Paragon for Practical Flow-Oriented Programming

Niklas Broberg David Sands Bart van Delft

Department of Computer Science and Engineering,
Chalmers University of Technology, Sweden
niklas.broberg@chalmers.se dave@chalmers.se vandeba@chalmers.se

Abstract

Broberg and Sands (POPL'10) introduced a logic-based policy language, *Paralocks*, suitable for static information-flow control in programs. Paralocks has a precise information-flow semantics, and is able to represent a wide variety of information flow idioms, including stateful policies, with just a few basic building blocks. The question of whether these desirable features can be added to a *real* programming language was left open. In this paper we present *Paragon*, a Java-based language with Paralocks policy labels. We show that the Paralocks idea can be scaled to a programming language with features such as dynamic allocation, exceptions, and object orientation. Moreover, the combination of Paralocks with the modularity mechanisms of Java achieve unforeseen benefits: the building-blocks for representing various policy idioms can be fully encapsulated and presented to the Paragon programmer as a library of abstract policies and policy-related functions. We describe the Paragon compiler, and show how a number of previously studied idioms including the *decentralized label model* and Jif primitives, can be coded as a library, and, moreover, how informal coding patterns can be expressed in Paragon and thus properly enforced by the compiler.

1. Introduction

For application security, the semantic concept of *information flow* is fundamental. Understanding and controlling how information flows between the end points of an application is central to the core security concerns of *confidentiality* (the property that sensitive information does not leak to inappropriate sinks) and *integrity* (the property that inappropriate information sources do not influence trusted data). The ability to specify and verify how information should flow through a program is thus a major step towards building secure systems.

Research on language-based information flow security in the last decade has focused heavily on the development and study of a plethora of information flow policies. Much of this work tackles the restrictiveness of classic fixed-lattice models of information flow [1] by studying controlled deviations from such policies, a.k.a. *declassification*. See [2] for an overview and motivation for the study of more complex policies, and [3] for a fairly recent structured overview of the various flavours of declassification found in the literature.

How has this research been reflected in the development of actual security-typed programming languages? The answer is not so much. In fact, there is still only one significant system, the Jif compiler from Myers et al. [4, 5], which provides a direct means to specify information-flow policies including aspects of dynamic policy change and declassification. Jif has served well as a vehicle to explore a particular decentralized information flow model [6, 7], as well as the kinds of policies required in particular applications e.g. [8–10].

Despite its unique position, Jif has not achieved widespread use. Some shortcomings of the Jif approach which may account for this are:

- it is tied to a specific information flow model (the *decentralized label model*, DLM) which is perhaps not the model appropriate for a given application,
- the policy language does not provide abstractions suitable for building new features and policy paradigms – for this the compiler itself has to be extended,[1]
- as a consequence of the previous point, Jif suffers from feature-creep: as the need for new policy features is discovered, successive releases of the language accumulate more and more features, making Jif increasingly complex to work with.

Paragon In this paper we describe a new security-typed language, Paragon, which builds on the recent policy language *Paralocks* proposed by Broberg and Sands [12], as well as experience from the Jif system. Paralocks (described in Section 2) offers to be a relatively simple core language with information flow as a primitive notion, in which a wide variety of stateful and dynamic information flow policies may be described without adding new primitives to the policy language. But the previous work on paralocks provides only a proof-of-concept static type system for a toy language, rather far from a realistic programming language. Questions and challenges which must be addressed in the adaptation of these ideas to a real language include:

- whether the model can be scaled to handle the features of a real programming language, such as objects, exceptions, dynamic allocation, aliasing, and so forth,
- since the encoding of complex policies requires *computation* of policies (as in the DLM example from [12]), how can static type-checking be used?
- Paralocks provides a "core calculus" for building information flow policies; what abstraction facilities are needed to make programming with paralocks palatable?

Contributions and Organisation The contribution of this paper is the presentation of a new language, Paragon, which meets these

[1] The underlying compiler technology of Jif, Polyglot [11], is built with extensibility in mind, but this is still a major undertaking.

challenges. Paragon is a security-typed extension to Java[2] supporting Paralocks policies. Section 2 recalls the Paralocks information-flow policy language upon which Paragon is based. Paragon itself is introduced (Section 3) through a series of examples. The examples illustrate how the features of Paralocks combine fruitfully with the basic abstraction mechanisms of Java: the internal building-blocks needed to represent various policy idioms can be encapsulated by standard Java features, so that a user of a given policy idiom is not exposed to the Paralocks representation. Section 4 dives into the details arising in the tracking of information flows in Paragon, while Section 5 briefly describes the Paragon compiler and runtime system. Since Jif holds a special place among the related works, Section 6 provides a more in-depth comparison between Paragon and Jif, pointing out both how our approach addresses some shortcomings of the Jif approach, and where Jif has directly influenced our design. We also show how Paragon can encode the DLM as a library which exports the basic Jif language primitives – policy constructors, declassification functions, and operations to declare code authority and the delegation relation between principals – with almost no extra notational overhead. We conclude with a broader overview of related work (section 7) and future work (section 8).

2. Paralocks

In this section we introduce the Paralocks policy language from [13] that forms the basis for the Paragon language.

The Paralocks policy language itself is largely "policy neutral" – it does not presuppose any particular labels, information flow levels, or any specific fixed hierarchy. Instead it builds on two basic components: *actors* and *parameterized locks*.

Actors The first basic building block of policies is an unstructured set of *actors*. These are the conceptual recipients of information. Actors model, for example, specific communication channels, principals, or security clearance levels.

A Paralocks policy describes how and when some actor may gain information about any piece of data labelled with that policy. We use the phrase "gain information" rather than "access" to stress that this is an information flow notion rather than an access control one. Since confidentiality and integrity are not built-in notions in Paralocks policies, an actor is a more primitive concept than a principal. [3]

Locks The second building block accounts for the stateful and dynamic component of policies, a form of boolean variables called *locks*. Locks are the interface between the code and the policy in the sense that they are used to communicate the security relevant state of the program to the policy.

In general, locks can be parameterized by zero or more actors (hence the name *para*locks). Each lock has a fixed arity. Typically, nullary locks (these are the *flow locks* from [13]) model general policy relevant events, for example the declassification of some data item, or a global state change which implies a change in flow policy; unary locks can be used to model *roles* e.g. Manager(Bob), and thus support a role-based version of information flow control with dynamic role assignment; binary locks are used to model relations between actors, such as a delegation e.g. Bob *actsFor* Alice.

Policies Policies are built from actors, locks and actor-variables by forming simple statements in first-order logic, namely *Horn clauses*. A policy is a collection (a logical conjunction) of clauses of the form

$$\forall a, \vec{a}_1, \ldots, \vec{a}_n.\, L_1(\vec{a}_1) \wedge \cdots \wedge L_m(\vec{a}_m) \Rightarrow a$$

This clause describes the conditions $L_1(\vec{a}_1) \wedge \cdots \wedge L_m(\vec{a}_m)$ under which information may flow to the actor a. Here the head of the clause a can be viewed as a shorthand for a distinguished implicit predicate MayFlowTo(a).

In the Paragon language and in the sequel we adopt a more Prolog-like notation and write e.g.

$$\forall x.\, \text{HighClearance}(x) \wedge \text{Safe} \Rightarrow x$$

as `'x : HighClearance('x), Safe`. As in this example, the `'`-prefix on any variable simply denotes that the variable is locally quantified (i.e. it is a bound variable in this policy).

