Paralocks – Role-Based Information Flow Control and Beyond

Niklas Broberg
Department of Computer Science and Engineering,
University of Gothenburg
and Chalmers University of Technology, Sweden
d00nibro@chalmers.se

David Sands
Department of Computer Science and Engineering,
Chalmers University of Technology, Sweden
dave@chalmers.se

Abstract

This paper presents Paralocks, a language for building expressive but statically verifiable fine-grained information flow policies. Paralocks combine the expressive power of Flow Locks (Broberg & Sands, ESOP'06) with the ability to express policies involving runtime principles, roles (in the style of role-based access control), and relations (such as "acts-for" in discretionary access control). We illustrate the Paralocks policy language by giving a simple encoding of Myers and Liskov's Decentralized Label Model (DLM). Furthermore – and unlike the DLM – we provide an information flow semantics for full Paralock policies. Lastly we illustrate how Paralocks can be statically verified by providing a simple programming language incorporating Paralock policy specifications, and a static type system which soundly enforces information flow security according to the Paralock semantics.

Categories and Subject Descriptors D.3 [*PROGRAMMING LANGUAGES*]; F.3.1 [*LOGICS AND MEANINGS OF PROGRAMS*]: Specifying and Verifying and Reasoning about Programs

General Terms Security, Languages, Verification

1. Introduction

Issues of software security can be crudely categorized into three broad domains:

- *Access control* deals with security at the end points of a system, to verify that an entity is allowed to access the system, and to what extent.
- *Information flow control* deals with security *inside* a system, between the end points, to ensure that data in the system is handled in a way that agrees with the security policy of the system. This is the domain that is most interesting from a programming language point of view, since it deals with security during execution.
- *Encryption* deals with security *outside* a system, to ensure that data can be protected even outside the trusted system environment.

The problems involved in research on encryption are quite different from the other two domains, but unsurprisingly there are many similarities between problems that arise in the access control and information flow control domains. In particular, problems regarding policy specification and modelling of principal actors are quite similar, much due to the fact that these issues are not purely technical, but rather relate to the interface between the system and its users (implementor, admins). Thus, many ideas relevant in one domain are equally applicable to the other, at least on a high level.

In the access control domain there exists plenty of research regarding policy specification mechanisms. Such mechanisms have traditionally been categorized into two separate groups: Mandatory (or static) access control (MAC), where an outside administrator assigns static privileges to principals, and Discretionary access control (DAC), where principals themselves can grant and revoke privileges to and from other principals. A later addition to the family of models is Role-based access control (RBAC) [SCFY96], which has become very popular and has seen wide-spread adoption both commercially and academically.

On the information flow control side, there has been far less focus on policy specification. We surmise that this has a very natural cause. In access control, which deals with the interfaces to a system, policy specification is the one core issue and a prerequisite for any further aspects of security. Information flow control on the other hand is more naturally focused towards issues of semantic security with respect to a policy, and most research in the domain has been devoted in that direction.

Papers on information flow control issues typically fall into one of two categories where the policy mechanism used is concerned. In the first category we find those that use a simple model built around a lattice of principals or sets of principals, going back to Denning's early ground-breaking work. The other category is the research that builds on the Decentralised Label Model (DLM), which is today something of a flagship of information flow control through its implementation in JIF. These two categories can somewhat crudely be said to correspond to the MAC (static Denning-style lattice) and DAC (decentralised and discretionary) models.

Interestingly and perhaps surprisingly there has been almost no work on marrying a fundamentally role-based model to information flow control (the exceptions being [SHTZ06, BWW08] which are discussed further in Section 8), despite the massive attention RBAC has received in the access control domain, both commercially and academically. The use of roles in an information-flow setting is discussed in Section 2.

In this paper we present Paralocks (Section 3), a language for building expressive but statically verifiable fine-grained information flow policies. Paralocks is based on the simple and expressive idea of Flow Locks [BS06] extended with the ability to express policies modelling roles (in the style of RBAC) and run-time principals.

The extension (*para*meterised *locks*) turns out to provide much more than just the ability to model roles: we show (Section 4) how relations such as delegation in discretionary access control can be represented by policies, and use this to give a sound and complete encoding of the *Decentralised Label Model* [ML97].

Unlike the *Decentralised Label Model* we also provide an information flow semantics for Paralocks (Section 5). This defines what it means for a program (whose state components are labelled with policies) to be secure.

As an illustration of how paralocks can be integrated into a programming language we give (Section 6) an example of a small programming language with a paralock type system for which we show that well-typed programs satisfy the semantic information flow condition.

Finally (Section 7) we outline a logically natural extension to the paralocks policy language to include recursively specified locks (DATALOG rules).

Our aim with this work is really two-fold. On the one hand we present the policy specification language Paralocks. We show how Paralocks naturally models roles, but also actors, groups and general relationships in a simple and structured way. On the other hand, we also present Paralocks as a very general framework for information flow control. This aspect lets us use Paralocks to reason about and give meaning to other mechanisms both current and future. Paralocks thus serves as a platform that can greatly simplify further research into various aspects of information flow control, such as specialised policy specification languages, and the relationship between information flow control and programming language design.

2. Roles and Information Flow

Roles are a natural concept in an organisational structure and are just as naturally tied to information flow controls as to access control. Consider a department consisting of managers, personnel and sales. These roles form a hierarchy as illustrated in the Hasse diagram below:

```
          Manager
          /     \
   Personnel    Sales
```

In role-based access control each role represents a set of users (later we will use the neutral term *actors*) endowed with a set of permissions. The hierarchy illustrated in the figure (roles + hierarchies are referred to as RBAC1 in the RBAC96 model [FSG+01]) represents the intention that the permissions granted to higher roles subsume those granted to lower roles.

Let us suppose that we take an information flow perspective on roles and we assume data is labelled with a role, representing the permission to gain information about that data. Then role-based information flow control would simply be the constraint that information may only flow upwards in the hierarchy. This is simply the Denning lattice-based model [Den76] with a relaxation on the requirements that the hierarchy forms a complete lattice.

In this setting the assignment of users to roles is of little direct concern from an information flow perspective, since users do not possess their own data, and are defined purely by the roles to which they are assigned. Inputs and outputs of the system would then be bound to roles, and some external mechanism would mediate the connection between roles and users.

However, if we admit the possibility of personal data then the information flow perspective becomes considerably richer. For example, if we had I/O channels directly to users then we would have an information flow problem with a dynamic policy: information flows to and from a given actor would depend on her current role.

Consider another scenario involving personal data: an auction site managing sealed-bid auctions for an *a priori* unspecified number of users. In such a scenario the roles of seller and bidder, respectively, immediately spring to mind. Other constraints on the auction influence the intended information flows:

- the seller can set a reserve price which is initially only visible to the seller;
- bidders provide sealed bids and can see their own bid but cannot see each others' bids;
- bidders learn of the winning bid, but only at the end of the auction;
- if the reserve price is not met then there is no winning bid;
- sellers cannot also be bidders for the same item.

In summary, to verify that code managing such auctions is well behaved raises a number of general challenges from an information flow perspective:

1. We need to model dynamic actors – actors whose concrete identity is not known or may not exist until runtime.
2. The data associated with a role (e.g. the bids) belongs to the actor and not the role (because bidders should not be able to see all bids - only their own bid).
3. Permissions associated with roles are assigned dynamically (in this example, the ability to read a winning bid is only granted after the auction is complete).
4. Declassification is required: the winning bid (or its absence) provides partial information about the secret reserve price of the seller.
5. We must be able to impose role constraints (a la RBAC2) to ensure that the seller cannot become a bidder on the same item.

In the next section we develop a policy language which is aimed at meeting these challenges. The policy language is built on top of *flow locks*, a versatile policy language for dynamically changing information-flow policies. The extension in question is motivated by the addition of *roles*. As we will show in the subsequent section (4), the extension turns out to provide considerably more than just the ability to represent roles.

3. Flow Locks and Roles

Flow locks are a simple security policy specification mechanism. Flow locks themselves are "policy neutral" – they do not presuppose any particular labels, information flow levels, or fixed hierarchy. The core idea is to logically specify the conditions under which a given actor in the system may gain information about some data. We use the phrase "gain information" rather than "access" at this point to stress that this is an information flow notion rather than an access control one. The conditions are specified using so called *locks*, which are boolean variables that may be manipulated by the execution of the program. A policy is then a set of logical clauses of the form $\Sigma \Rightarrow a$ (so called Horn clauses), where each clause specifies the conditions (Σ) under which data labelled with that policy may flow to actor a. Σ is a set of locks, which we interpret as a conjunction, i.e., for a to have access to the data then all locks in Σ must be true. The set of clauses is itself interpreted as a conjunction, so for an actor to have access it is enough that one clause allows it (a conjunction of implications is equivalent to a disjunction of the premises).

Consider a simplified form of the auction example in which we have two known buyers B_1 and B_2 and a single seller S and where the bidders may see each other's bids once they have placed their own. We associate two locks, bid_1 and bid_2 with the placing of bids by B_1 and B_2 respectively; bid_i will be assumed to become true once B_i has placed his bid. Then the policy for B_1's bid is

$$\{S; B_1; bid_2 \Rightarrow B_2\} .$$

This says that the B_1's bid may flow to S and B_1 unconditionally, and may flow to B_2 only when B_2 has placed a bid (as modelled by lock bid_2).

Using the terminology of flow locks, when a condition represented by a lock becomes true, we say that the lock is *opened*. Similarly when is becomes false we say that it is *closed*.

Consider a further example from [BS09] represented in Figure 1 which depicts three Denning-style information-flow lattices.

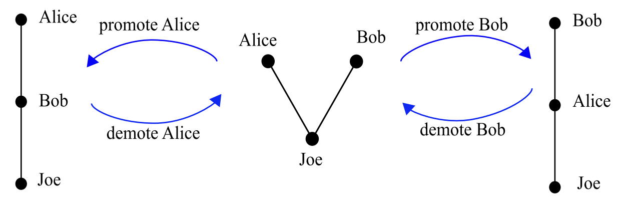

Figure 1. Example Dynamic Policy

In the leftmost lattice Alice is the top element. While Alice is "boss" all information may flow to her. If she is demoted, however, then the information flow lattice changes to the central figure. From there either Bob or Alice can be promoted to be the boss. Let us consider how to encode this intended scenario with flow locks. To represent this dynamic flow policy we begin, not surprisingly, by assuming three actors: *Alice*, *Bob*, and *Joe*. To model the transitions between policies we use two locks: *promoteA* and *promoteB*. The events of promotion and demotion are modelled by the respective opening and closing of these locks. When *promoteA* is open then Alice is boss. Closing *promoteA* (respectively, *promoteB*) corresponds to demoting Alice (resp. Bob).

To complete the picture we need to describe the corresponding policies for the data to be associated with Alice, Bob, and Joe. Joe is the simplest case, and his data has policy $\{Joe; Alice; Bob\}$ – i.e. it is readable by everyone at all times. Alice's data has policy $\{Alice; promoteB \Rightarrow Bob\}$ and Bob has the symmetric policy $\{Bob; promoteA \Rightarrow Alice\}$. For Bob this means that his data is readable by Alice only when Alice has been promoted. Note that if both locks are open then we have a situation not modelled in the figure: Alice and Bob become equivalent from an information flow perspective. If we want to rule this out we cannot do so using the policy on data, instead we must enforce this via an invariant property of the locks themselves.

