A Datalog Semantics for Paralocks

Bart van Delft, Niklas Broberg, and David Sands

Chalmers University of Technology, Sweden

Abstract. Broberg and Sands (POPL'10) introduced a logic-based policy language, *Paralocks*, suitable for static information-flow control in programs. Although Paralocks comes with a precise information-flow semantics for programs, the logic-based semantics of policies, describing how policies are combined and compared, is less well developed. This makes the algorithms for policy comparison and computation ad-hoc, and their security guarantees less intuitive. In this paper we provide a new semantics for Paralocks policies based on Datalog. By doing so we are able to show that the ad-hoc semantics from earlier work coincides with the natural Datalog interpretation. Furthermore we show that by having a Datalog-inspired semantics, we can borrow language extensions and algorithms from Datalog for the benefit of Paralocks. We explore how these extensions and algorithms interact with the design and implementation of *Paragon*, a language combining Paralocks with Java.

1 Introduction

Information flow is at the heart of many security policies. One way to build secure software is to ensure that all information flows conform to an intended policy – for example that trusted data is only influenced by trusted sources, or that confidential data never flows to public channels. Paralocks is a policy language for specifying flexible and dynamic information flow policies [4]. The language allows for the formulation of simple but expressive policies that are dynamic in the sense that they vary over time, but are still statically known at compile time. The *Paragon* [3] programming language is an integration of Paralocks policies with the Java programming language. The *Paragon* compiler's main job is to check that information will only flow according to the specified policies.

It is noted in [3, 4] that certain parts of Paralocks show great similarities with Datalog, a well-known query language for deductive databases. Datalog is a popular language for formalising access control and authorisation policies [10, 11, 15], but as far as we know no other work has used Datalog for information flow policies. A key difference is that we are mainly interested in comparing and combining policies, rather than in simply answering queries. In this work we reify the correspondence with Datalog and replace the original semantics[1] of Paralocks with one that is directly inspired by Datalog. In this paper we show

[1] The term "semantics" can be interpreted to mean two different things: either the dynamic semantics of information flow, or the semantics of operations on policies. What we deal with in this paper is the latter: the semantics of the Paralocks lattice and its operations. While the information-flow dimension of the semantics builds on the policy lattice, it is largely orthogonal to the issues studied in this paper.

1. how this new Datalog semantics for Paralocks can be related to the original semantics from [4],
2. how Paralocks can benefit from its new semantics by incorporating results from the extensive Datalog literature, and
3. what impact extensions imported from Datalog would have in a practical enforcement of Paralocks (i.e. Paragon).

The Paralocks semantics from [4] is mainly of an algorithmic nature, making its security guarantees less intuitive. A relation between Horn clauses and policy evaluation is made, but this relation was rather informal, and led to an unclear and inaccurate connection between the logical connectives (conjunction, disjunction and implication) and the policy operations (meet, join and comparison). We demonstrate that the Datalog semantics provides a more natural specification from which we are able to show the soundness and completeness of the original implementation-oriented definitions of the policy operations (Section 3).

For point 2, we identify two ways in which results from Datalog research can positively be adopted in information flow analysis: *language extensions* (Section 4) and *algorithms* (Section 5).

The Datalog language has been extended in several directions and for some of these extensions it seems natural to include these in a policy specification language as well. In this work we investigate two extensions, known as Datalog with negation [19] (Section 4.1) and Datalog with constraints [14] (Section 4.3), and discuss what effect these extensions have on the Paralocks language.

On a different tack, several algorithms have been proposed in Datalog literature that operate on Datalog programs. One operation that is of particular interest is the containment operation, since it resembles the problem of policy comparison in Paralocks. This operation checks if one policy is less restrictive than another, and is thus at the core of the whole information flow analysis. We identify two sub-classes of containment problems for Datalog programs that relate to Paralocks policies, and show (Section 5) that in general problems of one of these classes, uniform containment, can also be solved by algorithms from the other, conjunctive query containment. We also show how the discussed language extensions require the adoption of other Datalog results in the algorithms for policy comparison.

Finally, with point 3 we consider each of the extensions and algorithms in the practical context of Paragon. Since they affect what expectations are placed on the information flow analysis, not all of them are as easily deployable in practice as they are in theory.

These developments are preceded by a more detailed summary of the Paragon policy language, which follows in the next section.

2 Paralocks

Paralocks is a language for specifying dynamic information flow policies for data in a program. The intended usage is to annotate data sinks and sources in a program with policies that specify how data may flow between them. The flows in the program can then be validated using a static analysis, to ensure that they adhere to the policies as specified. Paralocks itself is inherently agnostic to the underlying programming language in which it is deployed. An integration of Paralocks with Java is currently being developed under the name Paragon, and we will use Java as our example throughout this section. We begin by presenting Paralocks, both intuitively and formally. We then continue by pointing out some of the practical issues that arise when integrating Paralocks with Java.

2.1 Information flow basics

Denning and Denning [9] pioneered a static (compile-time) approach to specifying and verifying the information flows occurring in a program. The core idea is that program variables (inputs, outputs, etc) are each labelled with a policy, where the set of all policies forms a complete lattice. A simple and standard example would be a trivial lattice consisting of two security levels, *Public* and *Secret*, and the policy comparison ordering $\mathit{Public} \sqsubseteq \mathit{Secret}$. Information flows in a program must follow the policy. Consider a simple assignment `x = y`. Here the data contained in variable `y` flows to `x`, a so called *direct flow*. For this assignment to be secure we require the policy on `y` to be *no more restrictive than* the policy on `x`. The intuition is that the data in `y` should not be allowed to flow anywhere via `x` that it would not already be allowed to flow to without going via `x`. To validate this flow, we need to determine that $\mathit{policy}(y) \sqsubseteq \mathit{policy}(x)$.