Example Consider policies for a simple conference-management system. The actors of interest are members of the roles Author, PCMember, and PCChair. The key global states of the system are modelled by nullary locks, SubmissionPhase and ReviewPhase, with the obvious meanings. In the scope of some actor A, a paper submission written by A can be stored in a location with a two-clause policy:

```
authorData= {A: Author(A) ;
             'x: PC('x),ReviewPhase}
```

Its contents can always flow to the author, and can flow to a PC member during the review phase. We may also wish to ensure the integrity of the paper once the review phase has begun. In other words the author should not be able to change the paper after submission. Although paralocks does not have "flows-from" policies, this integrity constraint can be enforced e.g. by ensuring that the data provided by the author is given policy

```
authorChannel = {A: Author(A);
 'x: PC('x),ReviewPhase,SubmissionPhase}
```

This means that such information can only flow to a PC member from the author during the submission phase. At any other time the respective policies for the author's channel and the storage object for the author's paper will be incompatible. (See the discussion of semantics and verification below.)

Finally we mention one further feature of paralocks which means that they are (possibly recursive) datalog programs: the ability to have rules which describe invariants that relate locks. In this example, the PCChair role can implicitly inherit from the PCMember role by the global lock invariant: `{ PC('x): PCChair('x)}` This means that the PCChair is also implicitly a PC member.

The Policy Lattice Interpreting policies as statements in first-order logic provides the a natural lattice structure on policies, where the policy ordering ($\sqsubseteq$) on individual clauses is just logical entailment. Specifically, we define $p \sqsubseteq q$ whenever p, viewed as a first order formula, entails q. Basically, the smaller the policy, the more flows it permits. The *bottom* element of the policy lattice is `{'x: }` (information may flow to any actor x, unconditionally), and the *top* element is $\{\}$ (information can never flow to any actor). In general the comparison of policies takes into account the locks which are known to be open. In the example above we have `authorChannel` $\not\sqsubseteq$ `authorData`, but

`SubmissionPhase` $\vdash$ `authorChannel` $\sqsubseteq$ `authorData`

where $\vdash$ is just standard logical entailment.

Validating secure information flows In [12, 15] a variant of non-interference is developed which provides the semantic definition of

[2] A substantial sequential subset of Java 7.x

[3] For example, a single principal in the D-Star policy language [14], and in more recent versions of Jif [5], is the subject of both confidentiality and integrity polices, and thus might well be modelled in a paralocks policy by a *pair* actors, one to express the information flow constraints concerning confidentiality, and another to express those dealing with integrity.

secure information flow in a system with data labelled with Paralocks policies. This defines the goal of the language-based enforcement mechanisms. Roughly speaking it ensures that what an observer might learn at each computation step is consistent with the policies of the objects currently being modified with respect to the current lock state. Here we provide a rudimentary intuition by considering when an assignment is sound and how to verify it. At any point during program execution, the permitted flows will depend on the locks which are open at that point. Thus a type system must safely approximate the locks which are guaranteed to be open at each program point. For example, suppose we have an assignment in straight-line code $x := v$. Suppose that at least locks L will be open (true) at that program point. For this assignment to be safe, the policy p of the variable x must be at least as restrictive as the policy q of the data v relative to the locks L. This is checked by verifying the computable condition: $L \wedge p$ entails q. In addition, if there are any global lock invariants these are added to the left-hand side. This problem, in the general case, amounts to the decidable question of the containment of a non-recursive datalog program in a recursive one[4].

3. Paragon by example

Paragon is a dialect of Java which adds the ability to label data with Paralocks policies. The compiler will ensure that information only flows according to the defined policies. In this section we introduce Paragon through a series of encodings of various information flow policy mechanisms. We present them by implementing each mechanism as a library, which serves a two-fold purpose. First, these examples allow us to introduce the features of Paragon step by step. and put both the basics and the more intricate parts into context. Second, it lets us demonstrate the generality of Paragon as an implementation language for a large variety of different policy mechanisms, and how, by the use of encapsulation, we can present each mechanism through a consistent interface.

3.1 Simple declassification

Our first example is just a classic two-level confidentiality lattice with a simple declassification mechanism, showing succinctly how class encapsulation gives us the possibility to encode a policy scheme as a library.

The interface of this scheme consists of three things: policies for data that is secret ("high") and public ("low") respectively (not to be confused with Java's notion of "public", i.e. exported from a class), and a method `declassify` that takes secret data as input and releases it as public.

First we define the policy `low` as the least restrictive policy, for data that anyone can see:

```
public static final policy low = { 'x: };
```

Policies in Paragon are first class values of a primitive type **`policy`**. This way we allow for policies that are not known until runtime (c.f. *runtime labels* [17]), further discussed in section 4.8. For a policy to be used to annotate a variable, we require that policy to be marked **`final`**, i.e. immutable. This ensures that the policies remain consistent throughout the program.

High data may be made visible to low observers through declassification. We represent this with a condition (lock) `Declassify`:

```
private lock Declassify;
```

Unlike policies, locks are not first class values in Paragon, and cannot for instance be stored in variables. Locks are always implicitly **`static`**, to avoid aliasing problems. We discuss aliasing issues in more detail in section 4.4.

The policy `high` is now simply that which specifies that the data may be made visible to a low observer when the lock is open:

```
public static final policy
    high = { 'x: Declassify };
```

The act of declassification then becomes a simple matter of taking data with policy `high` and, in a context where `Declassify` is open, *re-annotating* it with policy `low`. Such re-annotations typically happen at assignments, but can also happen at e.g. the return of a method. This is exactly what the method `declassify` does:

```
public static ?low <A>
  A declassify(?high A x){
    open Declassify { return x; } }
```

There are several interesting things to note about this method declaration. First, it shows how to use policy annotations in Paragon: We simply extend the list of possible *modifiers* on e.g. variables and methods. Here we see that the formal parameter `x` has a modifier **`?`**`high`, stating that an argument to the method should have a policy no more restrictive than this. The method itself has a modifier **`?`**`low`, denoting the *return policy*, i.e. the effective policy on data returned by the method.

Another thing to note is that neither of the policies we declared, nor the lock, were annotated with policies. They still have policies though: for fields the default policy is bottom if nothing else is specified. For locks, it is top.

Also, the body of the method consists of a single statement: a *scoped* **`open`** statement. The scoped **`open`** keeps the specified lock open for the extent of its body. In other words, it opens the specified lock at the start (if it was not already open), closes it when done (unless it was already open at the start), and rules out any (non-scoped) opens or closes of that lock throughout the body.

Finally, as suggested above, returning from a method causes a re-annotation of the returned data to the declared return policy of the method. Here the re-annotation is valid since it appears in a context where `Declassify` is open.

We also note that this method is now the *only* way to declassify data from `high` to `low`, since the `Declassify` lock is declared to be private to this class. Our library can thus have a simple, consistent interface through the use of standard encapsulation techniques.

The library exposes only the basic building blocks to the application programmer: the primitive policies `low` and `high` which can be used to label data, and a declassification method. The compiler statically checks that information in user code is according to the policy, and encapsulation of the `Declassify` lock ensures that the declassification method is the only way that high data can be relabelled as low.

3.2 Robust declassification

To achieve *robust declassification* [18], we need to introduce data integrity policies. Integrity is the dual of confidentiality, and we can handle the two concepts in just the same way. Robust declassification then requires that the choice of what data to declassify cannot be affected by *untrusted* data.

