\Omega \vdash_{\mathcal{C},i,R} \mathtt{trans}_\Phi\ \mathsf{S}$ ends with an application of (T-A1). Since $\mathcal{C}'.\Psi \neq \cdot$, (T-A2) applies, which essentially inlines the derivartion of $\Omega \vdash \mathtt{trans}_\Phi\ \mathsf{S}$ from (T-A1) into (T-A2). It remains to be shown that $\Omega \models_R \mathcal{C}'$. Clause 3 is relevant here, but since $\mathcal{C}'.\Psi.M = \mathcal{C}'.M$ clause 3.1 and 3.3 are satisfied directly. $\Omega \vdash \mathcal{C}'.\Psi.\mathsf{S}$ is shown using the premise of (T-A1) from the hypothesis.

(E-TR1-i): Identical to (E-TR1-0) except $updpsi_i$ ensures that $\lfloor \mathcal{C}'.\Psi.\mathsf{M} \rfloor_i = \lfloor \mathcal{C}.\mathsf{M} \rfloor_i$, which is sufficient for clause (3.3) of $\Omega \models_R \mathcal{C}'$.

(E-TR2), (E-TR3): Trivial.

(E-UP): Since $\mathtt{skip}$ is typeable in any context Ω, the interesting part here is showing that $\Omega \models_R \mathcal{C}'$. Clause (1) is satisfied by hypothesis since $\mathcal{C}'.\mathsf{M} = \mathcal{C}.\mathsf{M}$. Clause (2) is satisfied since, the premise of (E-UP) guarantees that by using *updpi*, $\lfloor \mathcal{C}'.\Pi \rfloor_j = \lfloor \mathcal{C}.\Pi \rfloor_j$. Furthermore, since $\Omega.Q \subseteq \Phi.Q$, and the premise of (E-UP) guarantees that $\lfloor \mathcal{C}'.\Pi \rfloor_i \vdash q \iff \lfloor \mathcal{C}.\Pi \rfloor_i \vdash q$. We use the same identity to show clause (3.3) — by the hypothesis we have that for all locations x such that its value in the memory snapshot differs from the value in memory, $Q \vdash \Omega.\mathrm{pc} \sqsubseteq \Omega.\Gamma(x)$, where $\forall q \in Q.\lfloor \mathcal{C}.\Pi \rfloor_i \vdash q$, and $Q \subseteq \Phi.Q$. But by the policy consistency condition we have also have that $\forall q \in Q.\lfloor \mathcal{C}.\Pi \rfloor_i \vdash q$. Thus, since Φ is unchanged, we can use the same Q used in the hypothesis to show that clause (3.3) hold for $\mathcal{C}'$. If $i \in \{1, 2\}$ clause (4) is satisfied by Corollary 17. If $i = 0$ clause (4) is satisfied by Lemma 18.

(R-UP): By assumption of $\Omega \models_R \mathcal{C}$ we have that $\mathtt{trans}_\Phi\ \mathsf{S}$ is type-able in a context with an empty set of policy assumptions Q. To show that $\Omega \models_R \mathcal{C}'$, we note that since $\lfloor \mathcal{C}'.M \rfloor_i \equiv \lfloor \mathcal{C}.\Psi.\mathsf{M} \rfloor_i$ and from clause (3.2), we can derive clause (1) of $\Omega \models_R \mathcal{C}'$. For clause (2) we note that $\Omega.Q$ is empty. For clause (3) we note that $\lfloor \mathcal{C}'.M \rfloor_i \equiv \lfloor \mathcal{C}'.\Psi.\mathsf{M} \rfloor_i$. Finally, clause (4) follows from Corollary 17 or from Lemma 18.

(R-SEQ), (E-TR4): Trivial.

(E-BRK): The derivation of $\Omega \vdash_{\mathcal{C},0,R} \langle S_1 \parallel S_2 \rangle$ ends with an application of (A-T3). Since $\lfloor \mathcal{C}' \rfloor_j \equiv \lfloor \mathcal{C} \rfloor_j$ and $S_j = S_j'$, we can once again apply (A-T3) for $\Omega \vdash_{\mathcal{C},0,R} \langle S_1' \parallel S_2' \rangle$ reusing the sub-derivation of S_j from $\Omega \vdash_{\mathcal{C},0,R} \langle S_1 \parallel S_2 \rangle$.

(L-SKIP), (L-ADD): Trivial.