If data is computed from several different sources, and is then assigned to a variable, e.g. `x = y+z`, then none of the policies on `y` and `z` respectively may be more restrictive than that on `x`. To simplify such comparisons we take the effective policy of an expression to be the *least upper bound* of all the policies on the various sources it is computed from. To validate this flow, we thus need to determine that $\mathit{policy}(y) \sqcup \mathit{policy}(z) \sqsubseteq \mathit{policy}(x)$.

The same operations are needed to track *indirect flows*, such as in the program `if (x) { y=0; }`. Here the value of `y` is indirectly affected by the value of `x`, causing a flow that must be validated by checking that $\mathit{policy}(x) \sqsubseteq \mathit{policy}(y)$.

Finally, if data is written to several different sinks within a conditional branch, e.g. `if (x) { y=0; z=0; }`, then none of the policies on `y` and `z` respectively may be *less* restrictive than the one on `x`. To simplify such comparisons we compute the *greatest lower bound* of all the policies on the various sinks. To validate the flows in the above program, we thus need to determine that $\mathit{policy}(x) \sqsubseteq \mathit{policy}(y) \sqcap \mathit{policy}(z)$.

The use of the Paralocks policy language follows this general approach. What sets Paralocks apart is the specific lattice used – a much richer language for specifying policies than the simple Denning-style lattice – together with a component that allows for a much-needed dynamic interpretation of policies.

2.2 Paralocks

Labelling a data source with a Paralocks policy specifies to whom the data may flow, *and under what conditions.* The two basic building blocks of policies are thus *actors* to whom the data may flow, and *locks*, which represent conditions (typically security-relevant properties of the system). A lock is said to be *open* when the condition it represents is fulfilled, and *closed* when it is not. As an example, the policy $\{A, B \Rightarrow x\}$ specifies that data with this policy may flow to actor x, assuming locks A and B are open. Locks thus provide the possibility for a dependence between the dynamic system state and the policy to be enforced.

At any given point in a program we can conservatively approximate the set of locks that are known to be open (via a static analysis such as a type system). We denote this set the *current lock state*. All policy comparisons are then done in the context of this lock state. Thus, as an example, if the lock A is known to be open, then the policies $\{A \Rightarrow x\}$ and $\{x\}$ (short for $\{\Rightarrow x\}$) are equivalent.

Locks may be parameterised over actors, forming lock *families* (hence the name *Para*(-meterised) *locks*). Clauses in a policy may quantify over actor parameters; for example, the policy $\{\forall x.A(x) \Rightarrow x\}$ specifies that the data may flow to any actor a for whom the corresponding lock $A(a)$ is open.

With these intuitions in hand, we can now go on and define Paralocks policies and operations formally.

Definition 1. *Paralocks policies.*

- *Policies are built from* actor identifiers a, b, *etc. referring to concrete actors, and* parametrised locks*, ranged over by* σ, σ' *etc. Each parameterised lock has a fixed arity,* $arity(\sigma) \geq 0$.
- *A* lock *is a term* $\sigma(a_1, \ldots, a_n)$ *where* $arity(\sigma) = n$. *The symbols* Σ, Σ' *range over sets of locks, and a lock state* LS *is a set of locks without free variables.*
- *A* clause c *is a term of the form* $\forall a_1, \ldots, a_n.\Sigma \Rightarrow a$, *in which* a *and zero or more actor identifiers in* Σ *may have been replaced with the quantified variables* $a_1, \ldots, a_n$. *We call* a *the* head *of the clause, and* Σ *its* body.
- *A* policy *is a set of clauses, written* $\{c_1; \ldots; c_n\}$.

Each Paralocks policy can be read as a conjunction of definite first-order Horn clauses. That is, each policy 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 $\forall a_1, \ldots a_n.(\sigma_1(\vec{b_1}) \wedge \cdots \wedge \sigma_m(\vec{b_m})) \Rightarrow \mathit{Flow}(a)$, where *Flow* is a special reserved lock that does not occur anywhere else[2].

A policy p allows information to flow to actor a in lock state LS if the Horn clause representation implies the validity of $\mathit{Flow}(a)$, denoted $p \cup LS \vDash \mathit{Flow}(a)$. The set of all allowed information flows according to a policy under a given lock state is $p(LS) = \{\mathit{Flow}(a) \mid p \cup LS \vDash \mathit{Flow}(a)\}$. The restrictions on the *Flow* parametrised lock ensure that it cannot be opened directly, and no recursive clauses are possible.

In the original Paralocks work, the meet and join operations were viewed intuitively from a logical perspective as relating to conjunction and disjunction respectively. With that perspective the join operation is viewed as an approximation of logical disjunction of Horn clauses, since (conjunctions of) Horn clauses

[2] $\vec{b_i}$ are vectors of actor identifiers and variables, their lengths depending on the arity of the locks in which they appear.

are not closed under disjunction. This mismatch of semantic domains led to an algorithmic, ad-hoc definition of the join and meet operations. In section 3 we will show, using Datalog to provide the desired semantics for policies, that these definitions of meet and join are in fact the desired ones, not just approximations.

Paralocks can be extended by generalising the lock state to allow recursive rules. A recursive rule is very similar to a policy clause as defined above, but where the head can be an arbitrary lock predicate. This allows for convenient definitions of *properties* of lock families; one useful example is transitivity, e.g. $\{L(a,b), L(b,c) \Rightarrow L(a,c)\}$. An ordinary open lock is then equivalent to a rule with that lock as the head and an empty body. Policies are interpreted as before, only now interpreted in the context of a generalised, possibly recursive lock state. For the rest of this paper we will assume the existence of recursive rules.

Paragon Paragon [3] is an extension of Java that integrates Paralocks policies. In Paragon, fields and variables are annotated with Paralocks policies to specify how they may be used in a program.

Paragon allows recursive rules but with the restriction that a recursive rule may only be introduced when a lock is declared, and then only with that lock as the head of the rule. This allows the use of *global* lock properties, guaranteed to hold throughout the program, but will not allow e.g. transitivity to be turned on and off during program execution.

For convenience we define a specialised notation for policy semantics for this scenario, where the global recursive rules are separated from the lock state: $p(G, LS) = \{\mathit{Flow}(a) \mid p \cup G \cup LS \vDash \mathit{Flow}(a)\}$. In a given program, all policies will be interpreted in the context of the same G.

3 A Datalog Semantics for Paralocks

In this section we introduce Datalog, a well-established deductive database query language, based on logic programming [6, 8, 19]. We demonstrate how its semantics can be used as an alternative semantics for Paralocks.

3.1 Datalog

We start with some definitions and terminology of a basic variant of Datalog found in literature, see e.g. [6, 12, 19]. In this paper $a, b, c, \ldots$ are *constants*, $X, Y, Z, \ldots$ are *variables* and $p, q, r, \ldots$ are *predicates*. Each predicate has a fixed arity ≥ 0. A *term* t is either a constant or a variable and $A = p(t_1, \ldots, t_n)$ is an *atom* provided that p is an n-ary predicate. A ground atom, i.e. one containing no variables, is called a *fact*. A *rule* r is of the form $A_0 :- A_1, \ldots, A_m$ with $m \geq 0$ and each A an atom. A_0 is called the *head* of the rule, $A_1, \ldots, A_m$ the *body* of the rule. A rule is called *safe* if all variables occurring in A_0 occur at least once in the body of that rule. A set of rules is called a *database schema*.

Predicates that appear in the head of a rule are called *intensional* predicates, whereas those appearing only in the body of rules are called *extensional* predicates. A set of ground atoms (facts) on extensional predicates is called an *extensional database* or EDB.

A *query* Q is a set of rules that together define a predicate q, i.e. all rules have the predicate q as their head and q does not occur anywhere else than within Q.

Definition 2 (Answer sets). *The* answer set *to a query Q on an EDB with database schema D, written $Q(D, \mathrm{EDB})$, is the set of all facts on q that can be derived using the deductive rules in both Q and D.*

We omit a formal definition of "the facts that can be derived". The standard operational version of the definition is obtained by iterating an *immediate consequence operator*, starting with the set EDB. The immediate consequence operator, given a set of facts DB, computes DB plus all facts which are an instance of the head of a rule in $Q \cup D$, and for which the corresponding instance of the body is contained in DB. Provided that the rules in Q and D are all safe, a fixed point can be reached in a finite number of iterations of the immediate consequence operator, and this is the set of derivable facts.

Assume two queries Q_1 and Q_2 defining the same predicate q. The query Q_1 is *contained* in Q_2 in the presence of a database schema D if for all EDBs each fact on q that can be derived by Q_1 can also be derived by Q_2. Similar, Q_1 is *uniformly contained* in Q_2 in the presence of a database schema D if for all DB (i.e. facts on both extensional and intensional atoms) each fact on q that can be derived by Q_1 can also be derived by Q_2 [16].

Definition 3 (Datalog Containment).
Regular containment
$Q_1 \preceq Q_2$ *in database schema DS iff* $\forall \mathrm{EDB}.Q_1(DS, \mathrm{EDB}) \subseteq Q_2(DS, \mathrm{EDB})$.
Uniform containment
$Q_1 \preceq^u Q_2$ *in database schema DS iff* $\forall DB.Q_1(DS, DB) \subseteq Q_2(DS, DB)$.

3.2 Paralocks and Datalog

It is fairly straightforward to see how the different elements of Paralocks can be related to Datalog. Lock families correspond to predicates, the lock state to an EDB, clauses are just a different syntax for rules, and the global policy is equivalent to a database schema. The information flow policies themselves correspond to Datalog queries, and therefore the policy evaluation $p(G, LS)$ can be translated into the query evaluation $Q(D, \mathrm{EDB})$ where each policy p can be seen as a query defining the distinguished predicate *Flow*.

The policy ordering is slightly tricky. As is common in information flow lattices, Paralocks defines the top element to be the most restrictive policy. In Datalog ordering, the query with the largest answer set is considered to be the top element. Thus the policy ordering corresponds to the *inverse* of Datalog query containment.

Two complications need to be addressed in this translation. To ensure termination of query evaluation or containment, all rules have to be safe. This does not hold for Paralocks, but by adding the distinguished atom $IsActor(X)$ to the body of a rule if the head is $Flow(X)$ we can make the each clause safe. We then add the guarantee that $IsActor(a)$ is opened at the point that actor a is created.

A second issue arises from the explicit difference in Datalog between extensional and intensional predicates. By direct translation, the Paralocks lock state is a set of both extensional and intensional predicates, whereas the algorithms for query evaluation and containment are defined explicitly only on EDBs.

Somewhat surprisingly, it turns out there is no absolute need for our translation to completely resemble Datalog in this respect. Instead of considering regular containment, we identify our policy ordering as being an instance of uniform containment (Definition 3). In general, uniform containment is considered to be just an approximation to real containment, but for our purpose it closely matches policy ordering. A similar situation is found by Dougherty *et al.* [11] who use Datalog and uniform containment for formalising dynamic access policies.

3.3 A Datalog Semantics

We show how the three policy operations of Paralocks – meet, join and policy ordering – can be semantically defined. The original definitions of meet and join, being of a more algorithmic nature, are shown to be correct computations of these operations. The original ordering check is replaced with an algorithm for uniform containment originating from Datalog, discussed further in Section 5.

Definition 4 (Policy operations). *Suppose that* $\sqcup$ *and* $\sqcap$ *are total binary operations of type* $Policy \times Policy \rightarrow Policy$*. Then we say that* $\sqcap$ *is the* meet *operation if for any policies* p_1, p_2 *and global policy* G *it holds that* $\forall LS.(p_1 \sqcap p_2)(G, LS) = p_1(G, LS) \cup p_2(G, LS)$*. Similarly,* $\sqcup$ *is the* join *operation if* $\forall LS.(p_1 \sqcup p_2)(G, LS) = p_1(G, LS) \cap p_2(G, LS)$*.*

In the context of global policy G*, the binary* ordering *relation* $\sqsubseteq$ *on policies is defined as follows:* p_1 *at most as restrictive as* p_2*, written* $p_1 \sqsubseteq p_2 \iff \forall LS.p_2(G, LS) \subseteq p_1(G, LS)$*.*

Note that the meet and join operations are *specifications* of the desired operations, but *a priori* we do not know whether they exist. There are two things to note in the definition of policy ordering. First, as mentioned in Section 3.2, the direction of the relation is inversed compared to the original Datalog one from Definition 3. Second, it quantifies over *all* possible lock states. However when policies are compared in programs we must be able to take advantage of our knowledge about what locks are known to be open at a given program point, and the standard Datalog ordering is too static for our needs. We therefore adapt this last definition so that it uses the additional knowledge that at least locks L will be open at the point of comparison; $p_1 \sqsubseteq_L p_2$ means that p_1 is no more restrictive than p_2 when at least locks L are open:

Definition 5 (Lock state aware ordering). *In the context of a global policy* G*, and for any lock state* L*, let* $\sqsubseteq_L$ *be the partial ordering on policies defined as:* $p_1 \sqsubseteq_L p_2 \iff \forall LS.L \subseteq LS \Rightarrow p_2(G, LS) \subseteq p_1(G, LS)$*.*

An algorithm for computation of the meet is straightforward since Datalog queries are effectively closed under conjunction.

Theorem 1. *The meet of two policies, $p_1 \sqcap p_2$, can be computed as $p_1 \cup p_2$.*

Proof. Both p_1 and p_2 only define rules on the predicate *Flow*, therefore the immediate consequence operator derives the same set of facts F for all other predicates in both $p_1(G, LS)$ and $p_2(G, LS)$, for any lock state LS. Since *Flow* does not appear in the body of any rule, the union of the facts on *Flow* that p_1 and p_2 derive individually from F, is the same that the policy $p_1 \cup p_2$ would derive on F, hence $p_1(G, LS) \cup p_2(G, LS) = (p_1 \cup p_2)(G, LS)$.

For the crucial join operation it is not as obvious that there exists a computation exactly matching the semantics. In the original work an ad-hoc definition was argued to be sound but not complete [4]. By working with Datalog rather than arbitrary Horn clauses we can now show it to be both sound and complete:

Theorem 2. *The join of two policies, $p_1 \sqcup p_2$, can be computed as*

$$\begin{aligned} p_1 \sqcup p_2 = &\{\Sigma_1 \cup \Sigma_2 \Rightarrow x \mid \Sigma_1 \Rightarrow x \in p_1; \Sigma_2 \Rightarrow x \in p_2\} \\ \cup\ &\{\Sigma_1 \cup \Sigma_2 \Rightarrow a \mid \Sigma_1 \Rightarrow a \in p_1; \Sigma_2 \Rightarrow a \in p_2\} \\ \cup\ &\{\Sigma_1 \cup \Sigma_2[x \mapsto a] \Rightarrow a \mid \Sigma_1 \Rightarrow a \in p_1; \Sigma_2 \Rightarrow x \in p_2\} \\ \cup\ &\{\Sigma_1[x \mapsto a] \cup \Sigma_2 \Rightarrow a \mid \Sigma_1 \Rightarrow x \in p_1; \Sigma_2 \Rightarrow a \in p_2\} \end{aligned}$$

Proof. We have to show that for any lock state LS, actor a:

$$\begin{matrix} \mathit{Flow}(a) \in p_1(G, LS) \\ \mathit{Flow}(a) \in p_2(G, LS) \end{matrix} \iff \mathit{Flow}(a) \in (p_1 \sqcup p_2)(G, LS)$$

$\Longrightarrow$: In order for $\mathit{Flow}(a) \in p_1(G, LS)$ and $\mathit{Flow}(a) \in p_2(G, LS)$ to hold there must be a clause in p_1 and a clause in p_2 that allow for the derivation of $\mathit{Flow}(a)$. It is clear from the definition of $\sqcup$ that for each combination of two clauses in p_1 and p_2 that agree on their heads (after possible substitution), there exists a single clause in $p_1 \sqcup p_2$ that derives the same fact.
$\Longleftarrow$: Similarly, each clause in $p_1 \sqcup p_2$ can be split into two clauses with one in p_1 and one in p_2, possibly with a variable in the head instead of a constant.

That the policy ordering can be given a complete algorithm as well is however not directly obvious. Since recursion is possible in general, Datalog literature tells us that the question of containment is undecidable [17]. The relatively mild restriction that Paralocks places on the query predicate, i.e. *Flow*, as shown in Section 2, to appear only as the head of policy clauses and nowhere else, gives us both a decidable problem and an algorithm to compute the ordering.

Theorem 3. *The ordering check whether a policy p_1 is at most as restrictive as a policy p_2, $p_1 \sqsubseteq p_2$, can be computed using the uniform containment check from Sagiv [16] as $p_2 \preceq^u p_1$.*

Proof. The correctness of the algorithm has been demonstrated by Sagiv [16].

A sound and complete computation for the policy ordering in the presence of a lock state can be obtained via a small modification to Sagiv's algorithm, which we demonstrate in Section 5.2.

This gives us a natural Datalog semantics for Paralocks, and at the same time demonstrates that we can use the same algorithmic implementations for meet and join operations as before. For policy ordering we actually find a better-suited method from Datalog containment which we discuss in detail in Section 5, incorporating the extensions made to the policy language in Section 4.

4 Adopting Datalog Extensions into Paralocks

With a Datalog semantics for Paralocks in place, a whole area of research becomes available to enrich the information flow framework. In this section we consider two popular extensions to regular Datalog: Datalog with negation, and Datalog with constraints. We describe how they could be incorporated into Paralocks, and what consequences they would have on the Paragon compiler.

4.1 Datalog with Negation

One common extension to regular Datalog is the inclusion of negation in the body of rules. For example, consider a Chinese-wall information policy [2] in which consultants may work for company A or company B, but not both at the same time. To model this company A's data would be given the policy $\{\forall x. \mathit{ConsultsForA(x)}, \neg \mathit{ConsultsForB(x)} \Rightarrow x\}$ (symmetrically for company B).

To ensure termination and decidability of Datalog queries it is well-known that negation cannot be added arbitrarily. Rules need to be *safe* with respect to negation, meaning that each variable occurring in a negated atom should appear in at least one positive atom in the same body. In the same way we ensured safety of rules in Section 3.2, we can add a positive $\mathit{IsActor}(X)$ atom for each variable in the body of a rule to make it safe. In addition, the set of rules defining a predicate needs to be stratified. That is, if a recursive path occurs in the definition of a predicate, negation should not be included in that path. For example, consider the database schema (or: global policy):

$$\{ \quad A(X) :- B(X), \neg D(X) \qquad D(X) :- B(X), \neg A(X) \quad \}$$

and an EDB $\{B(a)\}$. Under the query $Q = \{F(X) :- A(X)\}$ there are two possible answer sets depending on whether the first or second rule in the database schema gets evaluated first. Stratification of rules is required to prevent this non-determinism[3]. These concepts could be translated directly into Paralocks. Inclusion of negation is however only straightforward if we ignore the practical issues stemming from the information-flow analysis itself.

[3] Technically, this still does not suffice and some additional restriction on the evaluation algorithm applies [19].

The main complication arises from the particular property we have on policy ordering with respect to the lock state. Before introducing negation, this property was easily formulated: the policy ordering has to hold in each lock state at least containing the current one (Definition 5). In the presence of negation we loose monotonicity; the absence of a lock can now make a policy more liberal as well as more restrictive, making the notion of a 'more permissive lock state' complicated.

4.2 Datalog with Negation and Paragon

The effects of negation on the Paragon compiler would be numerous. To understand why we must understand a bit more about methods-handling in Paragon. Methods are annotated in several ways, including annotations that specify the policies of the method's arguments, returned value and side-effects. Of particular relevance here, is that methods must specify how they interact with the lock state, to allow the analysis to conservatively approximate the current lock state across method calls. Each method should thus detail what locks it *may* close, what locks it *will definitely* open.

To provide enough information for the ordering check, the analysis would need to track both an upper and a lower bound on the set of opened locks. Method signatures then also need to specify which locks are definitely closed and which might be opened. This would result in an additional burden to the programmer and error messages from the compiler would presumably become so complicated that they would only confuse the user. Finally, although we do not have any concrete evidence to support this, the conservative approximation of the lower and upper bound on opened locks is likely to result in the compiler becoming very conservative as well. This means that writing programs involving negation that actually type-check would become a tedious assignment.

These practical considerations make us reject the idea of adopting negation from Datalog directly into Paragon, despite the two languages being so similar. Instead we consider a different tack and attempt to simulate negation with a *dual lock*. During type-checking we replace each occurrence of a negated lock $\neg L$ with a (fresh) lock nL. Any statement that opens or closes L is extended with a statement that performs the inverse operation on nL. Starting the analysis with a lock state in which all nL locks are open, the type system matches the expectations from the programmer.

To give the formal guarantee that both locks will never be open at the same time we can adopt the "Datalog with integrity constraints" extension [19]. This extension allows for the specification of rules without heads; if a database satisfies its body that integrity constraint is violated. For each lock L used in negated form we add the integrity constraint $\leftarrow L, nL$.

What we cannot guarantee is that at least (and in combination with the paragraph above: exactly) one of the two locks is open. If we could, this would imply that we are able to express negation without having the stratification requirement. That is why the analysis starts with the nL locks being open at the start of the analysis. There is clearly a practical issue associated with this. Suppose we have a lock L of arity k, When we create a new actor we must open all of the negative locks that can involve that actor. For nL alone this means that we must open $k(m+1)^{(k-1)}$ locks where m is the number of other actors. In examples of Paralocks policies encountered so far k is at most 2 but this still seems too costly. One reasonable compromise would be to limit negation to unary locks (as used in the Chinese wall policy sketched previously). This would mean that creation of a new actor would entail opening n (negative) locks, where n is the number of unary locks.

In this way we can extend our policy language with some negation possibilities, without obtaining the complications that would arise from adopting the natural negation from Datalog.

4.3 Datalog with Constraints

Another interesting extension to Datalog is called *Datalog with Constraints*, or Datalog^c [14]. This extension allows the bodies of rules to be extended with constraints on the domain elements. For example, for natural numbers one could have the rule $A(X) \text{ :- } B(X), X > 11$. Li and Mitchell [14] present several examples of constraint domains. In our scenario we are considering a constraint domain that is only indirectly included by Li and Mitchell; the only two constraints applicable on actors are that they are either equal or unequal. Having constraints on the inequality of actors (the equality constraint is already implicitly present in standard Datalog) allows for separation-of-interest policies:

$$\{\forall x, y, z. \mathit{WorksFor}(y, x), \mathit{WorksFor}(z, x), y \neq x, z \neq x, y \neq z \Rightarrow \mathit{IsBoss}(x)\}$$

i.e. an actor is a boss if at least two different actors are working for her. This notion of difference is not expressible in the current definition of Paralocks.

Concretely, we can include the notion of a *constraint rule*, which is of the Datalog / Paragon form:

$$A_0 \text{ :- } A_1, \ldots, A_m, \phi \qquad\qquad A_1, \ldots, A_m, \phi \Rightarrow A_0$$

where ϕ is a conjunction of what is called *primitive constraints*, i.e. of the forms $x = y$ and $x \neq y$. A rule is *safe* if all variables occurring in ϕ occur also in $A_0, \ldots, A_m$. This implies that facts are no longer ground atoms, but can contain constrained variables, e.g. $\mathit{IsBoss}(X) \text{ :- } X \neq \mathit{Alice}$. These are called *constraint facts*. The evaluation of a query thus no longer returns an answer set consisting of facts, but one of constraint facts. p_1 is now at most as restrictive as p_2 if any constraint fact derivable by p_2 is logically entailed by at least one fact in p_1. The definition for the ordering check needs to be updated accordingly:

Definition 6 (Lock state aware ordering with constraints). *In the context of global policy G, and for any lock state L, $\sqsubseteq^c_L$ is the* constraints and lock state aware ordering *relation defined by* $p_1 \sqsubseteq^c_L p_2 \iff \forall LS.L \subseteq LS \Rightarrow \forall f_2 \in p_2(G, LS).\exists f_1 \in p_1(G, LS).f_1 \vDash f_2$.

As for Datalog with negation, the area of Datalog with constraints is well studied and can easily be incorporated into Paralocks. The algorithm for checking the policy ordering is easily modified to work with constraints using results from Datalog literature, as we will show in Section 5.

4.4 Datalog with Constraints and Paragon

As with negation the story becomes more involved when considering the practical application of constraints in Paragon. In Paragon, actors can be referred to by variables and may therefore be aliased. For example, whether the opening of lock $L(\mathtt{a})$ followed by the closing of lock $L(\mathtt{b})$ results in lock $L(\mathtt{a})$ being open depends on whether or not the program variables `a` and `b` are aliases for the same concrete actor. In the current implementation without constraints Paragon takes the conservative approach and assumes that a close statement on $L(\mathtt{b})$ closes lock L for all potential aliases of `b`. The current analysis only needs to track whether two program variables may alias or not. When constraints on actors can be added in policies we need to have an aliasing analysis that also tracks whether program variables are must-aliases.

5 Algorithms for Policy Ordering

In the previous section we explored how the Datalog semantics provided us with extension possibilities to the Paralocks policy language. In this section we consider the use of algorithms developed for Datalog. In one direction this can give us policy evaluation algorithms with a lower average complexity than the standard operational semantics (see e.g. [1]). More importantly, though, is that an algorithm for policy ordering can be found in Datalog query containment algorithms.

5.1 Uniform and Conjunctive Query Containment

We consider two Datalog containment relations that correspond to policy ordering: uniform containment and conjunctive query containment. We already mentioned uniform containment as a candidate in Section 3.2. In this section we introduce conjunctive query containment as a second option.

A *conjunctive query* (CQ) is a query that only has extensional predicates in the bodies of its rules. The decision whether a conjunctive query is (regularly, Definition 3) contained in another, possibly non conjunctive query, is known as the CQ containment problem.

Sagiv already noted in his introduction of uniform containment [16] that any CQ containment problem can be solved with a uniform containment algorithm. We show that the opposite direction holds as well. That is, each uniform containment problem, such as the policy ordering for Paralocks, can be addressed with a CQ containment algorithm. Since there is a much larger body of work on algorithms for CQ containment [5, 7, 12] than there is for uniform containment, this increases the collection of containment algorithms for any uniform containment problem. We can transform a uniform containment problem into a CQ containment problem as follows:

Definition 7 (Uniform containment problem transformation).

Suppose that we have a uniform containment problem $Q_1 \preceq^u Q_2$ with database schema DS such that the predicate defined by Q_1 does not occur in the body of any rule. The CQ transformed version $\tau(Q_1, Q_2, DS) = (R_1, R_2, DS', \theta, \theta^{-1})$ where

- *$\vec{p}$ is the set of all intensional predicates in DS,*
- *θ is a substitution mapping each predicate $p \in \vec{p}$ to a fresh predicate p^E,*
- *θ^{-1} is the inverse of θ,*
- *$R_1 = Q_1\theta$,*
- *$R_2 = Q_2$, and*
- *$DS' = DS \cup \{p(\vec{X}) \mathrel{:-} p^E(\vec{X}) \mid p \mapsto p^E \in \theta\}$.*

Note that the resulting R_1 only has atoms on p^E predicates, i.e. extensional atoms in its body. We show that the transformed problem holds in a regular containment check iff the original problem holds in a uniform containment check:

Theorem 4. *Given a transformation $\tau(Q_1, Q_2, DS) = (R_1, R_2, DS', \theta, \theta^{-1})$*

$$Q_1 \preceq^u Q_2 \iff R_1 \preceq R_2$$

Proof. $\Longrightarrow$: We have $\forall DB. Q_1(DS, DB) \subseteq Q_2(DS, DB)$. For any EDB, we need to show that $R_1(DS', \mathrm{EDB}) \subseteq R_2(DS', \mathrm{EDB})$. Let $\delta = \mathrm{EDB} \cup \mathrm{EDB}\theta^{-1}$. Since R_1 is only defined on extensional predicates, all facts are derived directly from the EDB; DS' is not used. $Q_1 = R_1\theta^{-1}$, so $R_1(DS', \mathrm{EDB}) \subseteq Q_1(DS, \delta)$. By assumption, $R_1(DS', \mathrm{EDB}) \subseteq Q_2(DS, \delta)$. And also $R_1(DS', \mathrm{EDB}) \subseteq Q_2(DS', \mathrm{EDB})$ since the $p(\vec{X}) \mathrel{:-} p^E(\vec{X})$ rules in DS' can derive the set $\mathrm{EDB}\theta^{-1}$. Since $Q_2 = R_2$, we obtain $R_1(DS', \mathrm{EDB}) \subseteq R_2(DS', \mathrm{EDB})$.

$\Longleftarrow$: We have $\forall \mathrm{EDB}. R_1(DS', \mathrm{EDB}) \subseteq R_2(DS', \mathrm{EDB})$. Let q be the predicate defined by both Q_1 and Q_2. For any DB, we need to show that for any fact $q(\vec{y})$ such that $q(\vec{y}) \in Q_1(DS, DB)$, it also holds that $q(\vec{y}) \in Q_2(DS, DB)$. $q(\vec{y})$ is derived using a rule $r \in Q_1$. Consider $s =$ the set of (intensional) $p(\vec{x})$ facts used in r to derive $q(\vec{y})$. Safely ignoring the filtering of q predicate on the definition of an answer set, this means that $s \subseteq Q_1(DS, DB)$, and since Q_1 cannot contribute to this also $s \subseteq Q_2(DS, DB)$ (*). Let $\delta = s\theta$, i.e. δ is an EDB. It follows that $q(\vec{y}) \in R_1(\emptyset, \delta)$, since $R_1 = Q_1\theta$; therefore also $q(\vec{y}) \in R_1(DS', \delta)$. By assumption follows $q(\vec{y}) \in R_2(DS', \delta)$ and $R_2 = Q_2$ thus $q(\vec{y}) \in Q_2(DS', \delta)$. Combining this with (*) gives us $q(\vec{y}) \in Q_2(DS, DB)$.

5.2 Checking Policy Ordering

Since uniform containment appears as the most natural Datalog interpretation for policy ordering we adopt the algorithm from Sagiv [16] as our basic implementation. The connection with CQ containment allows us to switch to a different algorithm in the future if that becomes favourable for reasons of flexibility or complexity, and already we incorporate some ideas borrowed from CQ algorithms (Section 5.3). The algorithm from Sagiv works by testing all canonical databases[4] and can be summarised as follows:

To check if $p_1 \preceq^u p_2$ (i.e. $p_2 \sqsubseteq p_1$), consider each rule $r = \mathit{head} :- \mathit{body}$ in p_1 individually. Let θ be a substitution such that all distinct variables in *body* are mapped to distinct fresh constants, i.e. constants not yet present in p_1, p_2 or the global policy G. Uniform containment holds iff for each rule the iterated check $\mathit{head}\theta \in p_2(G, \mathit{body}\theta)$ holds. That this algorithm tests all relevant (canonical) databases and therefore should only succeed if the containment holds over all databases should be quite intuitive. For a proof we refer to [16].

This algorithm checks for ordering as per Definition 4. Converting it into an ordering check as per Definition 5 that takes the current lock state into account requires not that much alteration; given the current lock state L we simply replace the iterated check by $\mathit{head}\theta \in p_2(G, L \cup \mathit{body}\theta)$.

The complexity of the algorithm is directly influenced by the number of rules in p_1 and the complexity of deciding whether $\mathit{head}\theta \in p_2(G, L \cup \mathit{body}\theta)$. That is, the complexity of the ordering algorithm is in the same order of complexity as query evaluation, which has worst case complexity EXPTIME-complete [8]. This complexity is dictated by the (maximum) sum of all arities of the predicates in the bodies of all clauses. In our use of Paralocks to model information flow policies and idioms so far we have not come across any predicates with an arity greater than two or clauses with more than three atoms in the body, which gives the indication that the impact of this complexity is still low. Further practical studies are required to confirm this suspicion.

5.3 Policy Ordering with Extensions

In this section we consider how the ordering operation based on uniform containment as described above needs to be adapted to include the extensions discussed in Section 4.

Negation For the inclusion of negation we do not use the natural Datalog negation but instead simulate negation by giving each lock L a counterpart nL (Section 4.1), therefore no changes to the ordering operation are required.

Constraints Extending the work by Sagiv, Ullman briefly considers constraint domains [18] using results on CQ containment from Klug [13]. Here we instead choose to translate the method deployed by Farré *et al.* [12]. The essence of both methods [12, 13] is however the same. To quantify over all databases the algorithms enumerate, as in Section 5.2, all relevant canonical databases. The Constructive Query Containment (CQC) method introduced by Farré *et al.* [12] incorporates so-called Variable Instantiation Patterns (VIPs) for this purpose[5].

[4] A canonical database is a representative of the family of databases that can be obtained by all possible one-to-one replacements of constants.

[5] The entire CQC method seems a tempting candidate to be used instead of uniform containment, but unfortunately does not perform well in the presence of recursive rules.

Since in our domain actors are only equal or unequal we adopt the *negation VIP* into our ordering check. The resulting algorithm is as follows:

Given K the set of all constant actors appearing in p_1, p_2 and G, for each rule $r = \mathit{head} :- \mathit{body}$ in p_1 let $\vec{x}$ be the set of all variables. Let $\Theta = s(\vec{x}, K)$ be a *set* of substitutions, where:

$$\begin{aligned} s(\{x\} \cup \vec{x}, K) = & \bigcup_{k \in K} \{\{x \mapsto k\} \cup \theta \mid \theta \in s(\vec{x}, K)\} \\ & \cup \quad \{\{x \mapsto k_n\} \cup \theta \mid \theta \in s(\vec{x}, K \cup \{k_n\})\} \quad k_n \notin K \\ s(\emptyset, K) = & \ \emptyset \end{aligned}$$

p_2 is at most as restrictive as p_1 iff for each rule, each $\theta \in \Theta$, the iterative check $\mathit{head}\theta \in p_2(G, L \cup \mathit{body}\theta)$ holds.

Textually, Θ is a list of substitution-combinations in which each variable in *body* is mapped to one of the existing constants or a fresh constant. This ensures that the algorithm tests all canonical databases. The number of iterative checks that needs to be performed increases with a factor V^K where V is the number of variables in a rule, and K the total number of constants. In general we expect this cost increase to be modest, since the (small) cases we encountered so far almost never had more than one concrete actor in a policy or global rule.

6 Conclusions

By changing the semantics for Paralocks policies from its original ad-hoc version to one based on Datalog, we have provided a more natural and intuitive understanding of the language. Still, the original semantics coincides with the Datalog version, and due to its algorithmic nature provides a good guidance for implementation. Another advantage is that we can transfer language extensions and algorithms from Datalog into Paralocks, although practical considerations arising from the use of Paralocks in a type checker must be taken into account.

The algorithms and extensions discussed in this work are to be added to the next generation of Paragon, an information-flow aware compiler for Java. We also aim to improve the feedback from the compiler by incorporating the discussed algorithm for policy ordering, since it allows us not only to say where in the program an illegal information flow occurs, but also to provide more feedback as to why this flow is illegal.

Acknowledgments This work has been partly funded by the Swedish research agencies VR and SSF, and the European Commission EC FP7-ICT-STREP WebSand project. Thanks to the ProSec group at Chalmers, in particular to Wolfgang Ahrendt and Pablo Buiras for discussions and feedback.

References

1. Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract).

In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '86, pages 1–15, New York, NY, USA, 1986. ACM.

2. David F.C. Brewer and Micheal J. Nash. The Chinese Wall Security Policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206–214, 1989.
3. Niklas Broberg. *Practical, Flexible Programming with Information Flow Control*. PhD thesis, Chalmers, Göteborg University, Göteborg, Sweden, 2011.
4. Niklas Broberg and David Sands. Paralocks – Role-Based Information Flow Control and Beyond. In *POPL'10, Proceedings of the 37th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2010.
5. Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. On the decidability of query containment under constraints. In *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '98, pages 149–158, New York, NY, USA, 1998. ACM.
6. Stefano Ceri, Georg Gottlob, and Letizia Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. on Knowl. and Data Eng.*, 1(1):146–166, 1989.
7. Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. *Theoretical Computer Science*, 239(2):211 – 229, 2000.
8. Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33:374–425, September 2001.
9. Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.
10. John DeTreville. Binder, a logic-based security language. In *IEEE Symposium on Security and Privacy*, pages 105–113, 2002.
11. Daniel Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and Reasoning About Dynamic Access-Control Policies. In *Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 632–646. Springer, 2006.
12. Carles Farré, Ernest Teniente, and Toni Urpì. Checking query containment with the CQC method. *Data & Knowledge Engineering*, 53(2):163–223, 2005.
13. Anthony Klug. On Conjunctive Queries Containing Inequalities. *J. ACM*, 35(1):146–160, January 1988.
14. Ninghui Li and John C. Mitchell. Datalog with Constraints: A Foundation for Trust Management Languages. In *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages*, January 2003.
15. Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust-management framework. In *IEEE Symposium on Security and Privacy*, pages 114–130, 2002.
16. Yehoshua Sagiv. Optimizing Datalog Programs. In *Foundations of Deductive Databases and Logic Programming*, pages 659–698. Morgan Kaufmann, 1988.
17. Oded Shmueli. Decidability and expressiveness aspects of logic queries. In *Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '87, pages 237–249, 1987.
18. Jeffrey Ullman. Information integration using logical views. In Foto Afrati and Phokion Kolaitis, editors, *Database Theory ICDT '97*, volume 1186 of *Lecture Notes in Computer Science*, pages 19–40. Springer Berlin / Heidelberg, 1997.
19. Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1988.