To model integrity we use an actor representing a user who *only* observes, and acts on, trusted data.

```
private static final actor trustor;
```

Actors, like policies, are values of a primitive data type, **`actor`**. Declaration of an actor implicitly creates an opaque identity for that newly created actor. Values of type **`actor`** are first class. Just

[4] A full treatment of the relationship between Paralocks and Datalog, and the algorithms for the policy lattice operations are explored in [16].

like for policies, for an actor to be mentioned in annotations on data that actor must be declared **final**, to ensure consistency. We declare the policy of trusted data as being fit to flow to the actor `trustor`:

```
public static final policy
    trusted = { trustor: };
```

Thus the role of the trustor is to be a witness to the fact that a given piece of data has only been computed from trusted sources.

Now we wish to combine the notion of trusted data with the simple declassification mechanism above, but then we run into a problem. Since we modelled `low` as bottom, all data marked with `low` would be implicitly trusted – i.e. observable by `trustor` – already! Our formulations of the policies in our previous library were too simple to be combined with the notion of integrity as required for robust declassification. We need to define new versions of `low` and `high` based on an explicit observer separate from `trustor`:

```
private static final actor observer;
public static final policy
  low  = { observer: },
  high = { observer: Declassify };
```

Now we can modularly form the combinations we need, e.g. trusted *and* low data would have the policy `trusted` ⊓ `low`.

In general the state of a lock may be observable, so locks themselves have a policy. When a policy for a lock is not explicitly given, as in the examples so far, the policy defaults to bottom. But to ensure robustness, i.e. that the choice whether to declassify is not based on untrusted data, we give the lock that governs declassification the policy `trusted`.

```
private ?trusted lock Declassify;
```

This means that the value of the lock may only depend on trusted sources, a property that the compiler will check for us. The type of `declassify`, which manipulates the lock, will reflect this in a *write effect* modifier, `!trusted`:

```
!trusted ?(low ⊓ policyof(x))
public static <A>
  A declassify(?high A x) {
    open Declassify { return x; } }
```

Indeed the same holds for other data marked as trusted – it cannot be affected by untrusted data, neither explicitly nor implicitly.

Notice how the method is polymorphic in whether its argument is trusted or not. The parameter is marked with policy `high`, which really means "no more restrictive than `high`", so arguments to the methods could be trusted, untrusted, or even low if we wanted. The return policy states that the result will have the same policy as the input, only it will now (definitely) be low (`low` ⊓ `high` is `low`).

This library could easily be extended with a mechanism for *endorsement*, similar to declassification, but using a different lock e.g. `Endorse`. We note that this way of writing declassifying functions that work over data with different policies forms a general pattern that implements "trusted declassifiers" as proposed by Hicks *et al* [19].

3.3 Sealed-bid auctions

Our next example is not a library but an actual application, which has been used as an example in several earlier papers [12, 13, 20]: a server for running online sealed-bid auctions. In this setting we want to model the following information flow properties:

- bidders provide sealed bids and can see their own bid, but cannot see each others' bids.
- bidders learn of the winning bid, but only at the end of the auction.

We only sketch the implementation of the system here, focusing on the parts that are interesting from a Paragon perspective and which allow us to illustrate further language features, leaving many other things underspecified.

A bidder is represented in the system as an actor. The bid placed by actor `a` should be visible only to `a` while the auction is running, and be released to all other bidders when the auction is complete assuming it was the winning bid. Additionally, the process of determining the winner of the auction can be considered a trusted context, where the result will reveal some information about the bids of all bidders (namely that they were no higher than the winning bid). We model this with the policy

```
{a: ;
'x: AuctionClosed, HasBid('x), Winner(a) ;
'x: DetermineWinner}
```

where we assume the existence of the locks `AuctionClosed` and `DetermineWinner`, and the two unary lock families `HasBid` and `Winner`, with intuitive interpretations.

We wrap a bidder and their associated information and operations as a class `Bidder`, starting with the following:

```
final actor id;
final policy bidpol = {id: ;
  'x:AuctionClosed,HasBid('x),Winner(id) ;
  'x:DetermineWinner};
?bidpol int bid;
```

We implicitly also assume a channel, `chan`, for communication with the actual bidder. Data received on this channel has the policy `bidpol` as well, but the presence of data, i.e. whether a person placed a bid, is public and has policy `{'x:}`.

We note that the actors here, unlike those used in the previous examples, are not marked as **static**. This means that each instance of `Bidder` will have a separate actor, uniquely generated when that instance is created.

When the bidder supplies a bid as requested, we signal this by opening the corresponding `HasBid` lock. If the bidder fails to supply a bid, we throw an exception:

```
+HasBid(id) !{'x:}
void getBid() throws !{'x:} NoBidExc {
  bid = chan.get(); open HasBid(id); }
```

Two things are worth noting here. The first is the `+HasBid(id)` modifier, which signals to the type checker that calling this method will open that lock, assuming the method call terminates normally. If it instead terminates with an exception, we make no such guarantees. The second thing to note is the write effect modifier on the declared exception. Roughly speaking, this policy denotes the level at which it will be possible to observe that the function has terminated with this exception. Java does not normally allow modifiers on declared exceptions – they are an addition in Paragon.

Running the auction now consists of four phases: Getting the bids from all the bidders, determining the winner, reporting the results, and handing out the spoils. The first phase simply loops over all bidders, gets the bid of each, catching exceptions along the way:

```
!bottom void collectBids() {
  for (Bidder b in bidders) {
    try { b.getBid(); }
```

```
      catch (NoBidExc e) {} }}
```

The only thing to note here is that the contents of the set `bidders` must be observable by all the bidders, due to the write effect of `getBid`. The same is true for the overall write effect of this method – every bidder can observe that the method has been called, so the only sensible write effect policy is bottom.

In the next phase we look at all the collected bids, determine the winner among them, and declare the auction closed. We first declare a policy `allBidders` as the part of the policy on bids that is not specific to a particular bidder:

```
final policy allBidders =
  {'x: AuctionClosed, HasBid('x) ;
   'x: DetermineWinner};

+AuctionClosed ?allBidders
   Bidder determineWinner() {
     Bidder winner;
     open DetermineWinner {
       for (Bidder b in bidders) {
         if (HasBid(b.id)) {
           if (winner == null
             || b.bid > winner.bid) {
             winner = b; }}}
   }
   open AuctionClosed;
   return winner; }
```

The local variable `winner` must have policy `allBidders` for the above code to be type correct. We don't need to explicitly annotate it with that policy though – Paragon performs inference of policies for local variables.

Also noteworthy is that the assignment to `winner` does not affect the write effect of this method, since `winner` is only available locally within the body of the method, so changes to it will not be visible from outside a call to the method.

The method is guaranteed to open the `AuctionClosed` lock, as signalled by the appropriate modifier.

Next we want to notify the bidders about the winning bid:

```
~AuctionClosed !allBidders void
  reportResult(?allBidders int winBid){
    for (Bidder b in bidders) {
      if (HasBid(b.id)) {
        b.chan.put(winBid); }}}
```

We assume that the channel to `b` makes the data sent on it available to `b`, i.e. it can only output data with policy (no more restrictive than) `{b.id:}`. To be allowed to send `winBid`, with policy `allBidders`, on this channel, we must know that we are in a context where the two locks mentioned in that policy are truly open. The modifier `~AuctionClosed` declares that this method *expects* that lock to be open whenever it is called. Calling it in a context where that lock is not guaranteed to be open is a type error, and consequently the body of the method may assume that the lock is indeed open. For the second lock, we rely on so called *runtime querying* for the status, through an **`if`** statement. If the condition of the **`if`** is a lock, the type checker can assume that that lock is open when checking the *then*-branch. Thus the re-annotation of `winBid` is correctly allowed.