The reason for keeping the guards as simple boolean flags rather than arbitrary logical expressions is that this makes it possible to mechanically check that programs conform to a flow lock policy using a type system. Ensuring that locks are open at appropriate times is an application-specific problem that can be seperately verified using a general purpose theorem prover, for example.

To track information flow in a program, data labelled with some policy p_1 is allowed to flow to a location with a different policy p_2, assuming that p_2 is *more restrictive* than p_1. This ordering of policies, which we write $p_1 \sqsubseteq p_2$, corresponds naturally to logical entailment when viewing policies as Horn clauses. In fact, it is easy to show that policies form a lattice, where the join and meet of two policies correspond to logical disjunction and conjunction respectively, and the partial ordering is logical implication as noted.

But whether a flow is permitted obviously also depends on the current lock state, to allow for e.g. declassification. To this end we express the comparison of policies in a specific context by *specialising* a policy p to a particular set of locks Σ that are known to be open. Formally this is defined for flow locks as

$$p(\Sigma) = \{\Delta \setminus \Sigma \Rightarrow x \mid \Delta \Rightarrow x \in p\}$$

and we allow data labelled with policy p_1 to flow to a location with policy p_2 provided that $p_1(\Sigma) \sqsubseteq p_2$, where Σ is the lock state in effect at the time of the flow.

3.1 Modelling roles

A naive approach to supporting roles with Flow locks could be to simply let the actors *be* the roles, and not have conventional actors at all. This would work with no changes to the current policy language, however as our examples in the previous section have shown, we often need to reason about both roles (or groups) *and* individual actors.

Thus we need a different approach to roles that retains the notion of an actor. Looking at what it means for some data to be accessible to a role R, the natural interpretation is that an actor may gain information about that data if the actor is a member of R. How do we express this as a flow lock policy? We need a lock that captures the condition that "a is a member of role R", which we henceforth write $R(a)$. But clearly the policy we want is not just for some specific actor a, but rather any actor x for which $R(x)$ holds. Logically we could easily write this as $\forall x.\, R(x) \implies x$. A role thus has a natural representation as a lock *family*, parametrised by actors.

To achieve this we propose two separate – though synergistic – extensions to the basic formulation of flow locks from [BS06]:

Parameterised Locks Locks which are parameterised over actors represent role memership. For example the role $Seller$ is represented as parameterised lock family, so if a is an actor then $Seller(a)$ is a lock which models a being a member of the seller role. Data labelled with the policy $\{Seller(a) \Rightarrow a\}$ is permitted to flow to a providing that a is a seller.

Actor Polymorphism To make parameterised locks practically useful we also need to be able to quantify over *all* actors, so that we could instead write the policy as

$$\{\forall x.\, Seller(x) \Rightarrow x\}$$

– meaning that data labelled with this policy may flow to *any* seller.

With this interpretation of roles, and these extensions to the policy specification language, we can easily formulate the policies from the examples in the previous section using flow locks. Let us then return to the challenges offered by those examples:

1. Actors whose concrete identity is not known until run-time can be handled by policies with actor polymorphism. As a simple example, the policy $\{\forall x.\, x\}$ is the most liberal policy, permitting its data to flow to any actor at all times. This does not require us to know the identity of all actors at policy creation time (as would be required using the original basic flow locks mechanism).
2. Fine grained policies at the level of individual actors combine easily with roles. For example, suppose we wish to generalise the scenario in Figure 1 to an organisation of 1000 employees – or a situation with an unknown number. Here we must combine a role (the boss) with the requirement that non-bosses cannot obtain information from each other (with the exception of Joe). The data of Joe would have policy $\forall x.\, x$ – it can flow to anyone. The data for any other individual a would have the policy $\{a; \forall x.\, Boss(x) \Rightarrow x\}$, which means that data labelled with this policy can flow to a and anyone who is a boss (at the time of the flow).
3. Permissions associated with roles are assigned dynamically by using standard (non-parameterised) locks. For example, the

largest bid might be stored in a variable with policy

$$\{\forall x.\{AuctionClosed, Bidder(x)\} \Rightarrow x\}$$

where the vanilla lock $AuctionClosed$ represents the property that the auction is complete and the reserve has been met. So in effect $AuctionClosed$ represents the condition under which the $Bidder$ role is assigned the permission to learn about the winning bid.

4. Declassification is inherited from the standard flow locks model. For example, the reserve price is available to the seller, but is declassified to bidders providing there is a winning bid:

$$\{Seller; \forall x.\{AuctionClosed, Bidder(x)\} \Rightarrow x\}$$

5. Role constraints – here the requirement that e.g. there is a single seller, and that seller and buyer cannot be the same actor – can be established by runtime invariants for flow locks. These can either be verified statically or enforced dynamically using a runtime representation of locks. In Section 7 we describe an extension which permits certain constraints on roles to be specified as part of the policy.

3.2 The Paralocks Policy Language

Now it is appropriate to summarise the policy language of this paper, and define its lattice structure. In summary, the policy language generalises flow locks policies with actor-parametrised locks, hence the name: *Paralocks*. The ordering on policies is based on a straightforward and natural logical interpretation of policies.

DEFINITION 3.1 (Paralock Policies).

- *Policies are built from* actor identifiers, *ranged over by* a, b, *etc. and* parameterised locks, *ranged over by* σ, σ' *etc. Each parameterised lock has a fixed arity,* $\text{arity}(\sigma) \geq 0$.
- *A* lock *is a term* $\sigma(a_1, \ldots, a_n)$, *where* $\text{arity}(\sigma) = n$. *Let* Σ, Σ' *range over sets of locks.*
- *A* clause c *is a term of the form* $\forall a_1, \ldots, a_n. \Sigma \Rightarrow a$.
- *A* policy p *is a set of clauses written* $\{ c_1; \ldots; c_n\}$.

We have already adopted a number of syntactic abbreviations in earlier examples: we write just σ instead of $\sigma()$ in the case that $\text{arity}(\sigma) = 0$. Similarly we drop the quantifier on clauses when there are no quantified variables. When the lock set Σ in a clause is empty, as in $\forall a_1, \ldots, a_n. \varnothing \Rightarrow a$ we write $\forall a_1, \ldots, a_n. a$. We will routinely write $\vec{a}$ to denote some sequence $a_1, \ldots, a_n$. Such a sequence will be treated as a set $\{a_1, \ldots, a_n\}$ when the context permits us to do so without ambiguity.

Policies have a natural reading as conjunctions of definite first-order Horn clauses. Each clause

$$\forall a_1, \ldots, a_n. \{ \sigma_1(\vec{b}_1); \ldots; \sigma_m(\vec{b}_m)\} \Rightarrow a$$

can be read as the Horn clause

$$\forall a_1, \ldots, a_n. (\sigma_1(\vec{b}_1) \wedge \cdots \wedge \sigma_m(\vec{b}_m)) \Rightarrow Flow(a)$$

where $Flow$ is a single unary predicate disjoint from the parameterised locks, representing the "may flow to" property.

Using this logical interpretation we obtain a natural lattice structure on policies, where the policy ordering ($\sqsubseteq$) on individual clauses is just logical entailment. Specifically, we define $p \sqsubseteq q$ whenever p, viewed as a first order formula, entails q. We will write $p \models q$ to denote this logical interpretation.

Following this natural interpretation we have the following definition:

DEFINITION 3.2 (Policy ordering). *Policy* p_1 *is* less restrictive than *policy* p_2, *written* $p_1 \sqsubseteq p_2$, *if* $\forall c_2 \in p_2. \exists c_1 \in p_1. c_1 \sqsubseteq c_2$, *where the ordering* $\sqsubseteq$ *on clauses is defined to be the least partial order (reflexive and transitive relation) satisfying the following:*

- *if* c_1 *and* c_2 *are equal up to (i) capture-free renaming of* $\forall$*-bound actors (ii) reordering of quantified actors and (iii) deletion of* $\forall$*-bound actors not occurring in the body of the clause, then* $c_1 \sqsubseteq c_2$;
- $\forall a_1, \ldots, a_n. \Sigma_1 \Rightarrow b \sqsubseteq \forall a_1, \ldots, a_n. \Sigma_2 \Rightarrow b$ *if* $\Sigma_1 \subseteq \Sigma_2$.
- $\forall a_0, a_1, \ldots, a_n. \Sigma \Rightarrow a \sqsubseteq \forall a_1, \ldots, a_n. ((\Sigma \Rightarrow a)[a_0 := b])$, *where* $[a_0 := b]$ *denotes the unconstrained substitution of* b *for* a_0.

We do not present a formal proof that this corresponds to the logical interpretation (in fact we did not spot the connection directly), but we note that clauses are equivalent to so-called *conjunctive queries* [CM77], and a policy thus a union of conjunctive queries. The ordering on clauses defined above can be seen as a construction of a *containment mapping* [Ull90]. The fact that $\forall c_2 \in p_2. \exists c_1 \in p_1. c_1 \sqsubseteq c_2$ is necessary and sufficient to check logical entailment of unions of conjunctive queries was established in [SY80].

At any point during program execution, the permitted flows will depend on the locks which are open at that point. To determine whether $p \sqsubseteq q$ in the context of some open locks Σ, we check the logical implication $\Sigma \wedge p \models q$. In the type system given in Section 6 we implement this check via the *specialisation* of policy p to a lock state Σ, written $p(\Sigma)$; we then check that $p(\Sigma) \sqsubseteq q$.

The meet operation on policies is simple to define as it corresponds exactly to conjunction of (sets of) Horn clauses. In our language, that means taking the union of the clauses of two policies, i.e. $p_1 \sqcap p_2 = p_1 \cup p_2$.

The join operation however is more tricky. Logically it corresponds to a best approximation of disjunction of Horn clauses, since in general (sets of) Horn clauses are not closed under disjunction. I.e. $p \sqcup q$ is the least policy such that $p \vee q \models p \sqcup q$. We can define the join directly as follows:

DEFINITION 3.3 (LUB). *In the following it is convenient to partition actor variables in to* $\forall$*-bound variables ranged over by* x, $\vec{y}$, *and free actor variables (i.e. actor constants) ranged over by* a *and* b. *We write* $\Sigma \Rightarrow b$ *to denote the policy* $\forall \vec{y}. \Sigma \Rightarrow b$ *where* $\vec{y}$ *are the* $\forall$*-bound variables of* $\Sigma \Rightarrow b$.

Let p *and* q *be policies. We will assume, without loss of generality, that all* $\forall$*-bound variables appearing in the head of any clause are named* x, *and that any other* $\forall$*-bound variables in any clause from* p *are distinct from the* $\forall$*-bound variables of* q.

Then we define

$$\begin{aligned} p \sqcup q = &\{\Sigma_p \cup \Sigma_q \Rightarrow x \mid \Sigma_p \Rightarrow x \in p; \Sigma_q \Rightarrow x \in q\} \\ &\cup \{\Sigma_p \cup \Sigma_q \Rightarrow a \mid \Sigma_p \Rightarrow a \in p; \Sigma_q \Rightarrow a \in q\} \\ &\cup \{\Sigma_p \cup (\Sigma_q[x := a]) \Rightarrow a \mid \Sigma_p \Rightarrow a \in p; \Sigma_q \Rightarrow x \in q\} \\ &\cup \{(\Sigma_p[x := a]) \cup \Sigma_q \Rightarrow a \mid \Sigma_p \Rightarrow x \in p; \Sigma_q \Rightarrow a \in q\} \end{aligned}$$

It can be shown that the set of paralock policies (quotiented by the equivalence relation generated from $\sqsubseteq$) form a complete lattice. We will not go into the proof here, but simply note the least (most liberal) policy $\bot = \{\forall x. x\}$ and the greatest (most restrictive) policy $\top = \{ \}$, which will be needed later.