(L-IFE): By (T-EBR), we are assured that if $\Omega \vdash \langle E_1 \parallel E_2 \rangle : \ell$, then $\exists \rho \in \rho_M \cup \rho_H.\rho \sqcup ell$. Thus, in $\Omega \vdash_{\mathcal{C},i,R} \mathtt{if}\ (\langle S_1 \parallel S_2 \rangle)\ E_1\ E_2$ we have that S_1 and S_2 are checked in contexts Ω' where $\rho \sqsubseteq \Omega'.\mathrm{pc}$. This is sufficient to guarantee the premise of (T-A3) necessary in the application of (A-T3) for $\Omega \vdash_{\mathcal{C},i,R} \langle \texttt{if}\ (E_1)\ E_1\ E_2 \parallel \texttt{if}\ (E_2)\ E_1\ E_2\rangle$.

(L-IFQ): By a similar argument to (L-IFE) and using the definition of $lab(q)$ to deduce that $\mathrm{roles}(q) \not\subseteq \rho_L \cup \rho_M$ implies $\exists \rho \in role_M \cup \rho_H . \rho \sqsubseteq C_\Pi(q)$.

(L-TR): Using (A-T3) with (A-T1) in the premise for $\Omega \vdash_{\mathcal{C},0,R} \langle \texttt{trans}_\Phi\ S \parallel \texttt{trans}_\Phi\ S\rangle$, and noticing that (T-TR) requires that in $\Omega \vdash \texttt{trans}_\Phi\ S, \Omega.\mathrm{pc} = \Phi.\ell$. ☐

C A Specific Choice of Metapolicy

In this section we show how a particular instantiation of the metapolicy $C_\Pi(\rho)$ and $I_\Pi(\rho)$ in terms of the delegation structure of an RT_0 policy satisfies the requirements of Lemma 13.

C.1 The Full RT_0 Language

In the paper so far we have only presented a restricted version of the RT_0 policy language. Table 1 shows the semantics of the the full RT_0 language, where the new statement forms that were previously elided are boxed. Role ρ is defined by statements of the form $\rho \longleftarrow e$ whose semantics is described informally at the top of Table 1. The semantics is formalized by interpreting the policy Π as a datalog program $SP(\Pi)$, called the *semantic program* of Π, defined in terms of one ternary predicate $m(A, r, D)$. Intuitively, $m(A, r, D)$ means that the principal D is a member of the role $A.r$. $SP(\Pi)$ is the set of all datalog clauses produced from policy statements in Π. The rules to generate $SP(\Pi)$ from Π are shown at the bottom of Table 1. Symbols that start with "?" represent logical variables.

The semantics of $SP(\Pi)$ is given using a model-theoretic approach. $SP(\Pi)$ is viewed as a set of first order sentences interpreted in the minimal Herbrand model. We write $SP(\Pi) \models m(A, r, D)$ when $m(A, r, D)$ is in the minimal Herbrand model of $SP(\Pi)$. The formal development of the Herbrand model is standard and is not repeated here. With this, we can define the semantics of role $A.r$ with respect to policy Π as

$$[\![A.r]\!]^\Pi \equiv \{Z \mid SP(\Pi) \models m(A, r, Z)\}$$

C.2 The Metapolicy $C^{\mathrm{del}}(\rho, \Pi)$

Table 1 also shows a metapolicy $C^{\mathrm{del}}(\rho, \Pi)$ which can be used to instantiate the confidentiality of a role definition $C_\Pi(\rho)$. The same metapolicy can also be used to instantiate the integrity of a role definition, $I_\Pi(role)$.

Intuitively, $C^{\mathrm{del}}(\rho, \Pi)$, treats the policy as though it were an undirected graph where the nodes represents roles ρ, and an edge (ρ_1, ρ_2) is present in the graph if ρ_1 delegates directly to ρ_2 or vice-versa. $C^{\mathrm{del}}(()\rho, \Pi)$ is the set of roles ρ' in the graph that belong to the same *connected component* as ρ.

The intention here is to show that it is possible to instantiate a metapolicy in terms of the struture of the policy itself, while still maintaining the requirements presented in Section 3.5. Thus, the semantics of this metapolicy is conditional on the state of the current policy itself. It might be possible to articulate more restrictive metapolicies if we make the conditions of policy observability more restrictive. For instance, a member of a role ρ can observe $[\![\rho]\!]^\Pi$ while not being able to observe the statement $s \in \Pi$ where $roledef\, s = \rho$.

This metapolicy places all roles related by delegation into an equivalence class. Thus, the premises of (C1) and (C2) of Lemma 13 are obviously satisfied. We provide below a Lemma similar to Lemma 13 that relies directly on the semantics of a policy, $SP(\Pi)$ rather than on premises (C1) and (C2), to show that $C^{\mathrm{del}}(\rho, \Pi)$ is a valid choice of metapolicy.

Lemma 20. *Given policies Π_1 and Π_2 and an observation level R such that $\Pi_1|_R = \Pi_2|_R$; then for any query q such that* $\mathrm{roles}(q) \subseteq \rho_L \cup \rho_M$, $\Pi_1 \vdash q \iff \Pi_2 \vdash q$.

Proof. We show that $\Pi \vdash q \iff \Pi\,|_R \vdash q$. Since we have $\Pi_1\,|_R = \Pi_2\,|_R$, the statement of the lemma follows immediately.

A formula q is decided by the interpretation of the policy Π as the semantic program $SP(\Pi)$. That is, $\Pi \vdash L_1 \sqsubseteq L_2$ iff $[\![L_2]\!]^\Pi \subseteq [\![L_2]\!]^\Pi$, where $[\![L]\!]^\Pi$ is defined by the semantic program $SP(\Pi)$. To show that $\Pi \vdash L_1 \sqsubseteq L_2 \iff \Pi|_R \vdash L_1 \sqsubseteq L_2$, it suffices to show that $[\![L_i]\!]^\Pi = [\![L_i]\!]^{\Pi|_R}$.

Under the choice of metapolicy where $C_\Pi(\rho) = C^{\mathrm{del}}(\rho, \Pi)$ and $I_\Pi(\rho) = C^{\mathrm{del}}(\rho, \Pi)$, it suffices to show the following two cases:

$$\begin{array}{ll} \multicolumn{2}{l}{\forall \rho \in \rho_L(R, \Pi) \cup \rho_M(R, \Pi).} \\ (1) & [\![\rho]\!]^\Pi = [\![\rho]\!]^{\Pi|_R} \\ (2) & [\![C^{\mathrm{del}}(\rho, \Pi)]\!]^\Pi = [\![C^{\mathrm{del}}(\rho, \Pi)]\!]^{\Pi|_R} \end{array}$$

We first consider the interpretation of roles ρ. From the semantics of RT_0 we have that $[\![A.r]\!]^\Pi \equiv \{Z \mid SP(\Pi) \models m(A, r, Z)\}$, where $SP(\Pi)$ denotes the semantic program of the policy Π. The evaluation of $SP(\Pi)$ is given in terms of the Herbrand universe of $SP(\Pi)$. Let $T^\Pi\!\uparrow^k$ denote the kth iterate in the generation of the Herbrand universe of $SP(\Pi)$. We have that $SP(\Pi) \models m(A, r, Z) \iff \exists k. m(A, r, Z) \in T^\Pi\!\uparrow^k$. We show by induction on k that interpretation $[\![\rho]\!]^\Pi = [\![\rho]\!]^{\Pi|_R}$.

Base case (k=1): $m(A, r, Z) \in T^\Pi \uparrow^1 \iff m(A, r, Z) \in SP(\Pi) \iff A.r \leftarrow Z \in \Pi$. But, by assumption since $A.r \in \rho_L \cup \rho_M$ we have that $\forall s. roledef\, s = A.r \Rightarrow s \in \Pi|_R$. Thus, $m(A, r, Z) \in T^{\Pi_R}\!\uparrow^1$.

Informal meaning of RT_0 policy statements s

Statement Name	Statement Syntax	Meaning
Simple Member	$A.r \longleftarrow X$	All principals in X are member of A's r role.
Simple Inclusion	$A.r \longleftarrow B.r_1$	A's r role includes (all members of) B's r_1 role. This represents a delegation from A to B, as B may add principals to become members of the role $A.r$ by issuing statements defining $B.r_1$.
Linking Inclusion	$\boxed{A.r \longleftarrow B.r_1.r_2}$	$A.r$ includes $D.r_2$ for every D that is a member of $B.r_1$. This represents a delegation from A to B and all the members of the role $B.r_1$. We call $B.r_1.r_2$ a *linked role*.
Intersection Inclusion	$\boxed{A.r \longleftarrow B_1.r_1 \cap B_2.r_2}$	$A.r$ includes every principal who is a member of both $B_1.r_1$ and $B_2.r_2$. This represents partial delegations from A to B_1 and to B_2. We call $B_1.r_1 \cap B_2.r_2$ an *intersection*.

Translation of an RT_0 policy Π into a datalog program $SP(\Pi)$ to interpret roles

$$\begin{array}{lcl}
(A.r \longleftarrow X) \quad \in \Pi & \Longleftrightarrow & \forall B \in X.(m(A,r,B)) \quad \in SP(\Pi) \\
(A.r \longleftarrow B.r1) \quad \in \Pi & \Longleftrightarrow & (m(A,r,?Z) :\!-\, m(B,r1,?Z)) \quad \in SP(\Pi) \\
(A.r \longleftarrow B.r1.r2) \quad \in \Pi & \Longleftrightarrow & (m(A,r,?Z) :\!-\, m(B,r1,?Y), m(?Y,r2,?Z)) \quad \in SP(\Pi) \\
(A.r \longleftarrow B.r1 \cap C.r2) \quad \in \Pi & \Longleftrightarrow & (m(A,r,?Z) :\!-\, m(B,r1,?Z), m(C,r2,?Z)) \quad \in SP(\Pi)
\end{array}$$

A metapolicy, $C^{\mathrm{del}}(\rho,\Pi)$, based on the closure of the delegation relation

$$C^{\mathrm{del}}(\rho,\Pi) \equiv \lim_{i\to\infty} C^{\mathrm{del}}_i(\rho,\Pi) \qquad [\![C^{\mathrm{del}}(\rho,\Pi)]\!]^{\Pi} \equiv \bigcup_{\rho' \in C^{\mathrm{del}}(\rho,\Pi)} [\![\rho']\!]^{\Pi}$$

$$\begin{array}{lll}
& C^{\mathrm{del}}_0(\rho,\Pi) & \equiv \{\rho\} \\
& C^{\mathrm{del}}_{i+1}(\rho,\Pi) & \equiv C^{\mathrm{del}}_i(\rho,\Pi) \;\cup\; \{\rho' \mid \exists \rho'' \in C^{\mathrm{del}}_i(\rho,\Pi).\rho'' \in \mathit{dlg}_{\Pi}(\rho') \;\vee\; \rho' \in \mathit{dlg}_{\Pi}(\rho'')\} \\
& \mathit{dlg}_{\Pi}(A.r) & \equiv \mathit{dlg}_0(\Pi, A.r, \Pi) \\
& \mathit{dlg}_0(\Pi_0, A.r, \emptyset) & \equiv \emptyset \\
\text{where} & \mathit{dlg}_0(\Pi_0, A.r, \{B.r_1 \longleftarrow e\} \cup \Pi) & \equiv \mathit{dlg}_0(\Pi_0, A.r, \Pi) \qquad (\text{where } A.r \neq B.r_1) \\
& \mathit{dlg}_0(\Pi_0, A.r, \{A.r \longleftarrow X\} \cup \Pi) & \equiv \mathit{dlg}_0(\Pi_0, A.r, \Pi) \\
& \mathit{dlg}_0(\Pi_0, A.r, \{A.r \longleftarrow \rho\} \cup \Pi) & \equiv \mathit{dlg}_0(\Pi_0, A.r, \Pi) \cup \{\rho\} \\
& \mathit{dlg}_0(\Pi_0, A.r, \{A.r \longleftarrow \rho_1 \cap \rho_2\} \cup \Pi) & \equiv \mathit{dlg}_0(\Pi_0, A.r, \Pi) \cup \{\rho_1, \rho_2\} \\
& \mathit{dlg}_0(\Pi_0, A.r, \{A.r \longleftarrow B.r_1.r_2\} \cup \Pi) & \equiv \mathit{dlg}_0(\Pi_0, A.r, \Pi) \cup \{B.r_1\} \cup \{D.r_2 \mid D \in [\![B.r_1]\!]^{\Pi_0}\}
\end{array}$$

Table 1. Semantics of full RT_0 policies Π and a metapolicy $C^{\mathrm{del}}(\rho,\Pi)$.

Inductive Hypothesis: $\forall i < k.m(A, r, z) \in T^{\Pi}\uparrow^{i} \iff m(A, r, Z) \in T^{\Pi_R}\uparrow^{i}$ where $A.r \in \rho_L \cup \rho_M$.