Tying all these pieces together we could now write the main code as follows:

```
collectBids();
Bidder winner = determineWinner();
if (winner != null) {
  open Winner(winner.id);
  reportResult(winner.bid);
  sendSpoils(winner); }
```

where we leave to imagination how `sendSpoils` should be annotated and implemented, but surmise that it requires the appropriate `Winner` lock to be open. We note that the re-annotation of `winner.bid` is allowed when using it as the argument to `reportResult`, since we know that `Winner(winner.id)` is guaranteed to be open.

In this section we introduced several new concepts, each of which is presented in more detail in a later section: Lockstate modifiers (section 4.1), runtime querying of locks (section 4.2), exceptions (section 4.6), policy inference (section 4.9), and instance actors and aliasing (section 4.4).

4. The Paragon language

The examples of the previous section have given a flavour of the language and its features, aimed for the casual reader or as a first introduction. In this section and the next we offer a more detailed account of the design and implementation of Paragon.

The remainder of this section is structured as a reference to the various features, each presented in a separate subsection, with no overall narrative. We will cover the following sections in turn: Types, policies and modifiers (4.1); Locks (4.2); Type parameters (4.3); Actors and aliasing (4.4); Type methods (4.5); Exceptions and indirect control flow (4.6); Field initializers (4.7); Runtime policies (4.8); Type and policy inference (4.9); The Paragon type system (4.10).

4.1 Types, policies and modifiers

In Paragon every information container (field, variable, 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 are read by it – we refer to this as the expression's *read effect*. 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.

Paragon (unlike Jif – see section 6.3) separates policies from base types syntactically by having all policy annotations as modifiers. All in all, Paragon adds ten new modifiers over Java. Two of them relate to policies:

- `?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.
- `!pol` denotes a write effect, and is used to annotate methods. They are also used to signal the write effects of thrown exceptions (see section 4.6) and of static initializers (section 4.7).

There are also three modifiers used only on methods and constructors to detail their interaction with the lockstate:

- `+locks` says that the 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 specified lock(s) must be open whenever the method is called.

The *opens* and *closes* modifiers are also used for exceptions, discussed in section 4.6.

The other five modifiers introduced by Paragon are the three short-hand modifiers for lock properties discussed in section 4.2,

the `readonly` modifier discussed in the same section, and the **`typemethod`** modifier discussed in section 4.5.

4.2 Locks

Locks in Paragon are not first class. They cannot be stored in variables, nor can they be passed as arguments to methods. The only way to manipulate the status of a lock is via **`open`** and **`close`** statements. However, the status of a lock may be queried at runtime. If a lock (or a conjunction of locks) is used syntactically as an expression, the type of that expression is considered by Paragon to be **`lock`**. If an expression of type **`lock`** appears as the condition of an **`if`**, **`while`** or **`do`** loop, or as the first operand of the ternary conditional operator `?:`, the type checker can assume that the lock is open when checking the branch corresponding to the condition being satisfied. Apart from this effect on the typing of programs, all expressions of type **`lock`**, in conditions or elsewhere, are implicitly cast to **`boolean`**.

Lock properties Lock families can be declared to have *properties*. A property specifies conditions under which some locks in the family are *implicitly* open. A concrete lock can thus be explicitly closed, but still remain open due to some property, such as transitivity, keeping it open implicitly.

For example, delegation between actors might be encoded by an `ActsFor` relation. This relation should be transitive and reflexive. This requirement can be stated as a lock property at the point of declaration[5]:

```
public lock ActsFor(actsForA,A) {
  ActsFor('x,'x);
  ActsFor('x,'y): ActsFor('x,'z),
    ActsFor('z,'y);}
```

Transitivity and reflexivity properties are a common pattern, so Paragon provides syntactic sugar for these:

```
public reflexive transitive lock
  ActsFor(actsForA,A);
```

Since lock properties must be attached to the declaration of a lock, they are modest restriction of the *Recursive Paralocks*, discussed in [12]. However it turns out that this restriction make the policy operations decidable, and the policy comparison operation specified in [12] become complete (as opposed to just being sound). Full details are given in [16].

A lock may be exported as `readonly`. This means that it may be used in the declaration of policies, in lockstate modifiers on methods and constructors, and in lock queries, but not opened or closed outside the scope of the defining class.

4.3 Type parameters

Java, since the introduction of "Generics" in Java 5.0, allows types and methods to be parametrized 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. The different entities are clearly not interchangeable, which implies the need for a simple *kind* system for type-level entities.

Thus in Paragon ordinary 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. For actors and policies we can simply reuse their types as kinds as well. We can do the same for locks though we need to be able to parametrize 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`**`[]`.

[5] The formal parameters of the lock declaration are only used to specify the arity of the lock, and to give a pnumonic hint to their meaning.

4.4 Actors and aliasing

Actors and locks together play a crucial role in the typing of Paragon code. Locks determine what flows are allowed at what points, and locks are often parametrized by actors. The typeability of some code may depend on a given lock, with some given actor arguments, being open. Formally, the type checker treats actors as *singleton types* [21].

However, the possibility of aliases greatly complicates things. If some code opens lock `L(a)` and then closes lock `L(b)`, is the first lock still open? Clearly that depends on whether or not `a` and `b` are two different actors.

Alias analysis in Java is a well-studied area, with many possible degrees of sophistication. For Paragon, erring on the side of caution is clearly crucial, so any analysis that *conservatively* approximates actor aliasing is adequate. The current implementation is relatively simple, and shortcomings in the alias analysis can be compensated for by adding runtime queries to locks. Paragon is not dependent on the details of the aliasing analysis so replacing the alias analysis with a more accurate one is always possible.

4.5 Type methods

A type method is Paragon's name for simple methods that can be evaluated by the type checker at compile time, in order to determine policies on variables, fields and methods. A more formally correct name would perhaps be type *functions*, since these methods must be both pure, i.e. have no side-effects, and deterministic. By deterministic we mean that the end result may only depend on values known statically when the method is called. That includes the method's arguments, as well as certain static fields. For a field to be useable in a type method, it must be static and final, have a primitive type, have a policy bottom, and have a simple initialiser that is itself pure and deterministic.

The fact that type methods can be used at compile time does not preclude them for being used at runtime as well, where they behave like static methods.

4.6 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.

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.

A caught exception is in essence a jump, where control is passed from the throw point to a catch block. Such a jump may be noticeable by anyone who can notice either the catch block being executed, or the statements in the normal control flow past the throw point. To avoid unintended flows, all such statements must be constrained by the context in which the throw appears. We refer to this as the exception's *area of influence*.

Since an exception might not be caught locally, the area of influence is not a local property in general. To handle this modularly we let methods that throw exceptions declare the write effects of those

expressions as modifiers on the exception types in the method's `throws` clause. This declared write effect serves as an approximation of the context where the `throw` appears. It is thus both used as the effective write effect of the `throw` statement itself, to ensure that it is not used in even more restrictive contexts, as well as a bound on the write effects of all statements in the area of influence.

Since uncaught exceptions are effectively premature exit points from a method, the *opens* and *closes* modifiers pertaining to a normal exit do not apply when entering a catch block. Hence we let the declared exceptions also take opens and closes modifiers, specifying the lock state that will be in effect at the start of a corresponding catch block.

For the cases where a thrown exception is caught locally, before ever reaching the top level of a function, there will be no need for approximations via declared policy or lockstate modifiers. Instead all the necessary information can be computed locally.

Interestingly, several other control flow mechanisms in Java can be treated as special cases of exceptions for purposes of policy inference: `return`, `break` and `continue`. These are simpler to handle than exceptions, since their area of influence is always contained locally.

4.7 Field initializers

Initializers for fields are simply expressions, and quite naturally the effective policy on such an expression cannot be more restrictive than that on the field. But expressions also have a write effect – i.e. the initialization taking place might cause visible changes elsewhere. Furthermore, an initialization could potentially fail with an exception.

Fields come in two different flavors: static fields and instance fields. For both, the main difficulty lies in handling their initialization – their side-effects and possibility for exceptions – and the solution differs between the two.

Instance fields are all initialized when the instance they belong to is created. This means that we can view the initialization code as being an implicit prelude to every constructor for the class. The solution is then natural – the write effect of the initializers cannot be less restrictive (more revealing) than the declared write policy on any constructor. Similarly, if an initializer could throw an exception, all constructors for the class must declare this.

Static fields have the same issues with write effects and exceptions, but the story is far less simple. In Java, all static fields of a class are initialized at the same time, whenever any one of them is used in the program. This in effect means that *any* use of a static variable will have the worst-case write effect of all the static initializers for the same class. We can analyze whether a static initialization is guaranteed to already have taken place before a given use of a static variable, to preclude it from carrying the write effect.

Regardless of whether a particular use of a static variable should be assigned a write effect or not, we need to notify the type checker of such write effects. We argue that the most natural place to do this is a modifier on the *class*, specifying in one place the upper bound on the write effects of *all* the initializers of static fields of that class.

Exceptions in static initializers are even trickier to handle. For now we require that initializers for static fields may not fail, but leave open the possibility for a more sophisticated solution in future work.

4.8 Runtime policies

Since policies can be used as values at runtime, and dynamically hoisted to the type level, we need ways to relate policies that are not known statically to other (static or dynamic) policies. To achieve this, Paragon needs to perform runtime entailment checks between policies. This problem has been studied by Zheng and Myers in the context of Jif [22], and we choose to follow their solution.

Similar to runtime lock queries, we thus allow inequality constraints between policies to appear as the condition in `if` statements and conditional `?:` expressions. The type checker can then know when checking the first branch that the inequality holds, and can allow flows that would otherwise have been untypeable.

4.9 Policy inference and defaults

To reduce the burden on the programmer to put in policy annotations, Paragon attempts to either infer, or supply clever defaults for, policies on variables, fields and functions. Paragon's policy defaulting mechanisms turn out to be essentially the same as those in Jif. We omit the details here.

Policy inference works through a straightforward constraint system, where all constraints arising from comparisons between policies, including the program counter (PC), are collected and resolved on a per-method basis. In the general form a constraint will be an inequality between two policy expressions, each of which can contain literal policies, variables denoting policies, and joins and meets.

4.10 The Paragon type system

Paragon's implementation follows a specification given by formal type rules for the core of Java but excluding inheritance. The type system is quite involved, so for space reasons we do not present it here. Instead we refer to [23] for the full details.

5. Compiling Paragon

In the previous sections we have presented the front-end of the language Paragon: its features and expected behavior, as well as the static semantics. In this section we briefly discuss how we compile a Paragon program into vanilla Java, and how the runtime aspects of Paragon are represented.

Once we know that a given program satisfies the intended information flow properties, we can safely remove all type-level aspects of policies, locks and actors. We must still retain the runtime aspects, and in some cases demote types to values.

All actor and policy type parameters on methods are demoted to formal (value) parameters, and type arguments to method calls are demoted to normal arguments. For type parameters to classes, each type parameter is also added as a field to the class, and as a formal parameter to each constructor of the class, with an initialization at the start of the constructor.

Actors are pure names; the only runtime property they possess is their unique identity. Many different representations could be considered – our implementation uses instance of class `se.chalmers.paragon.Actor`, which holds only a single integer value, unique for each actor. All declarations of actor fields and variables with no accompanying initialization are given initializers that generate a unique actor representation.

Lock families need to support opening and closing of individual locks in the family, parameterized by actors, as well as querying of current status. Again many different representations could be considered. Our implementation uses a `java.util.Set`, whose entries are arrays of actor representations for the actors for which a lock in the family is open. The three operations are then obvious. As a special case, locks with no parameters are simply boolean variables. We implement the two kinds as sub-classes of a class `se.chalmers.paragon.Lock`, which provides a uniform interface.

The only Paragon-specific statements are `open` and `close`, which become manipulations of the lock representations discussed above.

Our Paragon compiler is written in Haskell and comprises roughly 15k lines of code, including comments. Approximately half of that code is due to our policy type checker, and only a small fraction, just over 500 lines of code, deals with generation of Java code and Paragon interface files. On top of that, some 1500 lines of Java code are written for our runtime representations of Paragon entites[6].

The compiler can be downloaded from our Paragon website [24], or from the central Haskell "hackage" repository using the command `cabal install paragon`.

6. A comparison with Jif

Comparing Paragon to Jif is inevitable. Jif stands out as the only mature and currently maintained information-flow-typed programming language, and is a source of inspiration for the present work. Due to the unique position Jif has enjoyed in the domain of information flow research over the last decade, a fair amount of research has been done using Jif and DLM for context and examples. It is thus natural to ask how research done on or with Jif can carry over to Paragon.

In this section we make a brief but detailed comparison between Paragon and Jif. We begin by giving an overview of Jif and its policy specification language, the *decentralized label model* (DLM). Second we discuss some perceived short-comings of Jif, and how we deal with those aspects in Paragon. We then go on to compare various language features, and point out where Jif, or Jif-related research, has directly influenced our design choices. Finally we show how Paragon can encode the DLM as a library, and how Paragon could be used as a drop-in replacement for (non-distributed) Jif in existing examples.

6.1 Jif and the DLM

In this section we briefly introduce the core concepts of the *Decentralized label model* (DLM) of Myers and Liskov [25], and its realisation in *Jif*, a security-typed programming language that extends Java with DLM policies.

The Decentralized Label Model The decentralized label model is a model for confidentiality and integrity policies for data shared between principals. In the following description we will focus, as in the original DLM papers, on the confidentiality aspects which concern who can "read" data; integrity aspects are handled in a largely dual manner.

A DLM policy is a label that is built from *principals*. It consists of zero or more *owners* – reflecting that data may have come from zero or more sources. Each owner specifies a set of *readers* who are permitted by that owner to see the part of the information that comes from that owner. As an example consider the following label $\{\text{Alice} : \text{Chuck}, \text{Dave} \ ; \ \text{Bob} : \text{Dave}, \text{Eve}\}$. Here Alice and Bob are the owners and Chuck, Dave and Eve are readers. Data with this label might have been obtained by combining data from Alice and Bob in some way. The *effective readers* in this example is just Dave – i.e. unless there is some further delegation by Dave or declassification by either Alice or Bob then Dave is the only one to whom information with this label may flow.

Apart from labels, there is one other important component to the DLM, namely the *principal hierarchy*, specified by a reflexive and transitive *acts-for* relationship. If Eve *acts for* Dave then Eve can do anything that Dave can. In the example above this means Eve can also read data with the label given above.

Jif Jif (and its predecessor JFlow, [4, 5]) is a version of Java which adds statically-checked information-flow annotations in the form of DLM labels. Jif extends the core DLM model with a number of important features, including:

- Authority and Selective Downgrading: any piece of code in a Jif program runs on behalf of a certain set of principals, known as the *authority* of the code. The language contains a *declassify* operator which allows the policy of an expression to be weakened. But not just any weakening is permitted. Only parts of the policy owned by the current authority may be weakened in this way. For example suppose a piece of data is labelled with the policy $\{\text{Alice} : \text{Chuck}, \text{Dave} \ ; \ \text{Bob} : \text{Dave}, \text{Eve}\}$ as above. If the code runs with at least the authority of Alice then it can be declassified to $\{\text{Alice} : \text{Chuck}, \text{Dave}, \text{Eve} \ ; \ \text{Bob} : \text{Dave}, \text{Eve}\}$ in which case the information may then flow to Eve.
- Robustness: Jif (since version 3) can optionally be run in "robust" mode [18, 26]. In robust mode the decision to declassify and the data to be declassified cannot be influenced by low-integrity data.

Many other features of Jif are purely programming language issues rather orthogonal to the DLM and policies, and concern the tracking of information flows and the way the type system expresses these. Examples of such features include the treatment of exceptions, and the support for principals and labels that are only known at runtime.

6.2 Jif concerns

We had two main concerns with Jif when starting the Paragon project. The first was lack of an information-flow semantics for a DLM, in particular in conjunction with declassification and a dynamic principal hierarchy. Our earlier work [12, 15] focusses on providing a precise semantics for Paralocks from an information-flow perspective, although for a much simpler language than Paragon.

The second concern was that the policy model in Jif was too restrictive, in that it could not be used to express easily many of the proposed idioms for programming with information flow control. As case in point, the original DLM model and the early versions of Jif could not express robust declassification. To address this short-coming, Jif has *added* integrity labels on top of the already existing confidentiality labels, as integrity aspects could not be expressed using already existing features. Once added, most code which previously used only confidentiality labels could not be type checked without adding integrity labels[7].

In contrast, flexibility has been one of the main design goals for Paralocks and Paragon. We have shown in section 3 (and in our previous work on Paralocks) that Paragon can encode a number of existing idioms for information flow policies, including robust declassification, which we believe serves as evidence that we have succeeded. In Section 6.4 we show how the DLM model with declassification and a dynamic principal hierarchy can be encoded and encapsulated as a Paragon library.

6.3 Feature comparison

Types and policies vs labeled types In Jif, every value has a *labeled type*, bundling types and labels together both syntactically and semantically. We find this unfortunate since, while both types and labels affect type checking, they are largely orthogonal concepts as far as a programmer is concerned.

[6] As we note in section 8, we expect the size of our supporting libraries to grow significantly as we gain more experience by implementing larger systems in Paragon.

[7] One might hope that Jif's *default* integrity policies would make this unnecessary, but in our experience dummy integrity policies still need to be added manually.

Paragon keeps policies and types separate, keeping the former specified through modifiers instead of annotations directly on the types. We feel this allows for a cleaner separation of the Paragon additions from the vanilla Java code, making Paragon code more accessible to a Java programmer.

One potential critisism of this choice is that it becomes less straightforward to reuse components from existing Java libraries. For example, any collection-like class (e.g. `java.util.Collection<A>`) naturally has a type parameter denoting the type of its elements. Using labeled types, such a class could be reused directly, passing a labeled type as the corresponding type argument. With our approach, such a class would instead require two parameters – one for the type and another for the policy of the elements – making reuse of existing code ostensibly less viable.

However, in all but the most trivial cases, any use of components from existing Java libraries require the construction of interfaces – i.e. library stub files – that tell the compiler how the fields and methods therein behave with regard to information flow aspects, as pointed out in [8]. Such interfaces would easily incorporate the addition of more type parameters as well, with negligible extra work. Thus we do not see this as a problem in practice.

Locks vs authority and delegation The main strength of Paragon over Jif is the generality of the concept of locks. The Jif notions of authority and delegation are in Paragon (as we will see in more detail in Section 6.4) just special cases of lock families. Queries to the *acts-for* hierarchy in Jif then simply become a particular kind of runtime lock queries in Paragon. Similarly *method constraints*, used to specify constraints regarding the *acts-for* hierarchy and the authority of the calling code, are just special cases of *expects* lock state modifiers.

Furthermore, Paragon allows dynamic changes to locks during program execution, which when considered in Jif terms means that it would e.g. be possible to encode mechanisms to grow and shrink the acts-for hierarchy dynamically. Jif, in the latest version, only allows the acts-for hierarchy to be *extended* dynamically.

Program Counter Declassification In addition to a declassification operation for expressions, Jif contains a block-structured declassification *statement*. Conceptually this allows the level of the current program counter (PC) to be lowered in the context of the given statement. This has an unfortunate consequence: execution paths which bypass declassification statements may still implicitly declassify data. This is hard to justify from a semantic perspective. Since Paragon is designed to enforce a semantic definition of security [12] then this, naturally, cannot be modelled in Paragon.

Exception handling The theory behind exception tracking is similar in Jif and Paragon, and in many ways Paragon has benefitted from Jif's trail-blazing. For instance we did not need to discover for ourselves the potential problems of unchecked exceptions, or the need for null pointer analysis. With the introduction of Generics in Java, the earlier problem with proliferation of dynamic type casts largely went away, leading to much fewer potential occurrences of exceptions from such casts. As a consequence we have opted not to include a runtime type analysis like that in Jif, which predates Java Generics.

Regarding the effect of exceptions on the PC, in Jif methods are annotated with a single "end label", which effectively approximates the effect of *any* exception that could be thrown during its evaluation. This end label is added to the PC after a call to the method. Paragon takes a different approach to reasoning about it by considering for each possibly thrown exception, the surrounding statements that could give away that the exception has been thrown. We refer to this as the exception's *area of influence*. This gives us more fine-grained control, permitting different thrown exceptions to have different effects on the PC. Further, we can accurately pinpoint the end of a given exception's area of influence, allowing us to remove its effect on subsequent statements accordingly.

Static initializers Jif identifies the same problems we do for static initializers, a problem that has also been studied by Nakata and Sabelfeld in [27]. Jif adopts a more restrictive solution to the problem, by restricting the initializers to be exception- and side-effect-free. As noted in section 4.7, Paragon handles side-effects by requiring them to be declared as a modifier to the class.

Type parameters Jif saw the light of day before the introduction of Generics in Java. Still the authors of Jif recognized the need for parameterizing classes on labels and principals, so Jif rolled its own form of parameteric polymorphism, only allowing Jif-specific parameter kinds. While this work is impressive (and as a sidetrack led to the development of PolyJ [28], a competitor to GJ [29] that later became Java Generics), we have the relative luxury of having Java Generics as a starting point. Our type parameter extensions thus syntactically fit more smoothly with what is already available in Java. One particular advantage of this is that we can also build on Java's mechanism for type-parameterized *methods*, and pass arguments intended to affect the method's signature as type arguments instead of formal arguments as in Jif.

Regarding expressive power, we are not aware of any difference between our version and that of Jif, if only actor and policy parameters are considered. Our lock set parameters have no counterpart in Jif.

Aspects of dependent typing Both Jif and Paragon have limited forms of dependent types. Labels and principals in Jif, and policies and actors in Paragon, can be used both as first-class values at runtime, and in the specification of other labels/policies.

In Jif, a label or principal may be hoisted from the value level to the type level, and used as a type argument or in the construction of new labels, assuming the expression representing the label or principal is a *final access path*. To make programming with type arguments smoother, Jif introduces the restriction that formal parameters of a method are always final, so can always be used as the root for a final access path e.g. when writing the signature of the method.

The notion of final access paths carries over to Paragon, where they can be used to specify arguments to parameterized types, or in policy annotations. However, for type parameters on *methods* we do not require finality, as the requirements for consistency of types do not apply in the same way. Also, exactly *because* we have type parameters on methods (unlike Jif), the reason to restrict formal parameters to be final partly falls, since we would pass arguments that affect the method's signature as type parameters instead.

Runtime policies Runtime policies have been studied by Zheng and Myers in the context of Jif and the DLM [17], and their work is the basis for the current evolution of runtime labels in Jif. Their ideas can largely be directly applied to Paragon as well, and the language design regarding runtime policies in Paragon is directly taken from their work.

6.4 Example: Encoding the DLM

Broberg and Sands [12] showed how Paralocks can be used to encode the DLM, proving Paralocks strictly more general in the sense that the DLM policy lattice is a sub-lattice of Paralocks. In this section we show how that encoding can be implemented as a Paragon library. In the next section we will show how the encoding fares when used as a drop-in replacement for Jif in one of the larger case studies that have been conducted for security-typed code.

For the $ActsFor$ hierarchy, we need a lock family that represents a reflexive, transitive relation on actors (as discussed in Section 4.2)

```
public static reflexive transitive
  lock ActsFor(actsForA,A);
```

To mark the authority of any given piece of code we use a unary lock *RunsFor*:

```
public static lock RunsFor(A) {
  RunsFor('x): ActsFor('y,'x), RunsFor('y) };
```

Here no syntactic sugar will do to express the interplay between *ActsFor* and *RunsFor*.

Moving one step closer to Jif, we also introduce an explicit declassification function to ensure that declassification only happens at explicitly marked locations. We make it slightly more sophisticated than the simple ones shown in section 3. We here specify the target policy using a policy type parameter (4.3), and ensure that the argument is no more restrictive than that target policy guarded by the associated `Declassify` lock.

```
private lock Declassify;
private static final policy dec = {'x:
    Declassify};

public static ?TO <A, policy TO>
  A declassify(?(TO*dec) A x) {
    open Declassify { return x; }
}
```

Finally we need to represent DLM *labels* in this framework.The encoding of DLM into Paralocks was shown and proven correct in [12], and the interested reader is referred there for the details. We can simplify the programming of the earlier encoding by expressing it at the level of a single clause rather than a whole label. The policy of a label can then be represented as the *join* of those respective clauses. We can thus define the following type method[8] (4.5):

```
public typemethod policy
  lbl(actor owner, actor... readers) {
    policy c =
      { 'x : RunsFor(owner), Declassify };
    for (actor reader : readers) {
      c += { 'y : ActsFor(reader, 'y) }; }
    return c;
}
```

Our encoding here does not cover the integrity and robustness aspects of Jif. To model integrity a Jif principal could be modeled by *pair* of actors, one for the confidentiality part of the specification and one for the integrity. It turns out that modelling the integrity part of labels is not quite the dual of confidentiality as we initially expected, due to the way the `ActsFor` delegation hierarchy interacts differently with confidentiality and integrity. Integrity labels are actually easier to model than their confidentiality counterparts. However for a useful comparison we would also need to model robustness in this framework, something we have yet to do.

6.5 Case Study

As a case study for comparing Paragon with Jif, we have taken Askarov's development of a non-trivial cryptographic protocol that implements online poker without a trusted third party [8]. At the time (2005) the case study was according to the authors "the largest program written in a security typed language to date" – around 5k lines of Java. The example suits our investigation well since it only uses the confidentiality components of Jif, but still exercises a wide range of features and information flow patterns.[9]

Our first experiment simply used Paragon and the DLM library as a drop-in replacement for Jif. With a mechanical translation of the Jif code to Paragon, the program typechecks and illustrates that we cover at least the same information flow aspects that Jif does[10] . However, a more interesting experiment is to see where Paragon can improve over the Jif solution.

Askarov identifies several different classes of declassification in the application. We note in particular the following:

- Each player has a symmetric key that they use to encrypt their cards before communicating them during the game. The key is released when the game has ended, so the other players may verify the encrypted card hand.
- Each player also uses an asymmetric public-private key pair, for signing encrypted data and committing to the used deck permutation. The public key component is released immediately when the game starts, while the private key component is never released.

From the above we see four different information flow policies: the public key which is always available to the other players; the private key which may never be released, but which must still allow the minimal information flow that goes into signatures; the symmetric key that will be released at the end of the game, and before that allow information to flow into the encrypted values it generates; the cards that will be encrypted and communicated during the game.

Askarov notes further that Jif is not fully equipped to handle properly all the cases. In particular, Jif cannot deal with temporal policies, and for this purpose Askarov devises a programming pattern using so called "seals". Properly used they guarantee the integrity of the symmetric key until the end of the game, but they get no help from Jif: the seals are runtime monitors, implemented separately.

We further note that Jif also cannot separate between different providers of declassification, i.e. different trusted declassifiers [19]. Jif has only one primitive, `declassify`, which has to be used for all cases.

In Paragon we use locks to model the temporal state of the game, to ensure for instance that the symmetric key cannot be released until the game is over, but also that methods using the key for encryption can then no longer be invoked. We also use different locks to represent the contexts of different trusted declassifiers. For instance this allows us to separate the policy of the symmetric key, which influences data being encrypted, from the policy of the private key, which influences data being signed.

The full details of our implementation are out of the scope of this paper, and we refer the interested reader to our website for Paragon [24] where all the examples can be found.

In conclusion, we note that

- Paragon is able to give stronger information flow security guarantees than Jif for this example, since we can model temporal flow properties.
- Paragon can express information flow policies more precisely, for example by distinguishing between different trusted declassifiers.

[8] Note that the *varargs* parameter, denoted by the ellipsis, is vanilla Java 5 and not a Paragon innovation.

[9] A slightly more recent case study [9] offers other "real world" challenges which are worthy of study independently of the Jif-style policies, although this remains a topic for further work.

[10] The case-study used an older version of Jif without integrity labels. The Paragon version compiles essentially as-is whereas recompiling using the *current* Jif seems to require the addition of some integrity labels to policies even though they are not used (c.f. 6.2).

- The syntactic annotation overhead in Paragon and Jif is comparable, and (subjectively) not very large for either language. The runtime overhead is negligible for the example at hand, but we don't expect it to be an issue in practice for larger applications either.

7. 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 Paragon/paralocks, since they are based on datalog-like clauses to express properties like delegation and roles see e.g., [30–33]. 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 is the aforementioned works the only operation of interest is (run-time) querying of rules.

Languages with explicit information-flow tracking Two "real-sized" languages stand out as providing information-flow primitives as types. The first and most closely related to the present work is Jif, which we have already discussed in the previous section. The second is Flow Caml, a subset of OCaml extended with information flow annotations on types. Although Flow Caml only supports simple lattice-based security policies with little flexibility, it is notable that full ML-style type inference is supported, and a metatheory which covers both this and information-flow soundness [34].

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. The Spark Examiner, a commercial tool for static analysis and verification for a safety-critical subset of Ada, contains such an analysis [35].

Hammer and Snelting [36, 37] 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 [38], in particular using a policy lattice based on *Disjunctive Category Labels* (DC labels) [39]. Although DC labels are similar to DLM labels, it seems that Paragon lacks the form of disjunction needed to fully encode them. The Haskell/DC work, however, suggests to us a route to implementing a dynamic version of Paralocks (c.f. [40] mentioned below). Yang *et al*'s *Jeeves* language [41] 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 Expressive Type Systems With suitably expressive type systems and abstraction mechanisms, static information flow constraints can be expressed via a library. Li and Zdancewic [42] showed how to provide information-flow security also as a library. Russo et al [40] improve on this by showing how this can be achieved with a more natural programming style, and including side effects and declassification policies, among which are policies inspired by Flow Locks [13]. Most recently, Morgernstern and Licata [43] show that a rich variety of security policies can be encoded in an extension of the dependently typed programming language Agda.

A number of recent expressive languages aimed at expressing a variety of rich security policies do not have information flow control as a primitive notion (as Paragon or Jif). For example, the authorization policy language Aura can be persuaded to model information flow and declassification polices [44]. Fable [45] focuses on the general idea of label-based policies, allowing user-defined labels and typing constraints (via dependent types). One example is the encoding of a standard information flow lattice policy. A weakness of this approach, according to [46], is that "verification depends on intricate security proofs too cumbersome for programmers to write down". These concerns are in part addressed by Swamy et el's F* [46], which is the culmination of a series of languages (from the same group) including Fine [47], FX [48], and F7 [49]. F* is a full-fledged implementation of a dependently typed programming ML-stye programming language. An impressive collection of security-specific examples have been encoded in F*, although it may be fair to say that information flow is not naturally modelled in this setting, but has to be encoded using e.g. a monadic approach (c.f. [45]).

8. Conclusions and Further Work

We have introduced Paragon, a Java-like language for expressive stateful information-flow policies. We have argued that the combination of Java's encapsulation and the Paralocks policies enables information-flow idioms to be nicely encapsulated so that client applications can freely program with the provided abstractions without the distractions or dangers of seeing their internal representation. Based on the examples we have considered so far, the more complex features of the Paragon language are required only in the construction of information-flow policy libraries, and not in the application code *using* the libraries. Further work is needed to confirm (or refute) this claim.

It remains to build a library of idioms and more deeply explore the practicality of programming with Paragon. The case study by Hicks *et al* [9] identifies a number of areas for improvement in Jif, and many of the issues raised there are relevant to Paragon, in particular the need for extensive support libraries and the problem of lack of debugging tools. We are investigating the latter problem by developing a *dual syntax* for Paragon in which a Paragon program is represented using Java's annotations, thus enabling non information-flow related debugging to take place at the Java level.

The language itself – like almost all the related work – is missing a feature which is rather important for modern programming, namely threads. This direction demands both theoretical and practical work.

References

[1] D. E. Denning, "A lattice model of secure information flow," *Comm. of the ACM*, vol. 19, no. 5, pp. 236–243, May 1976. Cited on page 1.

[2] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J. Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003. Cited on page 1.

[3] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 15, no. 5, pp. 517–548, 2009. Cited on page 1.

[4] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1999, pp. 228–241. Cited on pages 1 and 8.

[5] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom, "Jif: Java information flow," Jul. 2001–2006, software release. Located at `http://www.cs.cornell.edu/jif`. Cited on pages 1, 2, and 8.

[6] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Transactions on Software Engineering and Methodology*, vol. 9, no. 4, pp. 410–442, 2000. Cited on page 1.

[7] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic, "Using replication and partitioning to build secure distributed systems," in *Proc. IEEE Symp. on Security and Privacy*, 2003. Cited on page 1.

[8] 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*, ser. LNCS, vol. 3679. Springer-Verlag, 2005. Cited on pages 1, 9, and 10.

[9] 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. Cited on pages 10 and 11.

[10] S. Chong, K. Vikram, and A. C. Myers, "SIF: Enforcing confidentiality and integrity in web applications," in *16th USENIX Security Symposium*, 2007. Cited on page 1.

[11] N. Nystrom, M. R. Clarkson, and A. C. Myers, "Polyglot: An extensible compiler framework for java," in *In 12th International Conference on Compiler Construction*. Springer-Verlag, 2003, pp. 138–152. Cited on page 1.

[12] 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. Cited on pages 1, 2, 4, 6, 8, 9, and 10.

[13] ——, "Flow locks: Towards a core calculus for dynamic flow policies," in *Programming Languages and Systems. 15th European Symposium on Programming, ESOP 2006*, ser. LNCS, vol. 3924. Springer Verlag, 2006. Cited on pages 2, 4, and 11.

[14] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières, "Securing distributed systems with information flow control," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'08, 2008, pp. 293–308. Cited on page 2.

[15] N. Broberg and D. Sands, "Flow-sensitive semantics for dynamic information flow policies," in *ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security (PLAS 2009)*. Dublin: ACM, June 15 2009. Cited on pages 2 and 8.

[16] B. van Delft, N. Broberg, and D. Sands, "A datalog semantics for paralocks," 2012, chalmers University of Technology. Cited on pages 3 and 6.

[17] L. Zheng and A. C. Myers, "Dynamic security labels and static information flow control," *Int. J. Inf. Sec.*, vol. 6, no. 2-3, pp. 67–84, 2007. Cited on pages 3 and 9.

[18] A. Myers, A. Sabelfeld, and S. Zdancewic, "Enforcing robust declassification," in *Proc. IEEE Computer Security Foundations Workshop*, Jun. 2004, pp. 172–186. Cited on pages 3 and 8.

[19] B. Hicks, D. King, P. McDaniel, and M. Hicks, "Trusted declassification: High-level policy for a security-typed language," in *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, June 10 2006. Cited on pages 4 and 10.

[20] A. Almeida Matos and G. Boudol, "On declassification and the non-disclosure policy," in *Proc. IEEE Computer Security Foundations Workshop*, Jun. 2005, pp. 226–240. Cited on page 4.

[21] D. Aspinall, "Subtyping with singleton types," in *In Eighth International Workshop on Computer Science Logic*. Springer-Verlag, 1995, pp. 1–15. Cited on page 6.

[22] L. Zheng and A. C. Myers, "Dynamic security labels and static information flow control," *International Journal of Information Security*, vol. 6, 2007. Cited on page 7.

[23] BlindedReference, 2011. Cited on page 7.

[24] "Paragon website," Website, 2012, http://www.cse.chalmers.se/~d00nibro/paragon. Cited on pages 8 and 10.

[25] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *Proc. ACM Symp. on Operating System Principles*, Oct. 1997, pp. 129–142. Cited on page 8.

[26] S. Zdancewic and A. C. Myers, "Robust declassification," in *Proc. IEEE Computer Security Foundations Workshop*, Jun. 2001, pp. 15–23. Cited on page 8.

[27] K. Nakata and A. Sabelfeld, "Securing class initialization," in *Trust Management IV - 4th IFIP WG 11.11 International Conference, IFIPTM 2010, Morioka, Japan, June 16-18, 2010. Proceedings*, vol. 321. Springer, 2010. Cited on page 9.

[28] A. C. Myers, J. A. Bank, and B. Liskov, "Parameterized types for Java," in *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL) , Paris, France*, Jan. 1997, pp. 132–145. Cited on page 9.

[29] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler, "Making the future safe for the past: adding genericity to the java programming language," in *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '98, 1998. Cited on page 9.

[30] T. Jim, "SD3: A trust management system with certified evaluation," in *Proc. IEEE Symp. on Security and Privacy*, 2001. Cited on page 11.

[31] N. Li, J. Mitchell, and W. Winsborough, "Design of a role-based trust-management framework," in *IEEE Symposium on Security and Privacy*, 2002, pp. 114–130.

[32] D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "Specifying and reasoning about dynamic access-control policies," in *IJCAR*, ser. LNCS, vol. 4130. Springer, 2006.

[33] M. Y. Becker, C. Fournet, and A. D. Gordon, "Design and semantics of a decentralized authorization language," in *Proc. IEEE Computer Security Foundations Symposium*. IEEE Computer Society, 2007, pp. 3–15. Cited on page 11.

[34] F. Pottier and V. