Paragon for Practical Programming with Information-Flow Control

Niklas Broberg, Bart van Delft, and David Sands

Chalmers University of Technology, Sweden

Abstract. Conventional security policies for software applications are adequate for managing concerns on the level of access control. But standard abstraction mechanisms of mainstream programming languages are not sufficient to express how information is allowed to flow between resources once access to them has been obtained. In practice we believe that such control - information flow control - is needed to manage the end-to-end security properties of applications.
In this paper we present Paragon, a Java-based language with first-class support for static checking of information flow control policies. Paragon policies are specified in a logic-based policy language. By virtue of their explicitly stateful nature, these policies appear to be more expressive and flexible than those used in previous languages with information-flow support.
Our contribution is to present the design and implementation of Paragon, which smoothly integrates the policy language with Java's object-oriented setting, and reaps the benefits of the marriage with a fully fledged programming language.

Keywords: information flow, static enforcement

1 Introduction

The general goal of this work is to construct innovative design methods for the construction of secure systems that put security requirements at the heart of the construction process, namely *security by design*. To do this we must (i) understand how we can unambiguously formulate the policy aims for secure systems, and (ii) develop technology to integrate these goals into design mechanisms and technologies that enable an efficient construction or verification of systems with respect to those policies.

We address this challenge using a programming language-centric approach, presenting a full-fledged security-typed programming language that allows the programmer to specify how data may be used in the system. These security policies are then enforced by compile-time type checking, thus requiring little run-time overhead. Through this we can guarantee that well-typed programs are secure by construction.

But which security policies might we want for our data, and why do we need special support to express them? Certain security policies, for example access control, are relatively easy to express in many modern programming languages. This is because limiting access to resources is something that good programming language abstraction mechanisms are designed to handle. However, access control mechanisms are often a poor tool to express the intended end-to-end security properties that we wish to impose on our applications.

Consider a travel planner “app” which permits you to plan a bus journey, and even add your planned trip to your calendar[1]. In order to function, the app must have access to the network to fetch the latest bus times, and must have access to your calendar in order to add or remove schedules. But an app with these permissions can, for example, send your whole calendar to anywhere on the net.

What we want is to grant necessary access, but limit the *information flows*. In this case we want to at least limit the information flows from the calendar to the network while retaining the app’s ability to read and write to both.

Research on controlling these information flows has progressed over the last decades. In this paper we identify three generations of security properties and control mechanisms for information-flow:

Information-Flow Control In the 70’s, Denning & Denning pioneered the idea of certifying programs for compliance with information flow policies based on military-style secrecy classifications [7,8]. They used program analysis to validate that information labelled with a given clearance level could never influence output at any levels lower in the hierarchy – so for example a certified program could never leak top-secret information over a channel labelled as public.

The language FlowCAML [24], a variant of ML with information-flow types and type inference, represents the state-of-the-art in support for static Denning-style confidentiality policies.

Beyond Mandatory Information Flow Control Although a rigid, static hierarchy of security levels may be appropriate in a military message-passing scenario, it became quickly apparent that such a strict and static information flow policy is too rigid for modern software requirements. In practice we need a finer-grained and more dynamic view of information flow.

The concept of *declassification* – the act of deliberately releasing (or leaking) sensitive information – is an important example of such a requirement. Without a possibility to leak secrets, some systems would be of no practical use.

For example an information purchase protocol reveals the secret information once a condition (such as “payment transferred”) has been fulfilled. Yet another example is a password checking program that inevitably leaks some information: even when a login attempt fails the attacker learns that the guess is *not* the password.

With this in mind, the *Jif* programming language [17,20] can be seen as the next milestone after the pure Denning-style approach. Jif is a subset of Java extended with information flow labels.

As well as implementing an important distributed view of data ownership, the so-called *Decentralised label model* [18,19], Jif included the possibility of *declassification*, which provides a liberal information flow escape hatch for programs which would otherwise be rejected.

[1] The example is based on a family of actual Android apps (e.g. de.hafas.android.vasttrafik).

Paragon, a Third-Generation IF Language Declassification, in many shapes and forms, has been widely studied in the research community in recent years [23]. The large variety of declassification concepts is testament to the fact that there is simply no single right way to control the flow of information that goes against the grain. Moreover, it is not always natural to view information flow policies as consisting of "good flows plus exceptional cases" at all; in some situations there is no obvious base-line policy, and the flows which are deemed acceptable may depend on the state of the system at any given moment.

In earlier work [5] we introduced a new highly versatile policy language, *Paralocks*, based around the idea of *Flow Locks*. We demonstrated its ability to model a wide variety of policy paradigms, from classical Denning-style policies to Jif's decentralised label model, as well as the capability to model *stateful* information flow policies. But the idea of using Paralocks as types in a statically-checked programming language was only demonstrated for a toy language. The question whether a Flow Locks-based policy language could feasibly scale to inclusion in a full-fledged programming language, to allow practical programming with information flow control, was left open.

The main contribution of this paper is to answer that question with an emphatic yes. We present the new programming language Paragon, which extends the Java programming language with information flow policy specifications based on an object-oriented generalisation of Paralocks. Not only does it turn out to be feasible, but the marriage of our stateful policy mechanism and Java's encapsulation facilities yields a whole that is greater than the sum of its parts: it allows for the creation of complex policy mechanisms as libraries, giving even stronger control over flows and declassification than the policy language alone.

The remainder of the paper is structured as follows. Section 2 presents the language Paragon: its policy language and integration in Java; followed by a simple showcase of Paragon in Section 3. In Section 4 we give an overview of the enforcement mechanism and details of our implementation. Section 5 discusses our experience from two larger case studies. Related work in Section 6 and conclusions in Section 7 round out the paper.

2 The Language Paragon

Paragon is largely an extension to the Java language and type system. Our choice for Java is motivated by its relatively clear semantics and the wide adoption of Java in both commercial and academical settings. In addition, it allows us to reuse existing ideas from, and simultaneously compare Paragon with, Jif [17,20], the only (other) java-based full-fledged security-typed programming language to date. We discuss Paragon's relation to Jif in more detail in § 6. We do not, however, rely on any particular features of Java for the integration of our policy language to work, and posit that it would be equally feasible to do this for other statically typed languages with safe memory management, e.g. ML or Scala.

In this section we give a high-level overview of Paragon and its various components, leaving more technical features, such as extending Java Generics, to the technical report version of this paper [1].

2.1 The Paragon Policy Language

The subjects of Paragon policies are the information-flow relevant entities, which we refer to as *actors*. An actor could be a user, a resource, a system component, an information source or sink, etc.; any entity that has an information-flow concern.

In Paragon, these entities are represented by *object references*. For instance, the code fragment below creates regular instances of the `User` and `File` class, where `alice` and `f1` can play a dual role; both as program variables, and as actors in Paragon policies.

```
User alice = new User();
User bob   = new User();
File f1 = new File();
File f2 = new File();
```

A *paragon policy* is used to label information containers in the program (fields, local variables), and specifies to which actors the information in that container is allowed to flow. A policy consists of a set of *clauses*, each specifying one particular actor, or one group of actors of a particular type.

For example, the policy `p1` states that information may flow to the specific users Alice and Bob, while the policy `p2` states that information may flow to any file:

```
policy p1 = { alice: ; bob: };
policy p2 = { File f: };
```

This makes the policy `{ Object o: }` the most permissive, and the policy with no clauses (denoted `{:}`) the most restrictive paragon policy.

A clause may have a *body* that constrains the states in which the information may flow to the actors specified in the head. These constraints come in the form of *locks*; typed predicates representing the policy-relevant state of the system.

A lock can be opened or closed for given actor arguments. Viewing a lock as a predicate, opening a lock corresponds to assigning it the value true. Below we define two locks, one for modelling the ownership of files, and another for the organisational hierarchy among users.

```
lock Owns(File, User);
lock ActsFor(User, User);
policy p3 = { File f: Owns(f, alice) } ;
policy p4 = { (User u) File f: Owns(f, u), ActsFor(u, alice) };
```

The policy `p3` expresses that information can flow to any file owned by Alice, while the policy `p4` states that `u` ranges over users, and that information having this policy may flow to any file `f` for which `f` is owned by some `u` such that `u` acts for `alice`. Note that variables that are mentioned in the head of a clause are universally quantified, whereas those only appearing in the body are existentially quantified.

A lock can be declared to have *properties*. A property specifies conditions under which some locks are *implicitly* open. For example, we might want to express that the acts-for relation is transitive and reflexive.

This requirement can be stated at the point of declaration by replacing line 2 in the above with:

```
lock ActsFor(User, User)
  { (User x)      ActsFor(x,x):
  ; (User x y z) ActsFor(x,y): ActsFor(x,z), ActsFor(z,y) }
```

Transitivity and reflexivity properties (as well as symmetry) are a common pattern, so Paragon provides syntactic sugar for these:

```
reflexive transitive lock ActsFor(User, User);
```

The Paragon Policy Language is an object-based generalisation of Paralocks [5,29], and is described in more detail in the technical report [1].

2.2 Information-flow policies in Paragon

The various flows of information that need to be controlled in Paragon are essentially the same as the ones occurring in Java. As is common in information-flow analysis we make a distinction between *direct* and *indirect* information flows.

Direct flows The typical direct flow is an assignment, where information flows directly from one location to another. Direct flows also happen at method calls (arguments flow to parameters), returns (the return value flows to the caller) and exception throws (an exception value flows to its enclosing catch clause).

Continuing in the style of the examples from the previous section, let `x` be a variable with the policy `{File f:Owns(f, alice)}` and `y` a variable with the policy `{f1:}` in the assignment `y = x;`. Whether or not the assignment will be flagged as an error by the Paragon compiler depends on the lock state in which the direct flow occurs.

If the `Owns(f1,alice)` lock is statically determined to be closed the compiler raises an error, since the information stored in `x`, according to its policy, should only flow into file `f1` when the file is owned by Alice, whereas the information in `y` can always flow to `f1`. In other words, the assignment has insecure information flow because it moves information to a place where it becomes visible to more actors than its policy declares. If, however, the lock is determined to be open, i.e. declaring that `alice` owns `f1`, the assignment occurs in a state where `f1` can already read the information in `x`, and so the program compiles successfully.