The reader may have noticed that the policy language defined here can contain locks parameterised over more than a single actor. This gives us more expressive power than just roles. In the next section we motivate and illustrate this generalisation.

4. From Roles to Relations: Encoding the Decentralized Label Model

Using actor-indexed lock families we have shown how we can model roles along-side specific actors in a natural logical setting, and how the two can co-exist in the same program. In this section we will show how, using a natural generalisation, we can model policies where information flow can depend on *relations* between actors. Such relations are useful in the description of a decentralised discretionary security model.

The core components of a decentralised discretionary model is the concept of *ownership*, and an *acts-for* relationship (sometimes referred to as *delegation* or a *speaks-for* relation [LABW91]), where an actor a who acts for b enjoys the same rights as b. In particular if actor a owns some data then b has full access to that data if b acts for a. The condition under which b may access the data is thus that "b acts for a". Logically this is easily modelled with a binary relationship between actors, which in the flow locks setting would naturally correspond to a lock family with *two* parameters. The policy mentioned here could then be written as $\{a; \forall x.\, \mathit{ActsFor}(a, x) \Rightarrow x\}$.

Going from one to two parameters, or indeed to n-ary lock families, is a straight-forward generalisation. There are no additional technical difficulties involved, and we already have the mechanics for quantification in place. (We have no immediate examples of lock families with more than two parameters, but see no reason to exclude them.)

Typically, acts-for relationships are modelled with two other properties, namely reflexivity and transitivity. Each actor acts for himself. If a acts for b and b acts for c, then models typically assume implicitly that a acts for c as well. With Paralocks, properties like transitivity and reflexivity are not built in. Locks are just boolean variables with no additional predefined semantics attached to them. If we want a transitive property for a particular relation like $\mathit{ActsFor}$, we must handle this explicitly.

A naive attempt could be to try to handle this on the policy level, e.g. by specifying the policy as

$$\{a; \forall x.\, \mathit{ActsFor}(a, x) \Rightarrow x; \quad \forall x, y.\, \mathit{ActsFor}(a, x), \mathit{ActsFor}(x, y) \Rightarrow y\}$$

This is not a viable approach since the above policy only works for one step of transitivity; for full transitivity we would need to explicitly list the transitive closure, and this would be at best cumbersome, and impossible if we could not statically enumerate all actors. There are two routes to deal with this issue. The first is to extend the expressive power of the policy language to enable such global invariants to be expressed as part of the policy. This is explored in Section 7. For now we take a simpler route, and view transitivity not as a property of a policy, but rather an intended invariant on the set of locks open at any given time. This invariant can easily be maintained at runtime by suitably encapsulating lock manipulation operations.

So if we ensure that any program using a policy involving delegation maintains the transitivity property of $\mathit{ActsFor}$, then it is enough for the policy to be stated (as before) as simply

$$\{a; \forall x.\, \mathit{ActsFor}(a, x) \Rightarrow x\}.$$

4.1 Encoding the Decentralized Label Model

To show the flexibility of our model, we show how it can be used to encode the Decentralized Label Model (DLM) of Myers and Liskov [ML97] [1].

The core component of the DLM is the *label*. Data is decorated with labels that govern how that data may be used. A label L specifies the *owners* of some data, written $\mathit{owners}(L)$, and for each owner the set of *readers* allowed by that owner, $\mathit{readers}(L, o)$. The intuition behind the owners is that data, at its origins, has a single owner who specifies its readers. The label on a piece of data reflects the various potential origins of the information in that data.

[1] In the earlier flow locks work [BS06] we sketched a possible DLM encoding, but the encoding there required all principals to be known statically, so that all relations between principles could be "hard wired" into the policy.

The decentralisation relates to the readers. Each owner can independently specify who they consider trusted to view the data. The *effective readers* of some data are those for whom all the owners have agreed may read it, i.e. the intersection of the separate reader sets for all owners. A label for some data may look like

$$\{o_1 : r_1, r_2 \,;\; o_2 : r_2, r_3\}$$

where o_1 and o_2 are owners and r_1, r_2 and r_3 are readers. Such data might be obtained by combining data from o_1 and o_2 in some way. The effective readers in this example is just r_2.

A label L_2 is said to be more restrictive than label L_1, written $L_1 \sqsubseteq_{DLM} L_2$, if it has at least the same owners, and each of those owners list fewer potential readers. Formally it is defined [ML97] as

$$\begin{aligned} L_1 \sqsubseteq_{DLM} L_2 = \mathit{owners}(L_1) \subseteq \mathit{owners}(L_2) \wedge \\ \forall o \in \mathit{owners}(L_1).\, \mathit{readers}(L_1, o) \supseteq \mathit{readers}(L_2, o) \end{aligned}$$

Data may be *relabeled* in two ways, through an assignment:

- Data with label L_1 can always be assigned to a storage location (a container) with label L_2 if L_2 is more restrictive than L_1, i.e. $L_1 \sqsubseteq L_2$.
- Data can be *declassified* by adding more readers for a given owner. In the DLM this can be done freely providing that the current process runs on behalf of the owner in question.

Apart from labels, there is one other important component to the DLM, namely the *principal hierarchy* and its associated acts-for relationship. The DLM lets principals represent both individual users and other notions like roles and groups, and membership for a user in a role can thus be modelled by letting the user act for that role. The acts-for relationship is transitive and reflexive.

This has two effects on the security of a program. First, if a acts for b and b is listed as a reader in a label, then a is also implicitly a reader. Second, if a piece of code runs on behalf of a, then it also implicitly runs on behalf of b, so code running on behalf of a may conduct declassifications in b's name.

To encode the DLM using Paralocks, we need to represent a number of things explicitly that are implicit in the DLM. The first of these is the acts-for relationship, which we've already discussed how to model earlier in this section. If the principal hierarchy states that b acts for a, then we expect the $\mathit{ActsFor}(a, b)$ lock to be open. We can account for changes to the hierarchy during execution by opening or closing the appropriate locks. We expect a concrete semantics to maintain the invariants for transitivity and reflexivity for the $\mathit{ActsFor}$ relationship as previously discussed.

Second, to account for declassification being possible only when the process runs with the authority of the owner of the declassified data, we need a lock family $\mathit{RunsFor}(a)$. We expect the appropriate locks to be open for those actors for whom the code runs. Further, we also expect the invariant that whenever $\mathit{ActsFor}(a, b)$ and $\mathit{RunsFor}(b)$ is open then $\mathit{RunsFor}(a)$ is also open, again making the implicit relationship explicit.

Third, since Paralocks take the perspective of the reader, as opposed to the owner as in in DLM, the policy needs to be explicit about the potential future readers to whom the data may be declassified. With respect to a given owner, we can freely add new readers as long as the code executes with that owner's authority. We can thus model the label $\{o :\}$, i.e. data owned by o with no added readers, with the policy

$$\{\forall x.\, \mathit{RunsFor}(o) \Rightarrow x\}$$

The intuition here is that in code running with o's authority, this data may be declassified to any actor x. [2] Adding a reader to the above policy, we get $\{o : r\}$, which we would represent as

$$\{\forall y.\, \mathit{ActsFor}(r, y) \Rightarrow y; \forall x.\, \mathit{RunsFor}(o) \Rightarrow x\}$$

The first clause here corresponds to the reader r. By reflexivity we will always have $\mathit{ActsFor}(r, r)$ open, and hence r, and anyone else who acts for r, will be able to read data labelled with this policy.

To handle the general case of the encoding we need to deal with the case of a *potential reader* (a reader who is a reader for one but not all owners). For these readers we need to consider the owners who do *not* permit r to read the data.

DEFINITION 4.1 (Label Encoding). *Suppose that r is a (potential or effective) reader for some label L, and O is a subset of owners for L. We say that the pair (O, r) is a* conflict pair *for label L if*

$$O = \{o \mid o \in \mathit{owners}(L), r \notin \mathit{readers}(L, o)\}\,.$$

Intuitively, O are the owners who have not permitted r to read data labelled L.

Now we can define the general encoding of Labels as policies $[\![\cdot]\!] : \mathit{Label} \to \mathit{Policy}$ by

$$\begin{aligned}[\![L]\!] = {} & \{\forall x.\, \{\mathit{RunsFor}(o) \mid o \in \mathit{owners}(L)\} \Rightarrow x\} \\ \cup {} & \{\forall y.\, \mathit{RunsFor}(o_1), \ldots, \mathit{RunsFor}(o_n), \mathit{ActsFor}(r, y) \Rightarrow y \\ & \qquad \mid (\{o_1, \ldots, o_n\}, r) \text{ is a conflict pair for } L\}\end{aligned}$$

The first clause in the definition of $[\![L]\!]$ says that data can be declassified to anyone providing it is in a context which runs with the authority of all owners. Otherwise a potential reader r (or anyone who acts for r) may read providing it does so in a context which runs with the authority of those owners who did not grant explicit access to r.

As an example, consider the encoding of the empty label:

$$[\![\{\ \}]\!] = \{\forall x.\, \{\} \Rightarrow x\} \cup \{\} = \{\forall x.\, x\}$$

The empty label has no owners, so implicitly anyone can read data with that label – as expressed explicitly in the flow locks encoding.

When combining labels from different data sources, the DLM simply performs the union of the respective owner policies, leaving the effective reader set implicit as the intersection of all readers. In our encoding the difference between effective and potential readers is rendered more explicit. Consider combining the two policies representing $\{o_1 : r_1, r_2\}$ and $\{o_2 : r_2, r_3\}$, which are

$$\begin{aligned}&[\![\{o_1 : r_1, r_2\}]\!] = \\ &\quad \{\forall x.\, \mathit{RunsFor}(o_1) \Rightarrow x; \\ &\quad\ \forall y.\, \mathit{ActsFor}(r_1, y) \Rightarrow y; \forall y.\, \mathit{ActsFor}(r_2, y) \Rightarrow y\} \\ &[\![\{o_2 : r_2, r_3\}]\!] = \\ &\quad \{\forall x.\, \mathit{RunsFor}(o_2) \Rightarrow x \\ &\quad\ \forall y.\, \mathit{ActsFor}(r_2, y) \Rightarrow y; \forall y.\, \mathit{ActsFor}(r_3, y) \Rightarrow y\}\end{aligned}$$

[2] Note that in a programming language enforcing a DLM (such as JFlow/Jif [Mye99, MZZ+06]) one might want to additionally constrain that declassification occurs at explicitly declared places in the code. This is easily modelled using regular flow locks by associating a *Declassify* lock with the portion of code which is designated as a declassification. This, however, is not part of the DLM model.

we get $[\![\{o_1 : r_1, r_2\} \sqcup_{DLM} \{o_2 : r_2, r_3\}]\!]$

$$\begin{aligned}&= [\![\{o_1 : r_1, r_2\ ;\ o_2 : r_2, r_3\}]\!] \\ &= \{\forall x.\, \mathit{RunsFor}(o_1), \mathit{RunsFor}(o_2) \Rightarrow x; \\ &\quad\ \forall y.\, \mathit{ActsFor}(r_2, y) \Rightarrow y; \\ &\quad\ \forall y.\, \mathit{RunsFor}(o_2), \mathit{ActsFor}(r_1, y) \Rightarrow y; \\ &\quad\ \forall y.\, \mathit{RunsFor}(o_1), \mathit{ActsFor}(r_3, y) \Rightarrow y\} \\ &= [\![\{o_1 : r_1, r_2\}]\!] \sqcup [\![\{o_2 : r_2, r_3\}]\!]\end{aligned}$$

Finally we can show that the lattice of labels in the DLM is a sublattice of the Paralocks policy lattice:

THEOREM 4.1. $L_1 \sqsubseteq_{DLM} L_2$ if and only if $[\![L_1]\!] \sqsubseteq [\![L_2]\!]$. Further, $[\![L_1 \sqcup_{DLM} L_2]\!] = [\![L_1]\!] \sqcup [\![L_2]\!]$, and similarly for $\sqcap$.