Inductive Step: Suppose $m(A, r, Z) \in T^{\Pi} \uparrow^{k}$. $m(A, r, Z) \in T^{\Pi}\uparrow^{k} \iff m(A, r, Z) : -b_1, \ldots, b_n \in SP(\Pi)^{INST}$ and $b_i \in T^{\Pi}\uparrow^{k-1}$; where $SP(\Pi)^{INST}$ is the ground instantiation of $SP(\Pi)$ and each b_i is a ground predicate. By the definition of $SP(\Pi)^{INST}$, this is only possible if there exists role predicates $B_1 \ldots B_n$ such that b_i is an instantiation of B_i and $m(A, r, ?X) : -B_1, \ldots, B_n \in SP(\Pi)$. Let ρ_i denote the role defined by the predicate B_i. To show that $m(A, r, Z) \in T^{\Pi_R}\uparrow^{k}$, it is sufficient to show that $\forall i.\rho_i \in \rho_L \cup \rho_M$, since b_i is in $T^{\Pi}\uparrow^{k-1}$.

Note, however, that if $m(A, r, ?X) : -B_1, \ldots, B_n \in SP(\Pi)$ then we must have $\rho_i \in \mathit{dlg}_{\Pi}(A.r)$. This follows immediately from the definitions of $SP(\Pi)$ and $\nabla(\rho, \Pi)$. Thus, for each i, $[\![C^{\text{del}}(\rho_i, \Pi)]\!]^{\Pi} = [\![C^{\text{del}}(A.r, \Pi)]\!]^{\Pi}$. From the definition of $\mathit{Obs}(R, \Pi)$ we have that $A.r \in \rho_L(R, \Pi) \cup \rho_M(R, \Pi) \iff \exists \rho_R \in R.C^{\text{del}}(A.r, \Pi) \sqcup \rho_R$. Thus, $\forall i.\rho_i \in \rho_L(R, \Pi) \cup \rho_M(R, \Pi)$.

Now, we consider the interpretation of metapolicy $C^{\text{del}}(\rho, \Pi)$. From the previous case, we have shown that $\forall \rho \in \rho_L \cup \rho_M.[\![\rho]\!]^{\Pi} = [\![\rho]\!]^{\Pi|_R}$. From the definition of $[\![C^{\text{del}}(\rho, \Pi)]\!]^{\Pi}$, if we are to show

$$[\![C^{\text{del}}(\rho, \Pi)]\!]^{\Pi} = [\![C^{\text{del}}(\rho, \Pi|_R)]\!]^{\Pi|_R}$$

then, it suffices to show that (1) $\forall \rho' \in C^{\text{del}}(\rho, \Pi).\rho' \in \rho_L \cup \rho_M$; and (2) $\forall \rho_1, \rho_2 \in \rho_L \cup \rho_M.\rho_2 \in \mathit{dlg}_{\Pi}(\rho_1) \iff \rho_2 \in \mathit{dlg}_{\Pi|_R}(\rho_1)$. The proof of (2) is trivial, since $\mathit{dlg}_{\Pi}(\rho_1)$ is a function of statements s where $\mathit{roledef}\, s = \rho_1$ and

$$\forall s.\mathit{roledef}\, s \in \rho_L \cup \rho_M \Rightarrow s \in \Pi \iff s \in \Pi|_R$$

To prove (1) we use the definition of $C^{\text{del}}(\rho, \Pi)$ as the $\lim_{i \to \infty} C_i^{\text{del}}(\rho, \Pi)$. We proceed by induction on i.

Base Case (i=0): $\rho \in \rho_L \cup \rho_M \Rightarrow C_0^{\text{del}}(\rho, \Pi) \subseteq \rho_L \cup \rho_M$. Since $C_0^{\text{del}}(\rho, \Pi) = \{\rho\}$ this is immediate.

Inductive Hypothesis: $\forall i < k.\rho \in \rho_L \cup \rho_M \Rightarrow C_i^{\text{del}}(\rho, \Pi) \subseteq \rho_L \cup \rho_M$.

Inductive Step: $\rho \in \rho_L \cup \rho_M$; consider $C_k^{\text{del}}(\rho, \Pi) = C_{k-1}^{\text{del}}(\rho, \Pi) \cup \{\rho_1, \ldots, \rho_n\}$. But, by the inductive hypothesis we have that $C_{k-1}^{\text{del}}(\rho, \Pi) \subseteq \rho_L \cup \rho_M$, $\forall \rho_i.\exists \rho' \in C_{k-1}^{\text{del}}(\rho, \Pi)$, such that, $\rho_i \in C_1^{\text{del}}(\rho', \Pi)$. Thus, by the inductive hypothesis $\rho_i \in \rho_L \cup \rho_M$. $\square$