Indirect flows An indirect flow is one where the effect of evaluating one term reveals information about a completely different term that was evaluated previously. The typical indirect flow is a side-effect happening in a branch that reveals which branch was chosen, which in turn reveals the value of the conditional expression that was branched on. Indirect flows also arise from other control flow constructions (including loops and exceptions), and field updates or instance method-calls (possibly revealing the object they belonged to).

Due to the delayed nature of these information flows, the lock state in effect at the time of the indirect flow might be different to that in effect at the point at which it is revealed. Therefore, indirect flows are handled conservatively, by not allowing the lock state to affect which of these flows are considered secure.

2.3 Policy annotations

When integrating the policy language into Java, the two core design issues are (i) how policies are to be associated to data, and (ii) how the lock state is specified, updated, and queried.

Policies as modifiers In Paragon every information container (field, variable, parameter, lock) has a policy detailing how the information contained therein may be used. Every expression has an effective policy which is (an upper bound on) the conjunction of all policies on all containers whose contents affect its resulting value – we refer to this as the expression's *read effect*.

Paragon separates policies from base (Java) types syntactically by having all policy annotations as modifiers. A modifier `?pol` denotes a policy on an information container, and the read effect of accessing that container. When used on a method we refer to it as the *return policy*, as it is the read effect on the value returned by the method. Using modifiers for policies allows for a clean separation of concerns, allowing us to analyse base types and policies separately.

Similarly, every expression (and statement) has a *write effect*, which is (a lower bound on) the disjunction of all policies on all containers whose contents are modified by the expression. Write effects allow us to control implicit information flows, by limiting the contexts in which expressions with side-effects may occur. A modifier `!pol` denotes a write effect, and is used to annotate methods.

Policy modifiers are also placed on exceptions declared to be thrown by a method. A read effect modifier on an exception denotes the read effect of inspecting the thrown and caught exception object. More interesting is the write effect modifier, which serves two purposes in relation to indirect flows. First, it restricts the contexts in which the exception may be thrown within the method. Second, it imposes a restriction of its own on all subsequent side-effects until the point where the exception has been caught and handled. Together, these two restrictions ensure that no information leaks can occur by observing whether or not an exception has been thrown.

All exceptions in Paragon must be *checked*, i.e. declared to be thrown by methods that may terminate with such exceptions. This implies the need for analyses that can rule out the possibility of exceptions, in particular for null pointers, to avoid a massive blow-up in the number of potential exceptions that must be declared. Paragon adds the modifier `notnull` for fields, variables and method parameters that may never be null, to aid the null-pointer analysis.

To reduce the burden on the programmer to put in policy annotations, Paragon attempts either to infer, or to supply clever defaults for, policies on variables, fields and methods. We omit the details of policy defaulting, and discuss the inference mechanism in § 4.

Lock state analysis Manipulation of the lock state is done programmatically through the use of the Paragon-specific statements `open` and `close`. The compiler performs a *lock state analysis* which conservatively approximates the set of locks guaranteed to be open at any given program point.

In cases where we cannot know statically that a lock is open, we allow runtime *lock queries* to guide the analysis: A lock can be used syntactically as an expression of type

boolean, with the value **true** if the lock is currently open. If a lock query appears as the condition to e.g. an **if** statement, the analysis can include the knowledge of the lock's status when checking the respective branches.

To facilitate modularity, Paragon introduces three modifiers, used on methods and constructors, to specify their interaction with the lock state:

- **+**`locks` says that the annotated method *will* open the specified lock(s), for every execution in which the method returns normally. We call this the *opens* modifier.
- **-**`locks`, dubbed the *closes* modifier, says that the method *may* close the specified lock(s), for *some* execution.
- ~`locks`, the *expects* modifier, says that the specified lock(s) must be open whenever the method is called.

The *opens* and *closes* modifiers are also used to annotate each exception type thrown by a method, to signal to the analysis what changes to the lock state can be assumed if the method terminates by throwing an exception of that type.

As a middleground between private and public locks, Paragon introduces the modifier **readonly** for locks, indicating that outside the class the lock can be queried, but not opened or closed.

3 Brief Example

To illustrate the language features of Paragon we present the scenario of a simple social network. In the network, users can befriend each other and share messages in the form of posts that can be read by their friends. The scenario contains two information flow policies that we want Paragon to enforce.

First, posts can only be read by a direct friend of the poster or, if the poster so indicates, by friends of friends of the poster. A user can decide, per post, whether it should be shared with friends-of-friends or not. Paragon should thus enforce that the network properly checks the friendship relations before allowing a user to read a post.

Second, to prevent injection or scripting attacks, a message should be properly *sanitised* before it is stored in the network. That is, we want to enforce the policy that all posted messages first pass through a sanitising function.

The Paragon implementation of this network is shown in Figure 1. Some policy annotations are omitted in the implementation, since Paragon provides default policies in these cases. For example, all fields that do not specify a read effect automatically get the least restrictive policy `{Object x:}`.

To establish the first policy we define the `Friend` lock to model friendships. Similarly we create a lock `FoFriend` to model friend-of-friend relations. Since the `User` class does not explicitly open or close this lock and exports it as **readonly** we know that it models a purely derived property of the `Friend` lock, and thus one that will evolve correctly as the friendship status changes over time.

With the locks in place we can now create the desired policy as `messagePol`, which we use for the read-effect on a post's content. We assume that the correct `Friend` instances are opened elsewhere in the program. Turning sharing with friends-of-friends on per post is handled in the `post` method by opening the `ShareFoF` lock for that post.

```
public class User {
  public reflexive symmetric lock Friend(User, User);
  public readonly lock FoFriend(User, User)
    { (User x y z) FoFriend(x,y) : Friend(x,z), Friend(z,y) };
  public void receive(?{this:} String data) {
    ... // User receives provided data
  }
}


public class Post {
  public lock ShareFoF(Post);
  public final User poster;
  public static final policy messagePol =
    { User x : User.Friend(x, poster)
    ; User x : User.FoFriend(x, poster), ShareFoF(this) };
  public final ?messagePol String message;
  public Post(?{Object x:} User p, ?messagePol String m) {
    this.poster = p;
    this.message = m;
  }
}


public class Sanitiser {
  private lock Sanitised;
  public static final policy unsanitised = {Object x : Sanitised};
  public static ?{Object x:} String sanitise (?unsanitised String s) {
    open Sanitised {
      return /* Sanitised string */ ;
    }
  }
}


public class Network {
  private static Post[] posts = new Post[10]; // Shifting list of posts
  private static int index = 0;               // Where to place the next post

  !{Object x:} static void post( ?{Object x:}           User    user
                               , ?Sanitiser.unsanitised String  message
                               , ?{Object x:}           boolean shareFoF ) {
    String sM = Sanitiser.sanitise(message);
    Post p = new Post(user, sM);
    if (shareFoF)
      open Post.ShareFoF(p);
    posts[index] = p;
    index = (index + 1) % posts.length;  // Next time overwrite oldest post
  }

  static void read(?{Object x:} User user, ?{Object x:} int i) {
     ?{user:} String res = null;
     Post p = posts[i];
     if (p != null) {
       if (User.Friend(user, p.poster))
         res = p.message;
       if (Post.ShareFoF(p))
         if (User.FoFriend(user, p.poster))
           res = p.message;
     }
     user.receive(res);
  }
}
```

Fig. 1. A simple social network application written in Paragon.

As an effect of calling this method the array `posts` is changed (among others). Any observer that may notice this change (i.e. of level `{Object x:}` and above) may thus notice that this method has been called. To prevent this method from being called in a context where these side-effects result in implicit flows, we are required to annotate the method with the corresponding write effect.

The user's `receive` method lets the user read the provided information, therefore arguments to this method should be allowed to flow to that user. All combined, we get Paragon's enforcement ensuring that the policy-relevant state is properly checked before sharing a post with another user.

Leveraging on Java's encapsulation mechanism we are able to provide the ingredients for the sanitisation policy entirely as a separate library. The lock `Sanitised` is private to the class, meaning that no code outside the class is able to open, close or even mention the lock. Therefore, any data labelled with the `unsanitised` policy cannot lose its `Sanitised` constraint, other than by actually sanitising the data by calling the exported `sanitise` method. With this library we can thus easily enforce our second policy by labelling each newly incoming message as `unsanitised`.

The example demonstrates the three different generations of information-flow control policies and how Paragon models them.

As per traditional non-interference, some flows are never allowed in the network. For example, Paragon enforces that a posted message can only flow to users in the network, and not to any other channel. We see an example of the exceptional information declassification pattern in the sanitiser library: the `sanitise` function serves as a declassifier, deliberately allowing the provided argument to flow to more actors. Finally, the locks used to model friendships exemplify third-generation information-flow policies. There is no explicit declassification of information, rather flows are allowed or not depending on the state of the system – in this case the state of the social network.

4 Enforcement of Paragon Policies

Enforcement of information flow policies in Paragon is no small task, and presenting the information flow type system in its entirety is beyond the scope of this paper. Instead, we sketch a high-level overview of the most important analyses involved, presented as a sequence of *phases*, and focus on the last phase in which information flows are tracked.

Phase 1: Type checking The first phase roughly corresponds to ordinary Java type checking, albeit with some additions for Paragon-specific constructs. Particularly, we must assure that arguments to locks are type correct, and that policy expressions used in modifiers are indeed of type **`policy`**. This phase also checks that potential (runtime) exceptions are properly handled.

Phase 2: Policy type evaluation Locks, policies, and object references all play a dual role, both as type-level and value-level entities. In this phase the values of each of these entities are statically approximated. For *locks* we ensure that, whenever a lock is queried, the information in the query is propagated to the respective branch (or loop body).

For fields and variables holding *actors*, i.e. object references, approximating their runtime values means performing an alias analysis. Our present analysis is simple but has performed well enough in practice. However, work is in progress to improve its precision by adapting the work by Whaley and Rinard [30].

Since policies can be used as values at runtime, and dynamically hoisted to the type level, our analysis needs to approximate policies as *singleton types*, similar to the analysis of actors. For each field or variable storing a *policy*, and for each policy expression appearing in a modifier, we thus calculate upper and lower bounds on the policy held by that variable at runtime.

Further, we need ways to relate policies that are not known statically to other (static or dynamic) policies, to improve precision. Similar to runtime lock queries, we thus let our policy analysis be guided by inequality constraints between policies appearing as the condition in **`if`** statements and conditional `?:` expressions. This problem has been studied by Zheng and Myers in the context of Jif [32], and our solution closely follows theirs.

Phase 3: Lock state evaluation The next sub-phase approximates the *lock state*, i.e. it calculates the set of locks which we can statically know to be open, at each program point. This amounts to a dataflow analysis over the control flow graph, to properly capture the influence of method calls and exceptions, and to handle loops. Each program point where a direct flow takes place is annotated with the lock state in effect at that point.

Phase 4: Policy constraint generation The constraint generation phase will result in a set of constraints on the form $p \sqsubseteq_{LS} q$ where p and q are policy expressions and LS is the lock state (calculated in Phase 3) at the program point where the constraint was generated (omitted if empty). As argued in §2.2 the lock state is only taken into account for direct flows. Policy expressions possibly contain meta-variables, for which the constraint solving phase then solves.

Phase 5: Policy constraint solving The last phase solves the generated constraints, on a per-method basis. A solution to a set of constraints is an assignment of policies to constraint variables that satisfies all the policy comparison constraints. The algorithm needs only determine whether there exists a solution, and does not need to actually produce one. The constraint solver is based on the algorithm presented by Rehof and Mogensen [21].

4.1 Paragon implementation

We have implemented Paragon in a compiler that performs type checking for policies, and compiles policy-compliant programs into vanilla Java code. Once we know that a given program satisfies the intended information flow properties, we can safely remove all Paragon-specific type-level aspects of policies, locks and actors.

We must still retain the runtime aspects, such as querying the lock state and performing inequality comparisons between policies. The Paragon runtime library provides Java implementations for locks and properties, including operations for opening, closing and querying locks to which the Paragon `open`, `close` and query statements are compiled. Similarly, the library provides Java implementations for policies and operations for performing runtime inequality comparisons between them.

Compiler statistics Our Paragon compiler is written in Haskell and comprises roughly 16k lines of code, including comments. Approximately half of that code is due to our policy type checker, and only a small fraction, just over 600 lines of code, deals with generation of Java code and the Paragon interface files needed for modular compilation. On top of that, some 1500 lines of Java code are written for our runtime representations of Paragon entites. The compiler can be downloaded from our Paragon website [1], or from the central Haskell "hackage" repository using the command `cabal install paragon`.

Runtime overhead Supporting lock queries and policy comparisons at runtime yields a negligible overhead on Paragon programs. Most of the additional generated code handles the initalisation of policies and locks upon class or object instantiation, as well as the opening and closing of locks, which should not give any significant performance penalty. More involved are the lock queries and policy comparisons themselves since they resemble essentialy Datalog program evaluation and respectively containment [29]. However, our experience shows that clause bodies consist of just a few atoms, and have yet to find an example involving locks with arity higher than two, so in practice we posit that this overhead is negligible as well.

5 Case Studies

We put the compiler to the test with two case studies, both based on applications written in the Jif programming language, to which we further relate in §6.

Mental Poker In [3], a non-trivial cryptographic protocol for playing online poker without a trusted third party is implemented in Jif. During the distribution of the cards, players communicate cards encrypted with a per-player, per-game symmetric key. That is, the receiving player cannot decrypt the received card. At the end of the game the players reveal their symmetric key such that the other player may verify the outcome of the card distribution. For the purpose of non-repudiation each player signs outgoing messages with her private key.

From an information-flow perspective we desire an implementation of this protocol to satisfy various policies. The public key of a player is visible to everyone, as it is used to verify the player's signatures, but the private key should never leave the player's client. The cards to be communicated should not be sent before they are encypted with the symmetric key and then signed. The symmetric key should remain confidential to the player until the end of the game.

The value of the symmetric key leaks partially when performing encryption. In our Paragon implementation (6.5k lines) this leak is controlled by a lock private to the class performing the encryption, similar to the approach taken in the sanitiser class from the example in §3. That is, the class ensures that only the result of the operation is released and not the value of the key involved. The symmetric key is protected with a policy guarded by this private lock. A similar approach is used to protect the private key, to only reveal the outcome of the signing operation in which it was involved. The cards to be encrypted are protected with both the private locks of the encryption and the signing operation, indicating that they have to go through both declassifiers before they can be sent to the other player. The symmetric key is also allowed to be released when the `EndGame` lock is open. That is, this lock is used to represent a policy-relevant state of the application.

By constrast, Jif uses owner-based policies. The Jif policies here can simply state whether the data is owned by a given player or not, and cannot, in an obvious way, express anything beyond that. The fact that a Jif program has access to exactly one declassification mechanism prevents it from distinguishing or controlling different forms of declassification. In this case study it cannot make a distinction between declassifications that are allowed due to encryption, and those due to signing. In addition, Jif does not provide a means to write temporal policies and needs to rely on programming patterns to prevent declassifications occuring in a state where they are not supposed to be allowed.

JPMail The second case study implements a functional e-mail client based on JPMail [11]. In JPMail the user can specify a mail-policy file, partly dictating the information-flow policies that the mail client has to enforce.

JPMail ensures that an e-mail is only sent if its body has been encrypted under an algorithm that is trusted by the receiver of the e-mail. Which encryption algorithms are trusted by what JPMail users is specified in the mail-policy file. In addition JPMail needs to enforce more static policies, e.g. preventing the login credentials from flowing anywhere else than to the e-mail server.

In the Paragon implementation (2.6k lines) these latter, static policies are easily modelled as specifying only the e-mail server as a receiving actor. The partly dynamic policy on the e-mail body is represented by a set of clauses of the form:

```
(User u) server: Receiver(mail, u), AESEncrypted(mail), TrustsAES(u)
```

That is, the e-mail can be sent to the mail server only if it has been encrypted under AES and the receiver of that e-mail trusts AES encryption. The `TrustsAES` and similar locks representing the user-specific policies are opened after parsing the mail-policy file, during initialisation of the client. The `Receiver` lock is opened based on the To-field information, and the `AESEncrypted` lock is encapsulated analogous to the encryption / signing locks of the previous case study.

The issues for the Jif implementation in the mental poker case study show up in the JPMail example [11] as well. Moreover, stateful policies are central to this example and are challenging to model in Jif; Hicks *et al*'s solution involves generating the policy part of the Jif source code from the mail-policy file, hardcoding the user-specific policies in the client. This implies that if a mail-policy file changes, the only way for the Jif solution to handle it is by recompilation of the code. By contrast, Paragon handles policy change mechanisms naturally (by opening and closing locks) without stopping the code or recompiling.

6 Related Work

In this section we consider the related work on languages and language support for expressive information flow policies. We focus on actual systems rather than theoretical studies on policy mechanisms and formalisms. We note, however, that there are several policy languages in the access control and authorisation area which have some superficial similarity with the Paragon Policy Language, since they are based on datalog-like clauses to express properties like delegation and roles, see e.g. [4,9,13,14]. Key differences are (i) the information flow semantics that lies at the heart of Paragon, and (ii) the fact that the principal operation in Paragon is comparison and combination of policies, whereas in the aforementioned works the only operation of interest is querying of rules.

Languages with explicit information-flow tracking Two "real-sized" languages stand out as providing information-flow primitives as types, namely FlowCAML and Jif – as discussed in the introduction.

Comparing Paragon to Jif is inevitable, being at the same time a competitor and a source of inspiration. Due to the unique position Jif has enjoyed in the domain of information flow research over many years, much research has been done using Jif for context and examples. It is thus natural to ask how research done on or with Jif can carry over to Paragon.

The main advantage of Paragon over Jif is undoubtedly the flexibility of the concept of locks, including their stateful nature. Where Jif has a single declassify construct, Paragon can provide different declassifying methods to work on different data, as needed by the domain at hand, and relate that declassification to the state of the program. Jif rigidly builds in some stateful aspects in the form of authority and delegations, which in Paragon would just be special cases of working with locks.

In many aspects, our work on Paragon has greatly benefitted from Jif's trailblazing, as well as research done in the context of Jif. Policy defaulting mechanisms, handling of runtime policies, and having all exceptions checked, are all features where we have been able to adopt Jif's solution directly.

In separate work, as of yet unpublished, we have conducted a complete and in-depth comparison between the two languages and all their features, including a Paragon library that gives a complete implementation of Jif, but the full details of that comparison are out of scope for this paper.

Compilers performing IF tracking Information flow tracking can be performed in a language which has no inherent security policies, lattice-based or otherwise. In such a setting one tracks the way that information flows from e.g. method parameters to outputs. Examples of tools performing such analysis are the Spark Examiner operating over a safety-critical subset of Ada [6], and Hammer and Snelting [10] explain how state-of-the-art program slicing methods can support a more accurate analysis of such information flows in Java (e.g. both flow sensitive and object sensitive).

Dynamic Information Flow Tracking with Expressive Policies Runtime information flow tracking systems have experienced a recent surge of interest. The most relevant examples from the perspective of the present paper are those which perform full information flow tracking (rather than the semantically incomplete "taint analysis"), and employ expressive policies. The first example is Stefan *et al*'s embedding of information flow in Haskell [25]. In principle one could also use Paralocks in a dynamic context, and we are currently investigating a stateful extension of their LIO framework which could be instantiated with the generalised Paralocks described in this work.

Yang *et al*'s *Jeeves* language [31] focusses on confidentiality properties of data expressed as context-dependent data visibility policies. The Jeeves approach is noteworthy in it's novel implementation techniques and greater emphasis on the separation of policy and code.

Encoding Information Flow Policies with Existing Type Systems With suitably expressive type systems and abstraction mechanisms, static information flow constraints can be expressed via a library [15,16,22].

A number of recent expressive languages are aimed at expressing a variety of rich security policies, but do not have information flow control as a primitive notion (as Paragon or Jif) [12,28]. F* [27], a full-fledged implementation of a dependently typed programming ML-stye programming language, is perhaps the most successful in this class, with a large number of examples showing how security properties can be encoded and verified by typing.

Typestate The way that Paragon tracks locks is related to the concept of *typestate* [26]. Typestate acknowledges that the runtime state of e.g. an object often determines which methods are safe to call. For example, for a Java `File` object, the method `read()` can only be called if the file has first been opened with the `open()` method. Systems with typestate, such as *Plaid* [2], support formal specification of typestate properties, and enforce that programs correctly follow the specifications. In Paragon, typestate properties can be specified through the use of lock state modifiers. Paragon cannot express features that depend on Plaid's first-class states, e.g. "an array of open files", but can otherwise express solutions to their motivating examples.

7 Conclusions and Further Work

It is our expectation that one day programming languages with built-in support for expressing and enforcing information-flow concerns become widely deployed. Paragon's strong integration with Java and its relatively natural yet expressive policy specification language lowers the threshold for adopting information-flow aware programming outside the research community. Still, much work is left to be done before Paragon can become a serious competitor to existing programming.

One notable direction for future work in the Paragon language is concurrency support. This direction requires both theoretical and practical work, in particular if declassification mechanisms are shared among threads. Another planned direction is to present a more substantial formalisation of Paragon's type system, including a proof of soundness with respect to information flow security.

Acknowledgments Thanks to the reviewers and all members of the ProSec group at Chalmers for improving this paper with every iteration. We thank our colleagues Wolfgang Ahrendt, Pablo Buiras, Filippo Del Tedesco, Willard Rafnsson and Alejandro Russo for their valuable comments and feedback. Furthermore we would like to thank Jens Lideström, Shayan Najd Javadipour, Javed Nazir and Yannick Zakowski for assisting in the implementation of the Paragon compiler. This work is partly funded by the Swedish agencies SSF and VR, and Websand EC FP7-ICT-STREP.

References

1. Paragon website. `http://www.cse.chalmers.se/research/group/paragon`, July 2013.
2. J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *OOPSLA Companion*, pages 1015–1022, 2009.
3. A. Askarov and A. Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. In *Proc. European Symp. on Research in Computer Security*, volume 3679 of *LNCS*. Springer-Verlag, 2005.
4. M. Y. Becker, C. Fournet, and A. D. Gordon. Design and semantics of a decentralized authorization language. In *Proc. IEEE Computer Security Foundations Symposium*, pages 3–15. IEEE Computer Society, 2007.
5. N. Broberg and D. 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.
6. R. Chapman and A. Hilton. Enforcing security and safety models with an information flow analysis tool. *ACM SIGAda Ada Letters*, 24(4):39–46, 2004.
7. D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.
8. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.
9. D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *IJCAR*, volume 4130 of *LNCS*. Springer, 2006.
10. C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009.
11. B. Hicks, K. Ahmadizadeh, and P. D. McDaniel. From languages to systems: Understanding practical application development in security-typed languages. In *ACSAC*. IEEE Computer Society, 2006.
12. L. Jia and S. Zdancewic. Encoding information flow in aura. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, 2009.
13. T. Jim. SD3: A trust management system with certified evaluation. In *Proc. IEEE Symp. on Security and Privacy*, 2001.
14. N. Li, J.C. Mitchell, and W.H. Winsborough. Design of a role-based trust-management framework. In *IEEE Symposium on Security and Privacy*, pages 114–130, 2002.
15. P. Li and S. Zdancewic. Arrows for secure information flow. *Theor. Comput. Sci*, 411(19), 2010.
16. J. Morgenstern and D. R. Licata. Security-typed programming within dependently-typed programming. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional Programming*, 2010.

17. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, January 1999.
18. 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.
19. A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.
20. 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`, 2001–2013.
21. J. Rehof and T. Mogensen. Tractable constraints in finite semilattices. In Radhia Cousot and DavidA. Schmidt, editors, *Static Analysis*, volume 1145 of *Lecture Notes in Computer Science*, pages 285–300. Springer Berlin Heidelberg, 1996.
22. A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in haskell. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell*, 2008.
23. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 15(5):517–548, 2009.
24. V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml`, July 2003.
25. D. Stefan, A. Russo, J. C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM symposium on Haskell*, 2011.
26. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.
27. N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bharagavan, and J. Yang. Secure distributed programming with value-dependent types. In *The 16th ACM SIGPLAN International Conference on Functional Programming*, 2011.
28. 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.
29. B. van Delft, N. Broberg, and D. Sands. A datalog semantics for paralocks. In Audun Jøsang, Pierangela Samarati, and Marinella Petrocchi, editors, *Proceedings of the 8th International Workshop on Security and Trust Management (STM)*, pages 305–320. Springer, 2012.
30. J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 187–206. ACM, 1999.
31. J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2012.
32. L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6, 2007.

Online resources

Additional material can be found on the Paragon website

`http://www.cse.chalmers.se/research/group/paragon`

Here can be found:

- The Paragon compiler (source and binary)
- Various example programs, including the one presented in this paper
- The case studies from § 5

- Earlier papers on the Paralocks policy language and semantics

The Paragon compiler can also be tested online in a browser. We provide an interface and some tutorials on the Paragon language on

`http://teachmeparagon.nowplea.se`

A The Generalised Flow Locks Framework

We introduce the basic building blocks in the policy language framework: actors, locks, clauses and distinguished actor sets.

Actors ranged over by a, b, etc. are policy-relevant entities with information flow concerns. Unlike Paralocks, in this framework we are agnostic to how actors are represented concretely. For the purposes of information flow analysis, the only relevant information associated with an actor is its identity, and the only criteria that an actor representation needs to fulfil are that actor identifiers are:

- abstract, in the sense that it is not possible to create new actor identifiers in a way not presented by the domain,
- opaque, in the sense that nothing can be assumed on the implementation of the identifier, and
- immutable, in the sense that the same actor identity always identifies the same actor.

In previous work we used these criteria constructively, to generate an abstract set of actor identities. Here we present a more general view, allowing us to use already exisiting entities as actors.

Distinguished actor sets ranged over by A, B, etc. are subsets on the domain of actors $\mathcal{A}$. Not all actor sets may be valid in a given instantiation of the framework, so the set of distinguished actor sets $\mathcal{D}$ is a subset of $\mathcal{P}(\mathcal{A})$. We futher assume that the join ($\sqcup$) is well-defined on $\mathcal{D}$, possibly resulting in an empty set, $\bot$ (note: the join of two sets does not have to be the exact intersection of these sets). Where it exists, we use $\top$ to denote the superset containing all actors.

Locks denote policy-relevant facts. A *lock family* is a predicate $\sigma(A_1, \dots, A_n)$ over actor sets A_i. A *lock* is a term $\sigma(a_1, \dots, a_n)$ where $a_i \in A_i$. For a given assignment of truth values to these terms we say that a lock is *open* if it is assigned true, and it is *closed* otherwise. A *lock state* is such an assignment represented as a set of all the locks that are open. We say that arity$(\sigma) = n$. Σ, Σ', etc. range over sets of locks (not necessarily of the same family).

Clauses ranged over by c_1, c_2, etc. are terms of the form

$$\forall a \in A, \forall \langle b_1, \dots, b_n \rangle \in B_1 \times \cdots \times B_n . a : \Sigma$$

where the locks in Σ may contain one or more of the quantified variables. We refer to a as the head, and Σ as the body of the clause. Intuitively, actors in the head of a clause can be seen as observers to whom information may flow, in the presence of some assignment in which all predicates in the body hold (for some instantiation of b_i), i.e. a lock state in which all locks in Σ are open.

Policies are sets of clauses, denoted by $\{c_1; \ldots; c_2\}$.

Lock properties are rules of the form

$$\forall \langle a_1, \ldots, a_n \rangle \in A_1 \times \cdots \times A_n, \\ \forall \langle b_1, \ldots, b_m \rangle \in B_1 \times \cdots \times B_m. \\ \sigma(a_1, \ldots, a_n) : \Sigma$$

that allow for locks to be implicitly opened. For example, the property $\forall x \in A, \forall y \in B.L(x) : M(x, y)$ under any lock set automatically opens the lock L for each actor in set A, for which the lock M in relation to some $y \in B$ is open. A set of lock properties is referred to as the *global policy* G.

A.1 Semantics and Operations in GFL

In [29] we presented a semantics for Paralocks based on Datalog, a logic programming based query language [CGT89]. We update this semantics to match the Generalised Flow Locks framework. In short, we reserve a unary predicate $Flow$ and transform policies into Datalog programs, where the head a of the clause becomes a conclusion of the fact $Flow(a)$. The semantics of a policy is then simply defined as the set of actors on which its Datalog transformed form can conclude the $Flow$ predicate.

We define the semantics of the necessary lattice operations in similar terms. The meet ($\sqcap$) of two policies is the union of $Flow$ facts derived by each individual policy, whereas the join ($\sqcup$), being the dual, matches the intersection. One policy is ordered less restrictive than another policy ($\sqsubseteq$) if its set of $Flow$ facts form a superset of the other.

We define the semantics of policies and the operations on them in more detail in appendix B, including the presence of a global policy and lock states. In appendix C we show that our implementations of the lattice operations are sound for the GFL framework, and sound and complete for the framework's specialization to the Paragon policy language, which is discussed next.

B Flow Locks framework: semantics and operations

B.1 Flow Locks Semantics

In [29] it was identified that the high similarity with Datalog programs provides a natural semantics for Paralocks policies. A Datalog program is a set of declarative logic rules where function symbols are absent, forming a decidable fragment of Horn logic. We shortly define the Datalog semantics and then provide a semantics to Flow Locks policies by transforming them into Datalog programs.

In Datalog an *atom* is of the form $R(\boldsymbol{T})$ and if the vector $\boldsymbol{T}$ does not contain any free variables, the atom is called a *fact*. A set of facts is called an *interpretation*. A *rule* is of the form $R_0(\boldsymbol{T}_0)$:– $R_1(\boldsymbol{T}_1), \ldots, R_n(\boldsymbol{T}_n)$ and a *program* is a set of these rules.

Definition 1 (Direct Consequence Operator). *The* direct consequence operator δ *takes a Datalog program Π, an interpretation I and produces another interpretation $\delta(\Pi, I) = I'$. The interpretation I' is computed as $I' = I \cup N$. For each rule $R_0(\boldsymbol{T}_0)$:− $R_1(\boldsymbol{T}_1), \ldots, R_n(\boldsymbol{T}_n) \in \Pi$ and any ground substitution θ such that $\forall i > 0 . R_i(\boldsymbol{T}_i)\theta \in I$, we have $R_0(\boldsymbol{T}_0)\theta \in N$.*

Intuitively δ can be thought of as 'applying all rules once' on a database of facts, and returning the old database extended with the newly derived facts.

Definition 2 (Models, $\models$). *A model for a Datalog program Π is an interpretation M such that $\delta(\Pi, M) = M$. A model is called* minimal *iff $M \subseteq M'$ for any model M' of Π. We write $\Pi \models f$ iff $f \in M$ and M is a minimal model for Π.*

To guarantee that we do arrive at a minimal fixed point each rule has to be *safe*: each variable appearing in the head of the rule must appear in the body of the rule as well.

Returning to Flow Locks, we transform a Flow Locks policy to a Datalog program as follows. Given S the set of defined actor sets, we assume the presence of a reserved predicate (lock) $\mathit{inA}_i(A_i)$ for each $A_i \in S$ that is open for exactly those actors that make up the set A_i. We further reserve the unary lock $\mathit{Flow}(\top)$ which can be opened for any actor. Then the function τ converts clauses and lock properties to Datalog rules, (global) policies and lock states to Datalog programs as follows:

$$\begin{aligned}
&\tau(\forall a \in A : \Sigma) = \mathit{Flow}(a) \text{ :− } \mathit{inA}(a), \Sigma \\
&\tau(\forall \langle a_1, \ldots, a_n \rangle \in A_1 \times \cdots \times A_n . \sigma(a_1, \ldots, a_n) : \Sigma) = \\
&\qquad \sigma(a_1, \ldots, a_n) \text{ :− } \mathit{inA}_1(a_1), \ldots, \mathit{inA}_n(a_n), \Sigma \\
&\tau(p) = \{\tau(c) \mid c \in p\} \qquad \tau(G) = \{\tau(lp) \mid lp \in G\} \\
&\tau(LS) = \{\sigma(\boldsymbol{a}) \text{ :− } \varnothing \mid \sigma(\boldsymbol{a}) \in LS\}
\end{aligned}$$

For brevity we omit the variables only occurring in clause bodies. Note that the $\mathit{inA}(a)$ predicate guarantees that all rules in the resulting program are safe.

We now define the semantics of a policy p in the presence of global policy G and lock state LS as:

$$[\![p]\!]^G_{LS} = \{a \mid \tau(p) \cup \tau(G) \cup \tau(LS) \models \mathit{Flow}(a)\}$$

That is, a policy can be semantically considered as the set of actors for which it can derive the Flow predicate, using additional rules in G and facts in LS.

B.2 Lattice operations

With this semantics for policies, the semantics for the lattice of Flow Locks policies is readily defined (we omit the G and LS annotations):

$$\begin{aligned}
[\![p \sqcup q]\!] &= [\![p]\!] \cap [\![q]\!] \\
[\![p \sqcap q]\!] &= [\![p]\!] \cup [\![q]\!] \\
[\![p \sqsubseteq q]\!] &= [\![p]\!] \supseteq [\![q]\!]
\end{aligned}$$

For the meet and join operation we can provide the following algorithms.

Definition 3 (Meet). *Since policies are sets of clauses, we simply compute $p \sqcap q := p \cup q$.*

Definition 4 (Join). *The join $p \sqcup q$ is the cross-product of clauses in p and q, with the join of two individual clauses computed as:*

$$\begin{aligned}&(\forall a \in A_p, b_{pi}{\in}B_{pi}.a : \Sigma_p) \sqcup (\forall a \in A_q, b_{qi}{\in}B_{qi}.a : \Sigma_q)\\&\quad = (\forall a \in A_p \sqcup A_q, b_{pi}{\in}B_{pi}, b_{qi}{\in}B_{qi}.a : \Sigma_p \cup \Sigma_q)\end{aligned}$$

For the purpose of the proof of completeness, we remove those clauses where $A_p \sqcup A_q = \varnothing$.

For the policy ordering operation we use the uniform containment operation for Datalog programs, as proposed by Sagiv in [Sag88].

Definition 5 (Policy Ordering). *We compute $p \sqsubseteq_{G,LS} q$ as $\tau(q) \sqsubseteq^U \tau(p) \cup \tau(G) \cup \tau(LS)$.*

By [Sag88], $\Pi_1 \sqsubseteq^U \Pi_2$ iff for each rule $R_0(\boldsymbol{T}_0) :\!-\, \Sigma \in \Pi_1$ the following holds. Let θ be a substitution replacing all variables in the rules with skolem constants. It should hold that $\Pi_2 \cup \Sigma\theta \models R_0(\boldsymbol{T}_0)\theta$.

All three operations were shown to be sound and complete for Paralocks. In the presence of distinguised actor sets we lose completeness of the join operator and the ordering comparison. However we regain that completeness when using Java's class hierarchy for the actor sets, as we do in Paragon. The soundness and completeness, in the Paragon setting, is proven in Appendix C.

It can be easily shown that both the Flow locks language from our early work [BS06], and its Paralocks [5] successor, are both specialisations of the Generalized Flow Locks Framework. We show the details in appendix B.3.

Logic-based languages with typing By extending the Flow Locks framework with actor sets our policy language goes beyond standard, single-domain Datalog. Adding types to Datalog or similar logic-based languages has been considered before. An early suggestion is an extension to Prolog by Mycroft and O'Keefe [MO84], based on Hindley-Milner. For Datalog Fruhwirth et al. [FSVY91] introduce a type system where types are presented as unary predicates, much like we use in the semantics of our policies. A similar approach is presented by Zook et al. [ZPSS09] but with a different, more efficient, way of enforcing the types. However, none of these perfectly matches the Flow Locks framework. The clauses in our policies have variables rather than predicates as heads, the types of which can be arbitrarily constrained. This property in particular does not yet seem to exist in other logic-based languages.

B.3 Flow Locks and Paralocks as instances of Generalised Flow Locks

Paralocks The only difference between Paralocks and GFL is the introduction of distinguished actor sets. Hence, by simply instantiating this with only one set, call it `Actor`, we get the same specification as Paralocks.

Flow Locks The previous Flow Locks language can be seen as an instance of Paralocks, and therefore by transitivity as an instance of the Flow Locks Framework. It simply is the same language as Paralocks, in which locks are unary and therefore no variables can occur in policies.

C Lattice operations proofs

In this appendix we show that our implementations of the lattice operations are both sound and complete in the setting of Paragon. To simplify the proofs we first introduce a tighter relation between the semantics of a policy and the direct consequence operator δ.

We introduce the assisting function

$$M(G, LS) = \{f \mid \tau(G) \cup \tau(LS) \models f\}$$

giving the minimal model of $\tau(G) \cup \tau(LS)$. By our restrictions, we know that this minimal model never contains a predicate on Flow. For any policy p, its transformation $\tau(p)$ only concludes facts on Flow. No rule in G has Flow in its body, so adding $\tau(p)$ to $\tau(G) \cup \tau(LS)$ will strictly extend the minimal model with facts on Flow. We can thus safely replace $\tau(G) \cup \tau(LS)$ with its minimal model as follows:

$$\begin{aligned} &\{f \mid \tau(p) \cup \tau(G) \cup \tau(LS) \models f\} = \\ &\qquad \{f \mid \tau(p) \cup \tau(M(G, LS)) \models f\} \end{aligned}$$

Furthermore, since Flow does not appear in the body of clauses in p, we have that

$$\{f \mid \tau(p) \cup \tau(M(G, LS)) \models f\} = \delta(\tau(p), M(G, LS))$$

This means that we can conclude:

$$[\![p]\!]^G_{LS} = \{a \mid \mathit{Flow}(a) \in \delta(\tau(p), M(G, LS))\} \tag{1}$$

C.1 Soundness and completeness of $\sqcap$

Since the meet of two policies is defined as being simply the union of the clauses in those policies, we have to show that

$$[\![p \cup q]\!]^G_{LS} = [\![p]\!]^G_{LS} \cup [\![q]\!]^G_{LS}$$

for any global policy G and any lock state LS. Let $m = M(G, LS)$, $\tau(p) = P$ and $\tau(q) = Q$. By definition of δ,

$$\delta(P \cup Q, m) = \delta(P, m) \cup \delta(Q, m)$$

From which we conclude:

$$\begin{aligned} &\{a \mid \mathit{Flow}(a) \in \delta(P \cup Q, m)\} = \\ &\qquad \{a \mid \mathit{Flow}(a) \in \delta(P, m)\} \cup \{a \mid \mathit{Flow}(a) \in \delta(Q, m)\} \end{aligned}$$

Which, by (1), is what we had to show.

C.2 Soundness and completeness of $\sqcup$

The join of two policies p, q is computed as $\{c_p \sqcup c_q \mid c_p \in p, c_q \in q\}$, where $\sqcup$ is defined on two clauses as

$$\begin{aligned}(\forall a \in A_p, b_{pi} \in B_{pi}.a : \Sigma_p) &\sqcup (\forall a \in A_q, b_{qi} \in B_{qi}.a : \Sigma_q)\\ = (\forall a \in A_p \sqcup A_q, b_{pi} &\in B_{pi}, b_{qi} \in B_{qi}.a : \Sigma_p \cup \Sigma_q)\end{aligned}$$

We show that this implementation is both sound and complete in the context of Paragon. That is

$$[\![p \sqcup q]\!]^G_{LS} = [\![p]\!]^G_{LS} \cap [\![q]\!]^G_{LS}$$

Let $m = M(G, LS)$, $\tau(p) = P$, $\tau(q) = Q$ and $\tau(p \sqcup q) = PQ$. By (1) proving this property is equivalent to showing that for any constant x:

$$\begin{aligned}&\mathit{Flow}(\mathrm{x}) \in \delta(P, m) \text{ and } \mathit{Flow}(\mathrm{x}) \in \delta(Q, m) \iff\\ &\qquad \mathit{Flow}(\mathrm{x}) \in \delta(PQ, m)\end{aligned}$$

Soundness ($\Leftarrow$) By having $\mathit{Flow}(\mathrm{x}) \in \delta(PQ, m)$ and by the definition of $\sqcup$ and τ we know that

$$\begin{aligned}\exists r_{pq} \in PQ \text{ such that } r_{pq} &= \mathit{Flow}(a) \text{ :- } \mathit{inA}_{pq}(a), \Sigma_p, \Sigma_q\\ &\text{and } \exists \theta_{pq} \text{ such that } a\theta_{pq} = \mathrm{x} \text{ and}\\ &\{\mathit{inA}_{pq}(\mathrm{x})\} \cup \Sigma_p\theta_{pq} \cup \Sigma_q\theta_{pq} \subseteq m\\ \exists r_p \in P \text{ such that } r_p &= \mathit{Flow}(a) \text{ :- } \mathit{inA}_p(a), \Sigma_p\\ \exists r_q \in Q \text{ such that } r_q &= \mathit{Flow}(a) \text{ :- } \mathit{inA}_q(a), \Sigma_q\end{aligned}$$

Where $A_{pq} = A_p \sqcup A_q$. Hence by having the assumption that the inA locks are always appropriatly opened, we can deduce from $\mathit{inA}_{pq}(\mathrm{x}) \in m$ that also $\mathit{inA}_p(\mathrm{x}) \in m$ and $\mathit{inA}_q(\mathrm{x}) \in m$. Thus, the rules r_p and r_q can each derive the fact $\mathit{Flow}(\mathrm{x})$ given m, and therefore $\mathit{Flow}(\mathrm{x}) \in \delta(P, m)$ and $\mathit{Flow}(\mathrm{x}) \in \delta(Q, m)$ which is what we had to show.

Completeness ($\Rightarrow$) By having $\mathit{Flow}(\mathrm{x}) \in \delta(P, m)$ and $\mathit{Flow}(\mathrm{x}) \in \delta(Q, m)$ we know that

$$\begin{aligned}\exists r_p \in P \text{ such that } r_p &= \mathit{Flow}(a) \text{ :- } \mathit{inA}_p(a), \Sigma_p \text{ and}\\ &\exists \theta_p \text{ such that } a\theta_p = \mathrm{x} \text{ and}\\ &\{\mathit{inA}_p(\mathrm{x})\} \cup \Sigma_p\theta_p \subseteq m\\ \exists r_q \in Q \text{ such that } r_q &= \mathit{Flow}(a) \text{ :- } \mathit{inA}_q(a), \Sigma_q \text{ and}\\ &\exists \theta_q \text{ such that } a\theta_q = \mathrm{x} \text{ and}\\ &\{\mathit{inA}_q(\mathrm{x})\} \cup \Sigma_q\theta_q \subseteq m\end{aligned}$$

By the definition of $\sqcup$ and τ, we also know that there has to be a rule $r_{pq} \in PQ$ such that $r_{pq} = \mathit{Flow}(a) \text{ :- } \mathit{inA}_{pq}(a), \Sigma_p, \Sigma_q$ where $A_{pq} = A_p \sqcup A_q$. Since we do not constrain the join operation on actor sets to be the exact intersection of sets, it could be that x is in the set A_p and the set A_q, but not in the set A_{pq}. Therefore, this implementation operation is not complete in general. However, in Paragon we use the

Java class hierarchy in which we do always get the exact intersection of sets. To be precise A_{pq} is always either the same as A_p or as A_q. That is, in order for an element to be both in A_p and A_q, one class has to be a subtype of the other (consider for example $\texttt{File} \sqcup \texttt{Object} = \texttt{File}$). Without loss of generality we say that $A_{pq} = A_p$. Then, assuming disjointness of variables in Σ_p and Σ_q apart for a, we construct $\theta_{pq} = \theta_p \cdot \theta_q$. We now have that $\{inA_{pq}(a)\theta_{pq}\} \cup \Sigma_p\theta_{pq} \cup \Sigma_q\theta_{pq} \subseteq m$. Therefore, r_{pq} can derive $Flow(a)$ on m, and thus $Flow(a) \in \delta(PQ, m)$ – which is what we had to show.

C.3 Soundness and completeness of $\sqsubseteq$

(*Sketch*) In [Sag88] Sagiv shows that the uniform containment operation is both sound and complete, if in $\Pi_1 \sqsubseteq^U \Pi_2$ program Π_1 is a non-recursive Datalog program. This is the case for our policy ordering check (definition 5), since $\Pi_1 = \tau(p)$, which cannot be recursive.

Sagiv's proof operates on a standard, *single domain* Datalog setting and consists of showing that for each database D for which Π_1 can derive some fact, Π_2 should be able to derive that same fact. In the Flow Locks framework we have multiple domains but translate this back to single-domain Datalog by modelling each domain as a reserved predicate. Therefore, Sagiv's proof will consider databases that are not possible to exist in the Flow Locks setting and is thus not complete.

For example, consider the domain of numbers, which we divide in the sets Neg of negative numbers, $\mathbb{N}$ of natural numbers and $\mathbb{Z}$ of integers. Note that $Neg \cup \mathbb{N} = \mathbb{Z}$. We define two policies p and q and their transformations:

$$\begin{aligned} p =&\quad \{\forall a : Neg.a; \forall a : \mathbb{N}.a\} \\ \tau(p) =&\quad \{Flow(a) :\!-\ inNeg(a); Flow(a) :\!-\ in\mathbb{N}(a)\} \\ q =&\quad \{\forall a : \mathbb{Z}.a\} \\ \tau(q) =&\quad \{Flow(a) :\!-\ in\mathbb{Z}(a)\} \end{aligned}$$

To compare $\tau(p) \sqsubseteq^U \tau(q)$ we could assume the database $D = \{inNeg(\mathrm{x})\}$, which is a valid database in Datalog. $\tau(p)$ can derive the fact $Flow(\mathrm{x})$ from this, $\tau(q)$ can not. Therefore the check concludes that the ordering does not hold. However, on the Flow Locks level we know that the database D can never exist, since every negative number is an integer, and we assume the approriate locks to be open. I.e., in Flow Locks, if $inNeg(\mathrm{x})$ is open, then $in\mathbb{Z}(\mathrm{x})$ is known to be open as well. To enforce this is standard Datalog we add appropriate lock properties.

In the other direction however, this does not work. To compare $\tau(q) \sqsubseteq^U \tau(p)$ we could assume the database $D = \{in\mathbb{Z}(\mathrm{x})\}$ from which $\tau(q)$ can derive $Flow(\mathrm{x})$, but $\tau(p)$ cannot. Again, we know that in Flow Locks D cannot exist, because either the lock for negative number or for natural numbers has to be open as well. But here we cannot use lock properties to model this – it is therefore that the uniform containment check is not complete in the general Flow Locks context.

Fortunately in Paragon we have instantiated the distinguished actor sets by Java's class hierarchy. And since any class can be extended we never arrive at a situation such as described here where we know that certain subclasses entirely make up the superclass. Therefore, the uniform containment check is still complete when specializing Flow Locks to Paragon's policy language.

D Additional Language Features

In this section we present some more technical features of the Paragon language.

D.1 Type parameters

Java, since the introduction of "Generics" in Java 5.0, allows types and methods to be parametrised by types, giving Java parametric polymorphism. Paragon introduces several new entities – actors, policies and locks – that affect typing in various ways. It is natural to extend the polymorphism to also include these aspects.

Thus in Paragon ordinary reference types have the implicit kind *type*. Type parameters of kind *type* need not be annotated, like in vanilla Java. For the Paragon-specific entities we introduce *kind annotations*, to separate them from each other and from ordinary types. type as its kind. For policies we can simply reuse their type as kinds as well. We can do the same for locks though we need to be able to parametrise over not just single locks, but rather sets of locks. To avoid introducing new keywords, we reuse the syntax for arrays for this purpose, i.e. the kind annotation on parameters taking sets of locks is **lock**`[]`. The following example represent a class that is parametrised on one actor of kind `File`, a policy and a base type:

```
public class C<File f, policy p, G> { ... }
```

The names `f`, `p` and `G` can be used inside the class body wherever members of their kind can occur. In addition, `f` is also passed as a regular instance of the `File` class.

D.2 Exceptions and indirect control flow

The static policy type system in Paragon tracks two kinds of information flows: direct flows arising from assignments, and indirect flows arising from control flow. It makes no attempt to track flows arising from termination – it is *termination insensitive*. If exceptions could not be caught, an exception would be the same as (premature) termination, which means we would not need to care about them. However, the catch mechanism makes exceptions rather a kind of control flow primitive, needing special attention.

In Java, subclasses of `RuntimeException` are *unchecked*. This means that methods need not declare if they could terminate with such an exception. Examples of runtime exceptions are `ArithmeticException` which for instance arises from division by 0, `ArrayOutOfBoundsException` and `NullPointerException`.

It should be obvious that any exception that can be caught is a potential channel for information flow, which means that in Paragon all exceptions must be checked. This in turn implies the need for analyses that can rule out the possibility of exceptions, in particular a null pointer analysis is needed to avoid that every instance field or method use incurs the need to declare a possible `NullPointerException`.

E Paragon type system

In this appendix we give the type system for the parts of the analysis corresponding to lock state approximation and policy constraint generation, for a sizeable subset of Java. Our implementation covers a larger subset still, but here we leave out a number of features that do not add anything to the presentation. The features left out are enums, static fields, arrays (as in the syntactic sugar), inner classes, casts, most operators, labeled statements, as well as expressions and statements whose typing would be very similar to those already covered (e.g. `do` is very similar to `while`).

Lock state approximation Let us first look at lock states, and define some convenient operations. First, in our analysis a lock state is represented concretely as the set of locks being open in that state. We will also have use for the concept of an unreachable state, represented with the distinguished value $\bot$. We let LS range over values in the domain of lock states including $\bot$.

Next, we will have use for the concept of a *lock modification*, representing a set of changes to be applied to a lock state. We model this concretely as a tuple (L_o, L_c), where L_o is a set of locks to be opened, and L_c is a set of locks to be closed. Here we overload $\bot$ and identify it with the tuple $(\varnothing, \varnothing)$, i.e. no modifications.

We now need two (overloaded) operations, $\triangleright$ and $\diamond$, respectively corresponding to sequential and parallel composition of lock states and lock modifications. We have that:

$$\begin{aligned}
LS_1 \diamond LS_2 &= LS_1 \cap LS_2 \\
LS \triangleright (L_o, L_c) &= (LS \setminus \{L(a_1, \dots, a_n) \mid L(b_1, \dots, b_n) \in L_c, b_i \simeq a_i\}) \cup L_o \\
(L_{o1}, L_{c1}) \diamond (L_{o2}, L_{c2}) &= (L_{o1} \cap L_{o2}, L_{c1} \cup L_{c2}) \\
(L_{o1}, L_{c1}) \triangleright (L_{o2}, L_{c2}) &= (L_{o2} \cup (L_{o1} \triangleright (\varnothing, L_{c2})), L_{c2} \cup (L_{c1} \setminus L_{o1}))
\end{aligned}$$

The above definitions hold when the lock state operands are not $\bot$. We have that $\bot$ is the identity element for $\diamond$, and the 0-element (annihilator) when appearing as the left-hand operand of $\triangleright$.

Now we can turn to the analysis itself. The behaviour of the annotating analysis is captured by the following declarative judgment, covering both expressions and statements:

$$LS \vdash e \rightsquigarrow LS', X$$

where

- LS and LS' are the lock states prior to and after evaluating e,
- X maps exception types (and pseudo-exceptions) to the delayed lock states that will hold once the appropriate catch-clause is reached. If no exceptions can be caused by the term under scrutiny, we omit X in the rule. We let X be total, where for every exception type τ not explicitly mentioned, $X(\tau) = \bot$.

We also assume a global environment M, which is a mapping from reference types and method names to a four-tuple, where the first three components are the locks that method respectively expects, opens and closes, and the final component, LX, is a mapping from exception types to lock modifications should the method terminate exceptionally. Again we let LX be total such that for every exception type τ not mentioned, $LX(\tau) = \perp$.

Further, a rule for this judgment is valid only if each occurence of the mentioned language constructs is annotated with the correct lock state.

We will consistently use superscripts (sometimes in conjunction with square brackets for scoping) to denote annotated information, while subscripts or primes serve to disambiguate terms. Technically, an expression e will also be annotated with information from previous passes. We will omit such annotations when they are not needed by the rule.

Looking at expressions first, we omit the rules for literals, the expressions **`this`** and **`null`**, and variables; they do nothing interesting in this context. We further omit the rules for binary operators as they do nothing but inductively call their sub-components, and for instance creation which is very similar to method calls. This leaves the rules for method calls, assignments, and the conditional `?:` operator. The rule for assignments is as follows, where the only interesting part to note is the lock state that the assignment gets annotated with, which will later be used by the constraint generation.

$$\frac{LS \vdash e \rightsquigarrow LS_e; X}{LS \vdash x =^{LS_e} e \rightsquigarrow LS_e; X}\ \text{(Assign)}$$

This rule deals with variable assignment. Nothing changes for field assignments; for primary field assignments (where the object is calculated from an expression) or array updates, the only difference is that the sub-expressions of those left-hand sides are analysed first.

The rule for method calls is more involved:

$$\frac{\begin{array}{c} LS \vdash e \rightsquigarrow LS_0; X_0 \\ LS_{i-1} \vdash e_i \rightsquigarrow LS_i; X_i \\ M(\tau, m) = (L_e, L_o, L_c, LX) \quad L_e \subseteq LS_n \end{array}}{LS \vdash e^{\tau}.m(e_1, \ldots, e_n)^{LS_n} \rightsquigarrow LS'; X'}\ \text{(Call)}$$

where $LS' = LS_n \rhd (L_O, L_C)$, and $X' = X_0 \diamond \ldots \diamond X_n \diamond LX$. Again, note the lock state annotation method call. As noted, the rule for instance creation is near identical (assuming that M also stores lock state signatures for constructors).

Finally (for expressions), the rule for the conditional operator:

$$\frac{\begin{array}{c} LS \vdash e_c \rightsquigarrow LS_c; X_c \\ LS_c \rhd L_i \vdash e_i \rightsquigarrow LS_i; X_i \end{array}}{LS \vdash e_c ?^{(L_1, L_2)} e_1 : e_2 \rightsquigarrow LS_1 \diamond LS_2; X_c \diamond X_1 \diamond X_2}\ \text{(Cond)}$$

The interesting thing to note here is that we expect the condition to be annotated, from a previous phase, with any locks known to be open or closed if the condition is true and false, respectively, and we take this into account when analysing and annotating the expressions in the branches.

Turning to statements, empty statements and expression statements add nothing interesting, and **`if`** statements are very similar to the conditional operator, so we omit these. Further, many of the pseudo-exceptions all follow the same pattern: **continue**, **break** and **return** without a value – we show only the first of these. Apart from these we have **while**, **throw**, **return** *with* a value, **try-catch-finally**, and the Paragon-specific **open** and **close** statements.

The rules for **open** and **close** were already shown in the main paper, but we repeat them here for completeness:

$$\frac{}{LS \vdash \texttt{open}\ L(a_1,\ldots,a_n); \rightsquigarrow LS \cup \{L(a_1,\ldots,a_n)\}}\ \text{(Open)}$$

$$\frac{}{LS \vdash \texttt{close}\ L(a_1,\ldots,a_n); \rightsquigarrow LS \setminus \{L(b_1,\ldots,b_n) \mid b_i \simeq a_i\}}\ \text{(Close)}$$

The rule for **while**-loops is as follows:

$$\frac{\begin{array}{c} LS \diamond LS_s \diamond X_s(\texttt{continue}) \vdash e_c \rightsquigarrow LS_c; X_c \\ LS_c \rhd L_t \vdash s \rightsquigarrow LS_s; X_s \end{array}}{LS \vdash \texttt{while}^{(L_t,L_f)}\ (e_c)\ \texttt{do}\ s \rightsquigarrow (LS_c \cup L_f) \diamond X_s(\texttt{break}); X'}\ \text{(While)}$$

where $X' = X_c[\texttt{continue} \mapsto \bot, \texttt{break} \mapsto \bot]$. First, note the annotations on the condition which affect the analysis of both the loop body and the lock state after the loop ends. Second, note the circular dependency between the lock state in effect after the body is executed and the lock state at the beginning of the condition (it is possible to unroll the loop and check the condition twice, to get an algorithmic version – only one unrolling is necessary, due to the pessimistic approximation of lock states). Third, the evaluation of the body s may have been prematurely aborted through a **continue** or **break**, and execution continued at the relevant places. In the exception state after the loop we want to reset the registered lock states for these pseudo-exceptions to what they were before the loop, in case there are nested loops. In effect, the **while**-loop serves as (two nested) **try-catch** for the two pseudo-exceptions.

For thrown exceptions we have the following rule:

$$\frac{LS \vdash e \rightsquigarrow LS_e; X_e}{LS \vdash \texttt{throw}^{LS_e}\ e^{\tau}; \rightsquigarrow \bot; X_e[\tau \mapsto X_e(\tau) \diamond LS_e]}\ \text{(Throw)}$$

Three things to note: first, the lock state after this point is $\bot$, marking an unreachable state ("dead code"); second, the **throw** is annotated with the lock state; third, we register in the outgoing exception map that an enclosing catch block might start in the lock state in effect here, LS_e. Since an exception of the same type could be thrown in several places within the same **try-catch**, we must also take previously registered exception states into account.

The rules for the pseudo-exceptions are simpler versions of this, so we omit them here.

Finally we have the rules for **try-catch-finally** blocks. Here we assume for simplicity that all such statements are unrolled to have either a single catch block, or a finally block. We then look at the two cases separately:

$$\frac{\begin{array}{c} LS \vdash s_t \rightsquigarrow LS_t; X_t \\ X_t(T) \vdash s_c \rightsquigarrow LS_c; X_c \end{array}}{LS \vdash \texttt{try}\ s_t\ \texttt{catch}\ (Tx)\ s_c \rightsquigarrow LS_t \diamond LS_c; X'}\ \text{(TryCatch)}$$

where $X' = X_c \diamond X_t[T \mapsto \bot]$. If we reach the start of the catch-block, the lock state in effect must be that which was registered by a corresponding **`throw`** (or several) inside s_t. All exceptions are still valid after the whole try-catch completes, except the one caught, as mirrored by the outgoing exception map.

The rule for **`try-finally`** is then as follows:

$$\frac{\begin{array}{c} LS \vdash s_t \rightsquigarrow LS_t; X_t \\ LS' \vdash s_f \rightsquigarrow LS_f; X_f \end{array}}{LS \vdash \texttt{try}\ s_t\ \texttt{finally}\ s_f \rightsquigarrow LS_t \diamond LS_c; X_t \diamond X_f} \text{(TryFinally)}$$

where $LS' = LS_t \diamond \{X(T) \mid T \text{ is a reference type}\}$. The **`finally`**-block will be executed regardless of what exceptions that may have been thrown inside the **`try`**-block, so we must assume that any of them may be the cause of the lock state in effect.

Policy constraint generation This phase is captured by the following judgment, where we assume that the expression e has been properly annotated from the lock state approximation phase:

$$EX; pcB; PCE \vdash e : p \rightsquigarrow PCE', \theta$$

where

- p is the effective policy of the expression.
- EX is an environment containing registered policies of enclosing exception handlers – catch blocks, or exceptions declared to be thrown in the enclosing method signature. We omit it for the rules that do not use it.
- pcB is a policy serving as a so called *program counter*, whose purpose is to constrain side-effects to prevent indirect flows. Such constraints are induced by being at a point in the control flow graph that was reached due to branches and/or (the presence or absence of) exceptions. pcB deals with branches, and since branches are lexically scoped, so pcB need only appear on the left-hand side.
- PCE deals with the indirect flows due to exceptions. Since separate exceptions have separate areas of influence, we need to know the influence of each exception separately in order to be able to turn them off appropriately. PCE is thus a map from exception types to policies, each of which serves as a program counter. Unlike branches, the influence of exceptions on the other hand follows the execution path, so PCE is propagated as a state that is successively updated.
- θ is the set of constraints that must hold for this expression to adhere to the stated information flow policies.

The judgment for statements is similar, only statements have no effective policy.

We implicitly assume an environment E containing policy signatures for fields, variables, methods and locks, as well as the declared return policy of the enclosing method. We also assume a set of global lock properties G, which would formally be added as a subscript to every constraint, e.g. $p \sqsubseteq_{G,LS} q$, however we omit it since it appears the same everywhere.

The rules for literals, **`this`** and **`null`** are uninteresting. The rule for field dereferencing is as follows:

$$\frac{pcB; PCE \vdash e : p_e \rightsquigarrow PCE', \theta_e \quad E(\tau, f) = p_f}{pcB; PCE \vdash e^{\tau}.f : p_e \sqcup p_f \rightsquigarrow PCE', \theta_e} \text{(Field)}$$

The only interesting thing to note is that the policy is the join of the policies of the containing object and the field. The rule for binary operators is very similar:

$$\frac{pcB; PCE_{i-1} \vdash e_i : p_i \rightsquigarrow PCE_i, \theta_i}{pcB; PCE_0 \vdash e_1 \oplus e_2 : p_1 \sqcup p_2 \rightsquigarrow PCE_2, \theta_1 \cup \theta_2} \text{ (Field)}$$

In the rule for conditionals we extend the branch PC to constrain indirect flows in the branches:

$$\frac{\begin{array}{c} pcB; PCE \vdash e_c : p_c \rightsquigarrow PCE_c, \theta_c \\ pcB \sqcup p_c; PCE_c \vdash e_i : p_i \rightsquigarrow PCE_i, \theta_i \end{array}}{pcB; PCE \vdash e_c ? e_1 : e_2 : p_c \sqcup p_1 \sqcup p_2 \rightsquigarrow PCE_1 \sqcup PCE_2, \theta_c \cup \theta_1 \cup \theta_2} \text{ (Cond)}$$

An assignment constitutes an actual information flow, so the rule for assignments is where many of the constraints arise:

$$\frac{\begin{array}{c} pcB; PCE \vdash e_o : p_o \rightsquigarrow PCE_o, \theta_o \\ pcB; PCE_o \vdash e : p_e \rightsquigarrow PCE_e, \theta_e \end{array}}{pcB; PCE \vdash e_o^\tau.f =^{LS} e : p_o \sqcup p_f \rightsquigarrow PCE_e, \theta_o \cup \theta_e \cup \theta} \text{ (FieldAssign)}$$

where $p_f = E(\tau, f)$ and $\theta = \{p_e \sqsubseteq_{LS} p_f, \bigsqcup PCE_e \sqcup pcB \sqsubseteq p_f, p_o \sqsubseteq p_f\}$. The three constraints generated at this rule have the following purposes:

- $p_e \sqsubseteq_{LS} p_f$ checks that the policy of the data flowing into field f is no more restrictive than the policy declared in the signature of f, relative to the lock state to allow for declassification.
- $pcB \sqcup PCE_e \sqsubseteq p_f$ ensure that no indirect flows arise due to enclosing branches (pcB) or by being within the area of influence of some expression(s) (PCE_e), respectively.
- $p_o \sqsubseteq p_f$ ensures that the value of the object expression $_o$ cannot indirectly be revealed, in a so called laundering attack, by changing a field whose policy is less restrictive than that of the object.

Finally, the rule for method calls:

$$\frac{\begin{array}{c} EX; pcB; PCE \vdash e_o : p_o \rightsquigarrow PCE_0, \theta_o \\ EX; pcB; PCE_{i-1} \vdash e_i : p_i \rightsquigarrow PCE_i, \theta_i \end{array}}{EX; pcB; PCE \vdash e_o^\tau.m(e_1, \ldots, e_n)^{LS} : p_o \sqcup p_m \rightsquigarrow PCE', \theta_o \cup \bigcup \theta_i \cup \theta \cup \theta_x} \text{ (Call)}$$

Here we assume $E(\tau, m) = (p_m, p_w, PP, PX)$, where PP is a mapping from parameter positions to the parameter policy signatures of the method, and PX is a mapping from the method's checked exception types (and pseudo-exceptions) to policies. We then have that $\theta = \{p_i \sqsubseteq_{LS} PX(i), pcB \sqcup PCE_n \sqsubseteq p_w, p_o \sqsubseteq p_w\}$, $\theta_x = \bigcup\{\{PX(\tau) \sqsubseteq EX(\tau), EX(\tau) \sqsubseteq PX(\tau)\} \mid \tau$ is an exception type$\}$, and $PCE'(\tau) = PCE_n(\tau) \sqcup PX(\tau)$ for all exception types τ.

The constraints in θ here are similar in spirit to the ones for assignment, only here the write effect of the method, p_w, is used instead of the policy of the updated field, and there are several direct flows from argument values to parameters. The constraints in θ_x check that the policies of the exceptions potentially thrown by the method match those expected by the environment. Since the policies are used both as constraints of effects in the area of influence, and as the effective policy of the exception value being thrown, the inequality must hold in both directions.

The outgoing exception PC map is updated to take into account all exceptions potentially thrown by the method.

For statements, the rules for empty statements and expression statements are uninteresting, and the rule for **`if`** statements mirrors that for conditional expressions.

The rule for **`while`**-loops is as follows:

$$\frac{\begin{array}{c} pcB \sqcup p_c; PCE' \vdash e_c : p_c \rightsquigarrow PCE_c, \theta_c \\ pcB \sqcup p_c; PCE_c \vdash s \rightsquigarrow PCE_s, \theta_s \end{array}}{pcB; PCE \vdash \texttt{while}\ (e_c)\ \texttt{do}\ s \rightsquigarrow PCE'', \theta_c \sqcup \theta_s}\ \text{(While)}$$

where

- $PCE'(\texttt{continue}) = \bot$ and $PCE'(\tau) = PCE(\tau) \sqcup PCE_s(\tau)$ for all other τ,
- $PCE'' = PCE_c[\texttt{continue} \mapsto \bot, \texttt{break} \mapsto \bot]$,

Note in particular that the entire loop, including the condition itself, is constrained by the policy of the condition.

The rules for **`return`** and **`throw`** are near identical, we show only that for return:

$$\frac{EX; pcB; PCE \vdash e : p_e \rightsquigarrow PCE_e, \theta_e}{EX; pcB; PCE \vdash \texttt{return}^{LS}\ e \rightsquigarrow PCE_e[\texttt{return} \mapsto EX(\texttt{return})], \theta_e \sqcup \theta}\ \text{(Return)}$$

where $\theta = \{p_e \sqsubseteq_{LS} EX(\texttt{return})\}$. Here we have a direct flow of the returned value back to the caller of the enclosing method, and thus the lock state is taken into account. The expected policy of the returned value is found in the enclosing exception environment.

The rules for the pseudo-exceptions are even simpler, so we omit them.

For **`try-catch-finally`**, again we assume an unrolling so that each block is either **`try-catch`** or **`try-finally`**.

$$\frac{\begin{array}{c} EX[T \mapsto p_x]; pcB; PCE \vdash s_t \rightsquigarrow PCE_t, \theta_t \\ EX; pcB; PCE_t[T \mapsto PCE(T)] \vdash s_c \rightsquigarrow PCE_c, \theta_c \end{array}}{EX; pcB; PCE \vdash \texttt{try}\ s_t\ \texttt{catch}\ (Tx^{p_x})\ s_c \rightsquigarrow PCE_c, \theta_t \cup \theta_c}\ \text{(TryCatch)}$$

Note first that we assume the parameter of the catch block to be annotated with its calculated policy p_x from a previous phase. This policy then serves as the expected policy of thrown exceptions of type T in the try block. The catch block is only in the area of influence from exceptions of type T if they were already thrown before reaching this **`try-catch`**.

$$\frac{\begin{array}{c} pcB; PCE \vdash s_t \rightsquigarrow PCE_t, \theta_t \\ pcB; PCE \vdash s_f \rightsquigarrow PCE_f, \theta_f \end{array}}{pcB; PCE \vdash \texttt{try}\ s_t\ \texttt{finally}\ s_f \rightsquigarrow PCE', \theta_t \cup \theta_f}\ \text{(TryCatch)}$$

where $PCE'(\tau) = PCE_t(\tau) \sqcup PCE_f(\tau)$. Here, the interesting thing to note is that we analyse the **`finally`**-block under the same exception PC map, PCE, as the **`try`**-block. The reason is that the **`finally`**-block will always be executed no matter what exceptions are thrown in the **try**-block, so it will not be in the area of influence of any of those. Any code following this **try-finally** statement will still be within the area of influence of any as-of-yet uncaught exceptions from within the **try**-block, which is mirrored by the outgoing exception PC map, PCE'.

Finally we look at the Paragon-specific statements **open** and **close**. Locks too have policies, and opening or closing a lock is an effect visible at the level of that policy, hence we must take implicit flows into consideration:

$$\frac{}{pcB; PCE \vdash \texttt{open}\ L(a_1, \ldots, a_n) \rightsquigarrow PCE, \{\bigsqcup PCE \sqcup pcB \sqsubseteq E(L)\}} \text{ (Open)}$$

The rule for closing is identical.

Appendix References

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.

BS10. N. Broberg and D. 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.

CGT89. 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.

FSVY91. Thom Fruhwirth, Ehud Shapiro, Moshe Y Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Logic in Computer Science, 1991. LICS'91., Proceedings of Sixth Annual IEEE Symposium on*, pages 300–309. IEEE, 1991.

MO84. Alan Mycroft and Richard A O'Keefe. A polymorphic type system for prolog. *Artificial intelligence*, 23(3):295–307, 1984.

Sag88. Yehoshua Sagiv. Optimizing Datalog Programs. In *Foundations of Deductive Databases and Logic Programming*, pages 659–698. Morgan Kaufmann, 1988.

vDBS12. B. van Delft, N. Broberg, and D. Sands. A datalog semantics for paralocks. In Audun Jøsang, Pierangela Samarati, and Marinella Petrocchi, editors, *Proceedings of the 8th International Workshop on Security and Trust Management (STM)*, pages 305–320. Springer, 2012.

ZPSS09. David Zook, Emir Pasalic, and Beata Sarna-Starosta. Typed Datalog. In *Practical Aspects of Declarative Languages*, pages 168–182. Springer, 2009.