The proof of these properties is given in the extended version of the paper. The relationship between $\sqsubseteq$ and $\sqsubseteq_{DLM}$ amounts to saying that the encoding is sound and complete with respect to the DLM rule for relabelling data.

What we have given here is an encoding of the DLM policy specification language only. One might expect to see a deeper comparison, in which we also compare the impact of the two on the security of programs, i.e. the formal semantic security definitions. The problem is that the DLM, or more accurately its implementation in JIF, does not have a formal semantic security model. There exist models for subsets or restricted scenarios for DLM, but it has never been covered in full. But with our encoding here, we are actually able to do just that, to provide a semantic security model for programs that use the DLM for their information flow control. Our full semantic model will be presented in the next section.

5. Paralock Security

In previous work [BS09] we have developed a simple and accurate context-sensitive security model for flow locks based on understanding when an attacker's knowledge about initial data values is permitted to increase, developed as a generalisation of the simple *gradual release* definition [AS07].

The semantic model developed in this section is an extension of the simple flow locks model from [BS09]. The difference is that we must handle both runtime actor allocation and runtime querying of the lock state, both of which may be sources of information flow.

5.1 Computation Model

We assume an imperative computation model – a labelled transition system – involving commands and states, but the definition is otherwise not specific to a particular programming language. We assume transitions of the form $\langle c, S\rangle \xrightarrow{\ell} \langle c', S'\rangle$ where c is a program and S is the program state. We assume that the semantics signals any flow of information, i.e. changes to the state, using labels l, where l is either a distinguished silent output τ (when there is no state update), or a value u corresponding to the value of the updated part of the state. So for example a simple assignment $x := 42$ would generate a $\xrightarrow{42}$ transition. We further assume that the state includes at least the following three components:

- A memory, i.e. a mapping from locations to the values they contain. We denote the memory of state S by $\mathbf{Mem}(S)$, and range over memories using variables M, N.
- A lock state, which is the set of all locks currently open. We denote the lock state of state S by $\mathbf{LS}(S)$, and use $\mathbf{\Sigma}, \mathbf{\Delta}$ to range over lock states.
- An actor mapping, keeping track of the concrete run-time representation of actors that the actor variables in the program rep-

resent. We denote the actor mapping part of state S by $\mathbf{Act}(S)$ use $\mathbf{\Lambda}$ to range over actor mappings.

Just as with program variables, actors have concrete representations at runtime, which differ from their representations in the program code. This is so we can handle e.g. dynamic creation of actors in a loop, where the same actor variable name is reused for a new actor each time around the loop. We call the runtime representations *concrete actors*, as opposed to the *abstract actors* (actor identifiers) found in the program code and policies.

As a consequence, since locks can take actors as arguments, at runtime locks will be parametrised by concrete actor representations. We refer to a lock with concrete actor parameters as a *concrete lock*. The lock state component of the state consists of the set of concrete locks currently open.

For both actors and lock sets we adopt the convention to use bold face identifiers when denoting concrete entities. For instance, Σ would represent a set of abstract locks in e.g. a policy stated in the program, while $\mathbf{\Sigma}$ ranges over sets of concrete locks. For actors, Λ ranges over sets of abstract actors, while (with a slight abuse of our convention) $\mathbf{\Lambda}$ will denote an actor mapping, and hence $\mathbf{\Lambda}(a)$ denotes the concrete actor corresponding to abstract actor a. We will also apply actor mappings to sets of abstract actors and to abstract locks and lock sets; the effect in each case is to replace each abstract actor with the corresponding concrete one.

One other important thing to realise is that since actors and locks have runtime representations, and can be manipulated and queried at runtime, they are subject to the same possibilities for information flows as the memory. This means that to ensure that all information flows are properly specified and tracked, locks and actors must have policies too, to govern how they may be used in a program. For a given state component t we write $pol(t)$ to denote the policy of t.

5.2 Validating flows

To ensure correct information flow in a program, all flows must be validated at each possible "level" that data can flow to. This is not specific to our setting, but a very general statement regarding information flow control. Each of these levels can be thought of as a potential attacker. For each such attacker, we must ensure that the attacker does not learn more than intended about the initial data.

The way to do this is, for each possible attacker, to split the state into a *high* and a *low* portion – the low portion being the part directly visible to the attacker. The security goal is to ensure that the attacker, by observing the low part, does not learn more than intended about the *high* part of the state. For standard noninterference the goal is that the attacker learns nothing. For gradual release [AS07] the goal is to ensure that nothing is learned for the observations that are not labelled as declassifications.

Since flow locks permit fine-grained flows where data can be effectively declassifed to an actor in a series of steps, each removing one condition (i.e. lock) that needs to be fulfilled, our "levels" need to account for both actors and lock sets. We thus define an attacker A to be an actor paired with a set of locks which we denote the *capability* of that attacker. The intuition is that an attacker $A = (\mathbf{a}, \mathbf{\Sigma})$ may see any data guarded from actor $\mathbf{a}$ by at most $\mathbf{\Sigma}$. Formally,

$$A \in Actors \times \mathcal{P}(Locks)$$

We write $\mathbf{Cap}(A)$ for the capability of A. Note in particular that attackers observe concrete things at runtime, so they represent concrete actors with concrete capability sets.

To formulate security in a "knowledge evolution" style, following [AS07, BS09], we first need a number of auxiliary definitions.

A *trace* is a sequence of labels denoting changes to the state. An A-*observable trace* is a trace where we mask out changes to pieces of the state that the attacker A cannot see. We say that an attacker $(\mathbf{a}, \mathbf{\Sigma})$ *can see* some part of the state with policy p iff $p \sqsubseteq \{\mathbf{\Sigma} \Rightarrow \mathbf{a}\}$. A transition is *visible to* A if A can see the portion of the state involved in the change. We write $\langle c, S\rangle \xrightarrow{u}_A \langle c', S'\rangle$ when $\langle c, S\rangle \xrightarrow{u} \langle c', S'\rangle$ and the transition is visible to A, and

$$\langle c, S\rangle \overset{\vec{u}}{\Longrightarrow}_A \langle c', S'\rangle$$

when there exists a sequence $\vec{\ell}$ of labelled transitions between the respective configurations, where the projection of $\vec{\ell}$ to the non-silent A-visible transitions is equal to $\vec{u}$. We sometimes omit result configurations if we only care about the output of a program, as in $\langle c, S\rangle \overset{\vec{u}}{\Longrightarrow}_A$. Note that the series of execution steps generating a trace need not be maximal, so the set of all A-observable traces of a given program-state pair for a given attacker A is prefix closed.

An A-*low state* is a projection of a state to exactly those parts visible to attacker A. Two states are A-*equivalent*, written $S \sim_A T$, if their A-low projections are equal.

With these definitions in hand, we can define the notion of *attacker knowledge* as follows:

DEFINITION 5.1 (Attacker knowledge). *The knowledge an attacker has of the starting memory after observing trace $\vec{u}$ of program c with a starting state who's A-low projection is L is*

$$\mathbf{k}_A(\vec{u}, c, L) = \left\{ S \mid S \sim_A L, \langle c, S\rangle \overset{\vec{u}}{\Longrightarrow}_A \right\}$$

i.e., the set of all possible starting states that might lead to that trace.

Note that knowledge grows (uncertainty decreases) during execution, so we always have that $\mathbf{k}_A(\vec{u}u, c, L) \subseteq \mathbf{k}_A(\vec{u}, c, L)$.

5.3 Security

To validate that all information flows in a program are secure according to the stated policies, each output must be examined in the context it takes place, which in our flow locks setting means the lock state in effect at the time of the output. Consider for example the simple program $x := y$, where $pol(x) = \{a\}$ and $pol(y) = \{\sigma \Rightarrow a\}$. Clearly this program is insecure in isolation, since the policy on x is less restrictive than that on y, but it would be secure providing that σ was already open.

To help with our definition, we first define the notion of an A-*observable run* of a program to be a non-empty A-observable trace of the program, paired with the lockstate in which the last output of that trace takes place. We formally define the set of all A-observable runs that could arise from a given program c starting in a state whose A-low projection is L, as

DEFINITION 5.2 (A-observable run).

$$Run_A(c, L) = \left\{ (\vec{u}u, \mathbf{LS}(S')) \mid S \sim_A L, \langle c, S\rangle \overset{\vec{u}}{\Longrightarrow}_A \langle c', S'\rangle \xrightarrow{u}_A \right\}$$

Now, for a given attacker, representing a particular split of the state into high and low portions, who observes an output, the requirement is that this output may not signify a data flow from "high" to "low" portions of the state, unless the lock state permits such flows. Note that a single attacker is a very course-grained representation of security, as it is *only* able to distinguish between "high" and "low", but no nuances. As a consequence, *any* lockstate that would allow *some* flow from high to low will do. The split of high and low depends on the capability of the attacker, so for an attacker A we have that some lockstate $\mathbf{\Sigma}$ allows flows from high to low as long as $\mathbf{\Sigma} \not\subseteq \mathbf{Cap}(A)$. If $\mathbf{\Sigma} \subseteq \mathbf{Cap}(A)$ then the only flows that are allowed fall completely inside the parts of the state that A considers low.

The fine granularity is obtained by quantifying over all possible such attackers, since for any bad flow there must exist an attacker for which the flow is from “high” to “low”, but without a permissive enough lockstate.

Our formal definition of top-level security for a program, denoted $\mathbf{PLS}(c)$, can then be defined in terms of runs as follows:

DEFINITION 5.3 (Paralock security). *A program is said to be* paralock secure*, written* $\mathbf{PLS}(c)$*, if for all attackers* A*, for all* A*-low states* L*, for all runs* $(\vec{u}u, \mathbf{\Delta}) \in Run_A(c, L)$ *we have that if* $\mathbf{\Delta} \subseteq \mathbf{Cap}(A)$ *then*

$$\mathbf{k}_A(\vec{u}u, c, L) = \mathbf{k}_A(\vec{u}, c, L)$$

Informally, if the lock state at the time of the update would not allow any flows from “high” to “low” portions of the state, then no knowledge may be gained about the initial state.

In practice we also need a generalised definition which accounts for subprograms that are secure in the context they appear, where “context” here means the actors which exist and the locks which are open. For space reasons we omit this generalisation here, but note that it is needed in the proof for the type system presented in the next section. The generalisation can be found in the appendix.

The above definition of security is termination sensitive – a program is insecure if high data can influence termination behaviour. We get a weaker but more easily verified *termination insensitive* version following the formulation from [AHSS08] as follows:

DEFINITION 5.4 (Termination-insensitive security). *A program* c *is said to be termination insensitive secure, written* $\mathbf{PLS_{TI}}(c)$*, if for all attackers* A*, for all* A*-low states* L*, for all pairs of runs which differ only at the last output* $(\vec{u}u, \mathbf{\Delta}), (\vec{u}u', \mathbf{\Delta}') \in Run_A(c, L)$ *we have that if* $\mathbf{\Delta} \subseteq \mathbf{Cap}(A)$ *then*

$$\mathbf{k}_A(\vec{u}u, c, L) = \mathbf{k}_A(\vec{u}u', c, L)$$

Here we do allow some knowledge to be gained by observing the next output, but only by the fact that there is an output in the first place.

6. Enforcement: An Example Paralocks Type System

In this section we give an example of how Paralocks can be combined with a concrete programming language, and present a type system which guarantees that well-typed programs are (termination insensitive) paralock secure. The underlying language we present is as simple as possible while still using the full expressive power of Paralocks, to focus on the interesting parts of the interaction.

Expressions: $e \ ::= \ n \mid x[\vec{a}] \mid e \oplus e$

Commands:
$c \ ::= \ x[\vec{a}] := e \mid$ `if` e `then` c `else` $c \mid$ `while` (e) $c \mid$ `skip`
$\mid c_1; c_2 \mid$ `open` $\sigma(\vec{a}) \mid$ `close` $\sigma(\vec{a}) \mid$ `newactor` a `in` c
$\mid$ `when` $\sigma(\vec{a})$ `do` c `else` $c \mid$ `forall` $\sigma(\vec{a})$ `do` c

Internal Commands: $c ::=$ `for` $\sigma(\vec{a})$ `in` $\mathbf{\Sigma}$ `do` c

Figure 2. Example language syntax

The language, found in Figure 2, is at its core a sequential imperative language, with assignments, conditionals and loops. Data sinks and sources are kept abstract and are uniformly represented as references, with each reference having an attached policy. For simplicity the only basic type is the integers. The internal command (`for`) is not part of the surface syntax and only arises in the operational semantics.

To manipulate locks we introduce the commands `open` $\sigma(\vec{a})$ and `close` $\sigma(\vec{a})$. These are the only commands in the language that can change the lock state component of the state. Unlike in the basic flow locks language, locks here will have runtime representations and can carry information, so the runtime use of locks will also be governed by policies. The `when` command is a conditional which queries the the state of a particular lock.

New actors can be introduced dynamically using the `newactor` a command, which generates a fresh concrete actor and brings a new actor variable a into scope for the enclosed subcomputation. Note that this could for instance be placed inside a loop, so the same variable name introduced by the same `newactor` command can represent many different concrete actors during execution.

In order to keep the language and in particular the type system simple, actors are not first class entities. To regain some of the lost expressive power from this choice, we reuse lock families as a sort of storage for actors. A lock family can be viewed as a “named collection of actors”, and to access the contents of such a collection we introduce the `forall` command, which loops over all open locks in some family, bringing the relevant actors into scope in the loop body for each iteration. We assume that the order in which locks are looped over is deterministic.

The creation and use of actors may also be a conduit for information flow at runtime, so like references and locks we could require actor variables to have policies too. For simplicity though, we assume that all actors introduced by `newactor` commands are public, i.e. with a policy $\{\forall x.\, x\}$. Actor variables bound by a `forall` command will carry information about the lock family used in the loop, so we assume they inherit the policy of that lock.

Regarding policies, it is important to note that the runtime policies on runtime entities will talk about concrete actors and locks, while in the program code the policies will mention abstract entities. We have no explicit declaration of references in the language, instead we assume that they are globally available. But since actor variables are *not* globally defined, this has the effect that policies on references (and locks) cannot contain free actors, as that could lead to name capture problems. In many settings this would be too restrictive, since it would preclude actor-specific data.

To enable actor-specific data while avoiding all the extra machinery that would have been needed to track scoping and name-capture problems for policies, we instead make this explicit at the top level by having actor-parametrised *families* of references. For full flexibility we allow any number of parameters on a family of references, just as with locks.

Locks are also globally available, and may have actor parameters. However, for simplicity we do not allow the policies on lock families to mention the actor parameters, and thus may not contain any free actor variables. In other words, for a family of references we could have different actors having access to each individual reference, e.g. $pol(x[a]) = \{a\}$, whereas for families of locks we only allow a single policy for the entire family, e.g. $pol(\sigma) = \{\forall x.\, \sigma(x) \Rightarrow x\}$.

With all this in place, there is no need for any control that actors in policies refer to the proper runtime actors, since they cannot appear free in policies.

To illustrate these language features consider a simple sealed-bid auction scenario. For example, if we wanted a ’bid’ variable for each bidder in a sealed-bid auction, we could model that with a family of references $\mathsf{bid}[a]$, parametrised by actors. Policies on such families can then use the actor parameter, so we could have

$$pol(bid[a]) = \{a; \forall x.\, \mathsf{AuctionClosed} \Rightarrow x\}$$

where the policy on the individual references in the ’bid’ family depends on the actor in question. As an example, the code representing the registration of a new bidder might be written:

```
newactor b in
  open Bidder(b)
  bid[b] := getBid
```

where we assume that `getBid` is an input channel from the actor in question, represented as a reference. The policy on the reference `bid[b]` would be $\{b; \forall x. \{\mathsf{Bidder}(x), \mathsf{AuctionClosed}\} \Rightarrow x\}$, stating that all bidders can gain information about this bid once the auction is completed.

The code fragment for concluding the auction and publishing the winning bid (the first of the largest bids) could then be written

```
1  maxBid := 0
2  forall Bidder(x) do
3    if bid[x] >= maxBid then
4      maxBid := bid[x]
5      forall Winner(y) do close Winner(y)
6      open Winner(x)
7    else skip
8  open AuctionClosed
```

To be able to compute the maximum bid before the auction is marked as closed (as in this example) we would give `maxBid` the policy $\{\forall x. \{\mathsf{Bidder}(x), \mathsf{AuctionClosed}\} \Rightarrow x\}$. We use a separate lock family to denote the winning actor, and by (line 5) closing all previous winners and then opening the lock for the new winner, we are assured that we only ever have (at most) one winner. We could then loop over all actors a for whom the $\mathsf{Winner}(a)$ lock is open, to do specific things relating to the winner.

Again we stress that while some of the language design choices here are unorthodox, this is just a consequence of keeping the language and type system relatively small. In a more realistic programming language incorporating Paralocks, there are a number of other language design considerations. Supporting first-class dynamic actors would be a more natural route in a richer language, and this would be naturally supported in the type system using singleton types. From the expressiveness viewpoint support for policies not known until runtime (cf. DLM runtime labels [ZM07]) could well prove useful, but would require more language features to enable static checking. However, the issues involved there are largely problems of *enforcement*. While interesting in their own right, they are orthogonal to the core issues of this work, namely the Paralocks policy specification language and its associated definition of security.

6.1 Operational Semantics

The operational semantics of our example language can be found in Figure 3. Transitions occur between configurations of the form $\langle c, S\rangle$, where c is the command and S is the program state. This state is a triplet of an actor mapping ($\mathbf{Act}(S)$), a lock state ($\mathbf{LS}(S)$) and a memory ($\mathbf{Mem}(S)$). For simplicity we lift updates on individual components to the full state, so for instance we write e.g. $S[x \mapsto v]$ to update the value of a variable in the memory, or $S \cup \sigma$ to add an open lock to the lock state. Since the three components have disjoint domains there should be no risk for confusion.

Apart from the labels on transitions, there should be no surprises in the rules for the ordinary imperative constructs. Regarding `open` and `close`, the only thing of note is that we need to map actor variables in locks to their concrete representations before updating the lock state.

The `when` command is very similar to the standard `if`, the only difference being that `when` queries the lock state instead of the memory.

The `newactor` command generates a fresh concrete actor representation and binds it to the variable name given. Since we assume all actors bound this way are public, we don't need to care about the particulars of the generation scheme. Syntactically the variable is scoped, but in the semantics we don't bother to remove it once we leave the scope, instead we rely on the type system to ensure that there can be no accidental capture.

$$\langle n, S\rangle \Downarrow n \qquad \frac{\langle e_1, S\rangle \Downarrow v_1 \quad \langle e_2, S\rangle \Downarrow v_2}{\langle e_1 \oplus e_2, S\rangle \Downarrow v_1 \oplus v_2}$$

$$\left.\begin{array}{c}
\langle x[\vec{a}], S\rangle \Downarrow \mathbf{Mem}(S)[x[\vec{\mathbf{a}}]] \\[1ex]
\langle \mathtt{open}\ \sigma(\vec{a}), S\rangle \xrightarrow{\mathtt{open}\ \sigma(\vec{\mathbf{a}})} \langle \mathtt{skip}, S \cup \{\sigma(\vec{\mathbf{a}})\}\rangle \\[1ex]
\langle \mathtt{close}\ \sigma(\vec{a}), S\rangle \xrightarrow{\mathtt{open}\ \sigma(\vec{\mathbf{a}})} \langle \mathtt{skip}, S \setminus \{\sigma(\vec{\mathbf{a}})\}\rangle \\[1ex]
\dfrac{\langle e, S\rangle \Downarrow v}{\langle x[\vec{a}] := e, S\rangle \xrightarrow{x[\vec{\mathbf{a}}](v)} \langle \mathtt{skip}, S[x[\vec{\mathbf{a}}] \mapsto v]\rangle}
\end{array}\right\} \vec{\mathbf{a}} = \mathbf{Act}(S)(\vec{a})$$

$$\frac{\langle e, S\rangle \Downarrow v \qquad v \in \{\mathtt{true}, \mathtt{false}\}}{\langle \mathtt{if}\ e\ \mathtt{then}\ c_{\mathtt{true}}\ \mathtt{else}\ c_{\mathtt{false}}, S\rangle \xrightarrow{\tau} \langle c_v, S\rangle}$$

$$\langle \mathtt{while}\ (e)\ c, S\rangle \xrightarrow{\tau} \langle \mathtt{if}\ e\ \mathtt{then}\ (c; \mathtt{while}\ (e)\ c)\ \mathtt{else}\ \mathtt{skip}, S\rangle$$

$$\frac{\langle c_1, S\rangle \xrightarrow{\ell} \langle c_1', S'\rangle}{\langle c_1; c_2, S\rangle \xrightarrow{\ell} \langle c_1'; c_2, S'\rangle} \qquad \langle \mathtt{skip}; c_2, S\rangle \xrightarrow{\tau} \langle c_2, S\rangle$$

$$\langle \mathtt{newactor}\ a\ \mathtt{in}\ c, S\rangle \xrightarrow{a(\mathbf{a})} \langle c, S[a \mapsto \mathbf{a}]\rangle \qquad (\mathbf{a}\ \textbf{fresh})$$

$$\langle \mathtt{when}\ \sigma(\vec{a})\ \mathtt{do}\ c_1\ \mathtt{else}\ c_2, S\rangle \xrightarrow{\tau} \begin{cases} \langle c_1, S\rangle & \sigma(\mathbf{Act}(\vec{a})) \in \mathbf{LS}(S) \\ \langle c_2, S\rangle & \text{otherwise} \end{cases}$$

$$\frac{\mathbf{\Sigma} = \{\vec{\mathbf{a}} \mid \sigma(\vec{\mathbf{a}}) \in \mathbf{LS}(S)\}}{\langle \mathtt{forall}\ \sigma(\vec{a})\ \mathtt{do}\ c, S\rangle \xrightarrow{\tau} \langle \mathtt{for}\ \sigma(\vec{a})\ \mathtt{in}\ \mathbf{\Sigma}\ \mathtt{do}\ c, S\rangle}$$

$$\begin{array}{c}\langle \mathtt{for}\ \sigma(\vec{a})\ \mathtt{in}\ \{\vec{\mathbf{a}}\} \cup \mathbf{\Sigma}\ \mathtt{do}\ c, S\rangle \xrightarrow{\vec{a}(\vec{\mathbf{a}})} \langle c; \mathtt{for}\ \sigma(\vec{a})\ \mathtt{in}\ \mathbf{\Sigma}\ \mathtt{do}\ c, S'\rangle \\ \vec{a} = a_1, \ldots, a_n \qquad S' = S[a_1 \mapsto \mathbf{a}_1, \ldots, a_n \mapsto \mathbf{a}_n]\end{array}$$

$$\langle \mathtt{for}\ \sigma(\vec{a})\ \mathtt{in}\ \varnothing\ \mathtt{do}\ c, S\rangle \xrightarrow{\tau} \langle \mathtt{skip}, S\rangle$$

Figure 3. Operational Semantics

Finally, the most complex command semantically is the `forall`, which loops over all locks in some particular family. Its execution is done in two steps. First, the set of locks in the family that are open is (deterministically) calculated, and second that set is looped over, one lock at a time. For this we need to extend the language with an internal command `for` $\sigma(a_1, \ldots, a_n)$ `in` $\mathbf{\Sigma}$ `do` c, to handle the actual looping. The transition rule for `forall` is then simple: gather all open locks in the relevant lock family and go to the next step, the `for`.

In the `for` we bind the relevant actors to the provided variables and then proceed to execute the body. Just like with the `newactor` rule, we don't care where the scope of the variables ends syntactically, relying on the type system to handle the scoping details.

The transition arrows are labeled with outputs that signal all direct information flows that take place during execution, which in this simple language means all changes to the program state. These are purely for the sake of reasoning about security and otherwise have no effect on the computation. The commands that have an effect on the state are assignments for the memory and `open` and `close` for the lock state. For the actor mapping, the `newactor` command can introduce a single new actor in scope, while the `forall` loop, via the auxiliary internal `for` construct, can bind a number of names in one transition step. All other base rules have no effect on the state, and thus yield the silent output τ.

6.2 Type System

To enforce security we use the type system in Figure 4. Since we only have integers as the base type for values, we don't need to track base types at all, so our type system only handles the security component.

For expressions, the typing judgement is simply $\Lambda \vdash e : r$, where r is a paralock policy which we call the *read effect* of the expression, as it intuitively specifies who may read data from references with this policy. In effect it will be the least upper bound of the policies on references used to compute the expression e. There should be no surprises in how this read effect is computed, though note that the rule for references handles both parametrised and unparametrised references, as we allow the vector of actors to be of length 0.

The typing judgement for commands is a bit more involved, but the various components should come as no surprise. The judgement is

$$\Lambda; \Sigma \vdash c \rightsquigarrow w, \Sigma'$$

where c is the command to type and w is a policy we call the *write effect* of the command. Intuitively this policy specifies who would be able to notice that the command was executed, by observing its effects on the state. It is thus the greatest lower bound of all policies on references, locks and actor variables whose values are affected by the command. The purpose of this policy is to track indirect flows, similar to the use of a "program counter" in many other systems. This can be seen in the rules for the commands that affect control flow, namely `if`, `while`, `when` and `forall`. All these rules compare the policy of the branching expression or lock with the write effect of the body of the command.

The write effect is straightforward to compute for most rules. For assignments, `open` and `close` it is simply the policy of the affected location or lock. The `newactor` command introduces actor variables with the policy $\bot$, which is thus its write effect, as $\bot$ is clearly at least as liberal as any write effect of the body. The most interesting rule in this regard is that of the `forall` command. We cannot in general know exactly which actors will be referenced in the loop iterations, so we assume it may be any of them, meaning that actors introduced by the `forall` that appear in the write effect of the body must be universally quantified. However, since the `forall` also binds actors to the relevant variables, and these variables inherit the policy of the lock, the write effect of the whole command will be exactly the policy of the lock, since we require that to be more liberal than any write effect of the body.

Λ is the set of actors in scope, for both commands and expressions, and the `newactor` and `forall` commands introduce new actors into this scope as expected. We use it only to ensure that any mention of actor variables as arguments to references and locks are done in a correct way, and that no variable names clash.

Σ is the set of locks assumed to be open when the command starts executing, and Σ' is a lower bound on the locks that will be open afterwards. The one place where Σ is actually used is in the assignment rule. In this rule we must determine whether the policy of the expression is compatible with the policy of the variable *relative to the current lock state*. The idea is the same as with flow locks – but the details are more complex. For example, suppose we have a policy $p = \{a, \forall x.\, \mathit{Actsfor}(a, x) \Rightarrow x\}$. Intuitively this says that a may always read the data, and that for any actor x, if the lock $\mathit{Actsfor}(a, x)$ ("flow from a to x is permitted") then x may also read. If we specialise this policy to a lock state $\Sigma = \{\mathit{Actsfor}(a, b)\}$ then the policy in force at that state $p(\Sigma)$ is $\{a, \forall x.\, \mathit{Actsfor}(a, x) \Rightarrow x, b\}$. I.e. in that state, b is also unconditionally permitted to see the data. Specialisation is most easily understood in logical terms: $p(\Sigma)$ is just the most liberal policy which is entailed by the conjunction of p and Σ.

In the definitions that follow we distinguish $\forall$-bound actor variables syntactically, using metavariable x.

DEFINITION 6.1 (Matching). *Let θ be a substitution from bound actor variables to free actor varibles. We say that Σ* matches *Σ'* with *θ if and only if the set of bound actors in Σ is equal to the domain of θ, and $\Sigma\theta \equiv \Sigma'$.*

For example, $\{\mathit{Actsfor}(a, x)\}$ matches $\{\mathit{Actsfor}(a, b)\}$ with $x ::= b$.

DEFINITION 6.2 (Specialisation). *For a policy p and a lock state Σ, we define the normalisation of p at Σ, written $p(\Sigma)$, as*

$$p(\Sigma) = \bigcup_{c \in p} \{c \cdot \Sigma\}, \qquad \textit{where}$$

$$(\forall \vec{x}.\, \Delta \Rightarrow b) \cdot \Sigma \overset{\text{def}}{=} \{\forall \vec{x}.\, \Delta_2\theta \Rightarrow b\theta \mid \Delta \equiv \Delta_1 \cup \Delta_2;\ \Sigma_1 \subseteq \Sigma;\ \Delta_1 \textit{ matches } \Sigma_1 \textit{ with } \theta \}$$

Note that $p(\Sigma)$ always contains Σ (to see this take Δ_1 and Σ_1 to be the empty set in the auxilliary definition above) – i.e. $p(\Sigma) \sqsubseteq p$ – normalising a policy always yields a more liberal policy.

Computing the outgoing lock state is straightforward in most cases, but a few rules are slightly complex. Actors introduced by `newactor` and `forall` are scoped, and when their respective scopes end we need to forget about any locks mentioning those actors, to avoid name clashes with potential future scopes reusing the same actor variable.

Most interesting perhaps is the rule for `close`, which has to account for potential aliasing issues between actor variables. Hence it is maximally pessimistic, and assumes that not only the lock that is explicitly mentioned will be closed, but also any other lock in the same family where the actor arguments may point to the same concrete actors at runtime. Two variables introduced by `newactor` commands can never be aliases of each other as they must represent fresh concrete actors. A variable introduced by a `forall` could alias any other variable though. We assume an implicit predicate alias where $\mathit{alias}(a) = \texttt{true}$ if a in the current scope is introduced by a `forall` construct, otherwise `false`. The result clearly depends on the context in which the function is called. We then define a simple may-alias relation as

$$a \simeq b \overset{\text{def}}{=} \mathit{alias}(a) \vee \mathit{alias}(b) \vee a = b.$$

We extend this relation to equal-length vectors of actors in a pointwise manner. Using this may-alias relation, the rule for `close` is suitably pessimistic about what abstract locks may actually be closed at runtime.

6.3 Security

We show that well typed programs are paralock secure. The proof can be found in the accompanying online appendix. Here we just note the main technical stepping stones – the first of which is the standard property that reduction preserves typability:

LEMMA 6.1 (Preservation). *Let us say that state S is* compatible with *Σ if $\mathbf{LS}(S) \supseteq \mathbf{Act}(S)(\Sigma)$. Similarly we say that state S is compatible with an actor set Λ if $\mathit{dom}(\mathbf{Act}(S)) \supseteq \Lambda$.*

Now suppose that $\Lambda; \Sigma \vdash c \rightsquigarrow w, \Delta$ and $\langle c, S\rangle \xrightarrow{\ell} \langle c', S'\rangle$. Then if Λ and Σ are compatible with S then $\Lambda'; \Sigma' \vdash c' \rightsquigarrow w', \Delta'$ for some Λ' and Σ' compatible with S', $w \sqsubseteq w'$ and $\Delta \subseteq \Delta'$.

$$\frac{}{\Lambda \vdash n : \bot} \qquad \frac{\vec{a} \subseteq \Lambda \quad \forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(x[\vec{a}])}{\Lambda \vdash x[\vec{a}] : pol(x[\vec{a}])} \qquad \frac{\Lambda \vdash e_1 : r_1 \quad \Lambda \vdash e_2 : r_2}{\Lambda \vdash e_1 \oplus e_2 : r_1 \sqcup r_2}$$

$$\frac{\vec{a} \subseteq \Lambda}{\Lambda; \Sigma \vdash \texttt{open}\ \sigma(\vec{a}) \rightsquigarrow pol(\sigma), \Sigma \cup \{\sigma(\vec{a})\}} \qquad \frac{\vec{a} \subseteq \Lambda}{\Lambda; \Sigma \vdash \texttt{close}\ \sigma(\vec{a}) \rightsquigarrow pol(\sigma), \Sigma \setminus \{\sigma(\vec{b}) \mid \vec{a} \simeq \vec{b}\}}$$

$$\frac{}{\Lambda; \Sigma \vdash \texttt{skip} \rightsquigarrow \top, \Sigma} \qquad \frac{\Lambda \vdash e : r \quad r(\Sigma) \sqsubseteq pol(x[\vec{a}]) \quad \vec{a} \subseteq \Lambda}{\Lambda; \Sigma \vdash x[\vec{a}] := e \rightsquigarrow pol(x[\vec{a}]), \Sigma}$$

$$\frac{\Lambda \vdash e : r \quad \Lambda; \Sigma \vdash c_i \rightsquigarrow w_i, \Sigma_i \quad r \sqsubseteq w_1 \sqcap w_2}{\Lambda; \Sigma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2} \qquad \frac{\Lambda \vdash e : r \quad \Lambda; \Sigma \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad r \sqsubseteq w}{\Lambda; \Sigma \vdash \texttt{while}\ (e)\ c \rightsquigarrow w, \Sigma' \cap \Sigma}$$

$$\frac{\Lambda; \Sigma \vdash c_1 \rightsquigarrow w_1, \Sigma_1 \quad \Lambda; \Sigma_1 \vdash c_2 \rightsquigarrow w_2, \Sigma_2}{\Lambda; \Sigma \vdash c_1; c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_2}$$

$$\frac{\Lambda; \Sigma \cup \{\sigma(\vec{a})\} \vdash c_1 \rightsquigarrow w_1, \Sigma_1 \quad \Lambda; \Sigma \vdash c_2 \rightsquigarrow w_2, \Sigma_2 \quad pol(\sigma) \sqsubseteq w_1 \sqcap w_2 \quad \forall a \in \vec{a}.\, pol(a) \sqsubseteq pol(\sigma)}{\Lambda; \Sigma \vdash \texttt{when}\ \sigma(\vec{a})\ \texttt{do}\ c_1\ \texttt{else}\ c_2 \rightsquigarrow w_1 \sqcap w_2, \Sigma_1 \cap \Sigma_2}$$

$$\frac{\Lambda \cup \vec{a}; \Sigma \cap \Sigma' \vdash c \rightsquigarrow w, \Sigma' \quad pol(\sigma) \sqsubseteq \forall \vec{a}.\, w \quad \vec{a} \cap \Lambda = \varnothing}{\Lambda; \Sigma \vdash \texttt{forall}\ \sigma(\vec{a})\ \texttt{do}\ c \rightsquigarrow pol(\sigma), \Sigma' \cap \Sigma \setminus \{\sigma(\vec{b}) \mid \vec{a} \cap \vec{b} \neq \varnothing\}}$$

$$\frac{\Lambda \cup \{a\}; \Sigma \vdash c \rightsquigarrow w, \Sigma'}{\Lambda; \Sigma \vdash \texttt{newactor}\ a\ \texttt{in}\ c \rightsquigarrow \bot, \Sigma' \setminus \{\sigma(\vec{b}) \mid a \in \vec{b}\}} \qquad \frac{\Lambda; \Sigma \vdash c \rightsquigarrow w, \Sigma'}{\Lambda; \Sigma \vdash c} \quad \text{(Top level judgement)}$$

Figure 4. Flow Lock Type System

The second basic property is the global ("big step") property of the effect components of the typing derivation. Stated informally (to convey the intuition without all the technicalities), they say that whenever $\Lambda; \Sigma \vdash c \rightsquigarrow w, \Delta$ then

- If data labelled w is not visible to attacker A then any computation of c in any start state compatible with Σ will not produce any A-visible output (and hence will not modify the parts of the state with policy w or stronger).
- A terminating computation of c in a state with at least locks Σ open will result in at least locks Δ being open.

Finally we have proven the main theorem of this section, namely that a well-typed program is guaranteed to be secure by our semantics for Paralocks. Since the type system as stated is termination insensitive, for instance it allows "high loops" to precede "low writes", we formally have that well-typed programs are termination-insensitive paralock secure:

THEOREM 6.1. *If $\varnothing; \varnothing \vdash c$ then c is termination-insensitive paralock secure ($\mathbf{PLS_{TI}}(c)$).*

This in turn is a corollary of a theorem involving a generalisation of the $\mathbf{PLS_{TI}}$ property. Again, the details are available in the appendix.

7. Recursive Paralocks

In Section 4 we presented an encoding of the DLM. One aspect of a DLM policy was the treatment of the $ActsFor$ relation; implicitly we required that whenever we open a lock $ActsFor(a, b)$ then we must also open all transitive consequences. It is intended that this invariant is implemented explicitly by encapsulating the open operation appropriately within a program which uses a DLM policy.

In this section we explore an extension to the policy language which allows us to specify such properties explicitly, avoiding the need to encode them explicitly in the program. The extension is a natural logical one: allow relations between locks and flows to be specified recursively as part of a global policy component.

In this section we briefly explore the implications of this extension to the questions of *policy* (c.f. §3.2), *expressiveness* (c.f. §4), *semantics* (c.f. §5), and *enforcement* (c.f. §6).

7.1 Policy

Policies will now consist of two parts. Firstly we have policies on memory objects just as before: collections of clauses which have an actor variable (bound of free) as their head. For the purposes of this section it will be useful to write a clause $\forall a_1, \ldots, a_n.\, \Sigma \Rightarrow a$ as $\forall a_1, \ldots, a_n.\, \Sigma \Rightarrow Flow(a)$, thus making the "may flow to"-predicate explicit. The extension we make is to add a *global policy* G which is also a set of clauses. These clauses differ in that their heads may be locks – and thus they may be recursive. For example, in a DLM encoding we would include the following two clauses in the global policy:

$$\forall y.\, ActsFor(y, y);$$
$$\forall x, y, z.\, ActsFor(x, y), ActsFor(y, z) \Rightarrow ActsFor(x, z)$$

This style of policy specification is already familiar in a security context: it amounts to the use of DATALOG programs as policy specifications, and has been used in numerous logics for access control policies – e.g., [Jim01, DeT02, LMW02]. We permit one further useful feature: global policies, in addition to using locks, may also use the distinguished $Flow$ predicate in their specification (see Section 7.2 for examples).

Policy comparison To compare policies p and q we must now take into account the global policy. We write $p \sqsubseteq_G q$ to mean that policy q is more restrictive than policy p in the context of global policy G. We can define this relation by giving a straightforward interpretation in first-order logic. As before we can interpret each clause in p, q and G as first-order Horn clauses, and sets of clauses are interpreted as logical conjunction. Then we define

$$p \sqsubseteq_G q \overset{\text{def}}{=} G \wedge p \models q$$

To see that this does "the right thing", consider some lock state Σ. Suppose that $\Sigma \wedge G \wedge q \models Flow(a)$ – i.e. that in some concrete lock state the policy q permits information to flow to a. Then it can be readily seen that $p \sqsubseteq_G q$ ensures that $\Sigma \wedge G \wedge p \models \mathit{Flow}(a)$ – i.e., p allows any flow that q does.

7.2 Expressiveness

Here we consider a couple of simple examples using recursive paralocks.

Denning Lattices, Reloaded The flow locks encoding of standard Denning-style information flow lattices involves identifying security levels with actors, and representing a security level j by the (lock-free) policy $\{\mathit{Flow}(k) \mid j \leq k\}$. Recursive Paralocks provide several alternative ways to specify this. One example is to represent the policy for data of level k as just $\{\mathit{Flow}(k)\}$. The global policy then must define the covering relation of the lattice (represented as a binary lock $\prec$), together with the rule

$$\forall x, y.\, \mathit{Flow}(x), x \prec y \Rightarrow \mathit{Flow}(y).$$

So, for example, the three point lattice $L \leq M \leq H$ would be represented by the global policy

$$\{L \prec M; M \prec H; \forall x, y.\, \mathit{Flow}(x), x \prec y \Rightarrow \mathit{Flow}(y)\}\,.$$

The "Complete" DLM In the case of the DLM encoding we already mentioned the ability to express reflexivity and transitivity for the $\mathit{ActsFor}$ hierarchy. Similarly the invariant on the $\mathit{RunsFor}$ lock can then be specified as

$$\forall x, y.\, \mathit{ActsFor}(x, y), \mathit{RunsFor}(x) \Rightarrow \mathit{RunsFor}(y)$$

We can also move part of each policy into the global part by applying the $\mathit{ActsFor}$ hierarchy to Flow predicates:

$$\forall x, y.\, \mathit{ActsFor}(x, y), \mathit{Flow}(x) \Rightarrow \mathit{Flow}(y)$$

which says that whenever flow to x is permitted then flow to y is also permitted providing y acts for x. With this rule we no longer need to be explicit about the $\mathit{ActsFor}$ relationship in the data policy itself, so for example we can encode a label $\{o_1 : r_1; o_2 : r_1, r_2\}$ more succinctly as

$$\{\forall x.\, \mathit{RunsFor}(o_1), \mathit{RunsFor}(o_2) \Rightarrow x; r_1; \mathit{RunsFor}(o_1) \Rightarrow r_2\}$$

As an interesting side note, the original version of the DLM ([ML97]) was considered incomplete; a follow-up paper ([ML98]) identified a "complete" policy ordering. Specifically the new formulation made the policy ordering more liberal by weakening it to allow two new $\sqsubseteq$-monotone operations on policies:

- An owner o_1 may to be replaced in a label with an owner o_2 that acts for o_1,
- If a reader r_1 is listed by some owner, and r_2 acts for r_1, then we can also add r_2 as a reader for that owner.

Our original encoding is faithful to the original DLM, and cannot not handle these components, specifically because the policy ordering was ignorant of the transitive nature of the $\mathit{ActsFor}$ relationship and its relation to the $\mathit{RunsFor}$ property. With the extension presented here, policy ordering becomes "complete" and thus corresponds to the revised version of the DLM [ML98].

7.3 Semantics

The definition of security needs only minor modifications to handle recursive paralocks. There are just two places where the definition needs to be modified:

- Generalise $\sqsubseteq$ to $\sqsubseteq_G$ in the definition of the parts of a state that the attacker can see.
- In the definition of security, generalise the comparison between lock state $\mathbf{\Delta}$ and attacker capability $\mathbf{Cap}(A)$ to take into account the global policy G by replacing the condition

$$\mathbf{\Delta} \subseteq \mathbf{Cap}(A) \qquad \text{(logically: } \mathbf{Cap}(A) \models \mathbf{\Delta}\text{)}$$

by $G \wedge \mathbf{Cap}(A) \models \mathbf{\Delta}$.

7.4 Enforcement

Here we consider the impact that recursive paralocks have on the integration with the language and type system of Section 6.

The global policy is essentially DATALOG[3]. DATALOG and modest extensions thereof, has proved to be a popular basis for e.g. access control logics because, among other things, a query (the access control mechanism itself) can be answered in polynomial time. This is also useful in the present context. At runtime we need to enumerate all actors in a given role (the *forall*-construct in our example language), and check whether a particular lock is open (the *when* construct). It is necessary that these can be answered precisely and efficiently, and this is possible because they are datalog queries.

However, type checking is another matter. We do not need to answer datalog queries within the type system, we need to implement policy comparison ($\sqsubseteq_G$). In all other regards we conjecture that the type system given in Section 6 is sound for recursive paralocks – providing we generalise the policy ordering and least-upper-bound operation accordingly.

In the case of assignment $x := e$, for example, where x has policy q, and e has policy r, and we know that at least locks Σ are open, then we need to determine whether $\Sigma \wedge G \wedge r \models q$. This problem (and the similar problem of determining whether $r \sqsubseteq_G q$) is the problem of *containment* of a non-recursive datalog program q in a recursive one $\Sigma \cup G \cup r$. This containment problem is known to be decidable, although EXPTIME-complete (see e.g. [CV97]). Whether this complexity is a problem in practice remains to be seen.

Similarly we need a generalised form of least upper bound operation where $p \sqcup_G q$ denotes the least policy r such that $p \wedge G \models r$ and $q \wedge G \models r$. This kind of operation – i.e., finding a DATALOG program which gives a best approximation to the disjunction of two DATALOG programs – does not to our knowledge seem well studied in the DATALOG literature. However, we note that using $p \sqcup q$ just as before (thus ignoring G) would provide a safe upper bound operation which would be adequate for use in the type system.

8. Related Work

Considering related work, there are two main dimensions along which our work can be compared: policy and semantics.

Policy As mentioned, the Paralocks policy language is built on our earlier work on Flow Locks, and flow locks themselves are a special case of paralocks. As such paralocks can also encode integrity policies and a variety of declassification policies. We refer the reader to [BS06] for more examples.

The Paralocks model is policy neutral – it provides the means to construct information flow policies but does not prescribe any particular kind of policy. In this regard our goals are related to the recent security policy programming language Fable [SCH08]. Fable allows programmers to specify a custom label language for data, and constraints (via dependent types) on how programming operations may use labelled data. Static typing ensures that there is no way to "bypass" the policy. The Fable approach is very flexible and general, but as a result it provides no general extensional semantic security condition – one must construct these on a case-by-case basis. Paralocks is not intended to be as general purpose as

[3] Certain policies that we have used are not *safe* in the DATALOG sense (see e.g. [CGT89]) For example $\forall x.\, \mathit{Runsfor}(o) \Rightarrow \mathit{Flow}(x)$ is *unsafe* because x does not appear in the body of the clause, and so it generates infinitely many instances. However, at any point during run time, the domains of actors is finite and known. Hence the rule can be thought of as a shorthand for $\forall x.\, \mathit{Actor}(x), \mathit{Runsfor}(o) \Rightarrow \mathit{Flow}(x)$.

Fable (for example it does not aim to be able to encode security automata) but focuses on the higher level concept of information flow. As a result we can give a single information-flow semantics to all Paralock policies. It may be possible, however, to use Fable's fine-grained data manipulation policies as an implementation platform for a Paralocks type system, although the only information flow encoding described so far in Fable [SCH08] is a standard information flow lattice policy.

Paralocks deal with dynamic policies in the sense that permitted information flows change over time. We assume that locks are opened and closed at appropriate points in a given program in order to communicate the intended changes. We could say that the program synchronises with the policy via lock operations. An alternative approach is to assume that the trigger for policy change lies outside the program itself. To some extent this is what is done with the acts-for hierarchy in Jif (DLM) programs – and we can take the same perspective using runtime lock checking instead of opening locks from within the program. This works well if all policy changes make the policy more liberal, but asynchronous changes corresponding to the closing of locks are potentially problematic [HTHZ05]. This is related to the problem tackled in the RX policy language [SHTZ06]. This work (and the more recent refinement [BWW08]) is the only other language-based security work of which we are aware which uses roles in an information-flow setting. The main thrust of their approach is to specify and manage information flows which are caused by policy changes. Role management ideas are used to control policy updates. In common with this approach, our semantics also tracks information flows caused by "role management" – something which was not necessary in the earlier flow locks work due lack of run-time locks. We believe that many features of their meta-policies can be directly encoded using paralocks, but we have yet to investigate such examples.

One important feature of Paralocks that was not supported by flow locks is the notion of a *run-time actor*. In this regard the work of Tse and Zdancewic's extension of the DLM with *run-time principles* [TZ04] is closely related, and our inclusion of a run-time lock test was motivated by that work.

We note some further similarities to work on RBAC models. Paralocks permit finer granularity than standard RBAC, with the use of user-specific policies. This level of control appears similar to that provided by *role templates* [GI97]. Our ability to model accesses which are triggered by arbitrary state conditions (modelled via locks) has similarities to *environment roles* [CLS$^+$01]. Related to this, the role activation rules of the OASIS model have a superficial similarity with paralock policies (see e.g. [BEM03]) although these rules would be more like lock invariant specifications in our model.

The extension to recursive paralocks described in Section 7 brings the work much closer to the logic-based access control work (e.g. [Jim01, DeT02, LMW02]). One line of work by Dougherty et al [DFK06] deals specifically with the issues that arise in situations where changes in the environment entail dynamic changes to access control policy. This is analogous to our problem of reasoning about policies in the presence of a program which has side-effects on the policy.

Semantics Semantic models for complex information flow policies are problematic. In some cases – e.g. in the DLM – there is simply no information flow model. In others (e.g., [TZ04]) the semantic models are simply *noninterference in the absence of policy change*. For semantic models of declassification and other dynamic information flow policies which attempt to do more than this (e.g. the "noninterference between policy updates" approach in [SHTZ06, BWW08]), we have argued [BS09] that many semantic models suffer from *flow insensitivity*. Flow insensitivity here means that the semantic conditions are not really fully semantic, since they flag insecurity simply because they do not have an sufficiently accurate model of the context of a given "insecure looking" subcomputation. In contrast, the semantic model here is based on a revised and much improved flow locks semantics developed in [BS09] which in turn builds on the knowledge-style semantics from gradual release [AS07] which avoids this pitfall.

Paralocks do not provide direct control of one of the "dimensions of declassification" [SS05] – namely the specification of *what* information flows from a given data source. For example we might specify that A can read only some part of data (e.g. a checksum) rather than all of it. Although certain simple "what" policies are easily encoded in the Paralocks model (e.g. in the checksum example by ensuring that the checksum is declassified to variable readable by A), this is clearly not the main focus of the Paralocks approach. Semantically this kind of policy falls within the PER model [SS01] and a more constructive perspective on this kind of model is provided by the *abstract noninterference* framework [GM04]. Recent work by Banerjee et al. [BNR08] present a language for expressive declassification policies which broadly lie in this class. Policies are intended to be verified by a mixture of static typing and static verification. In common with our work, Banerjee *et al* also adopt the gradual release style of information flow semantics.

9. Conclusions and Future Work

As we noted initially, our aim with this work is really two-fold. On the one hand we present the policy specification language Paralocks, and show its usefulness for handling a variety of information flow challenges. On the other hand, Paralocks is also a very general framework that is capable of expressing and encoding a wide variety of information flow policy mechanisms, and importantly give such mechanisms a concrete information flow semantics. It is our hope and belief that Paralocks can thus serve as a platform that can simplify further research into policy mechanisms, both future and present, and help give a better understanding of the relationship between various mechanisms.

A different area of future research is to look closer into the interplay between Paralocks and concrete programming language constructs, with a particular focus on type systems guaranteeing Paralock security. The example language we gave in section 6 is very simplistic in many ways, and there are many interesting challenges in adding features like first-class actors (which would be needed, among other things, to avoid our ad-hoc introduction of actor-indexed data) or policies not known until runtime (c.f.runtime labels [ZM07]).

A third potential direction for future research is to fully exploit the connection to logic-based access control languages. Here we can benefit from various well-behaved extensions to DATALOG such as the addition of stratified negation and constraints on data; see e.g., [BFG07] for an elegant authorization language combining such extensions. But the potential here is not just the transferral of technical results. The connection offers new opportunities to transfer *policy concepts* from access control to an information flow context.

Appendix An extended version of this article (available from the authors) contains proofs of the main technical results stated in the paper.

Acknowledgements Thanks to the ProSec group at Chalmers, to Reiner Hähnle and Mike Hicks for useful comments and suggestions. The POPL referees provided outstanding feedback, including independently suggesting the extension outlined in Section 7. This work was partly funded by the Swedish research agencies VR & SSF.

References

[AHSS08] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination insensitive noninterference leaks more than just a bit. In *Proc. European Symp. on Research in Computer Security*, 2008.

[AS07] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 207–221, May 2007.

[BEM03] A. Belokosztolszki, DM Eyers, and K. Moody. Policy contexts: Controlling information flow in parameterised RBAC. In *IEEE 4th International Workshop on Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003*, pages 99–110, 2003.

[BFG07] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. Design and semantics of a decentralized authorization language. In *Proc. IEEE Computer Security Foundations Symposium*, pages 3–15. IEEE Computer Society, 2007.

[BNR08] A. Banerjee, D. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *Proc. IEEE Symp. on Security and Privacy*, pages 339–353. IEEE Computer Society, 2008.

[BS06] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Programming Languages and Systems. 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *LNCS*. Springer Verlag, 2006.

[BS09] Niklas Broberg and David Sands. Flow-sensitive semantics for dynamic information flow policies. In *ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS 2009)*, Dublin, June 15 2009. ACM.

[BWW08] Sruthi Bandhakavi, William Winsborough, and Marianne Winslett. A trust management approach for flexible policy management in security-typed languages. In *Proc. IEEE Computer Security Foundations Symposium*, pages 33–47, 2008.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog(and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[CLS+01] Michael J. Covington, Wende Long, Srividhya Srinivasan, Anind K. Dev, Mustaque Ahamad, and Gregory D. Abowd. Securing context-aware applications using environment roles. In *SACMAT '01: Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 10–20. ACM, 2001.

[CM77] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *STOC*, pages 77–90, 1977.

[CV97] Surajit Chaudhuri and Moshe Y. Vardi. On the equivalence of recursive and nonrecursive datalog programs. *Journal of Computer and System Sciences*, 54(1):61 – 78, 1997.

[Den76] D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.

[DeT02] John DeTreville. Binder, a logic-based security language. In *IEEE Symposium on Security and Privacy*, pages 105–113, 2002.

[DFK06] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *Automated Reasoning, Third International Joint Conference, IJCAR 2006*, volume 4130 of *Lecture Notes in Computer Science*, pages 632–646. Springer, 2006.

[FSG+01] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed nist standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, 2001.

[GI97] Luigi Giuri and Pietro Iglio. Role templates for content-based access control. In *RBAC '97: Proceedings of the second ACM workshop on Role-based access control*, pages 153–159. ACM, 1997.

[GM04] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 186–197, January 2004.

[HTHZ05] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic updating of information-flow policies. In *Workshop on Foundations of Computer Security*, pages 7–18, June 2005.

[Jim01] T. Jim. SD3: A trust management system with certified evaluation. In *IEEE Symposium on Security and Privacy*, pages 106–115, 2001.

[LABW91] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 165–182. ACM, 1991.

[LMW02] N. Li, J.C. Mitchell, and W.H. Winsborough. Design of a role-based trust-management framework. In *IEEE Symposium on Security and Privacy*, pages 114–130, 2002.

[ML97] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. on Operating System Principles*, pages 129–142, October 1997.

[ML98] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symp. on Security and Privacy*, pages 186–197, May 1998.

[Mye99] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, January 1999.

[MZZ+06] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001–2006.

[SCFY96] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *Computer*, 29(2):38–47, Feb 1996.

[SCH08] N. Swamy, B.J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 369–383, 2008.

[SHTZ06] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages. In *Proc. IEEE Computer Security Foundations Workshop*, 2006.

[SS01] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher Order and Symbolic Computation*, 14(1):59–91, March 2001.

[SS05] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 255–269, June 2005.

[SY80] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27, 1980.

[TZ04] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. In *Proc. IEEE Symp. on Security and Privacy*, pages 179–193, 2004.

[Ull90] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., New York, NY, USA, 1990.

[ZM07] L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6, 2007.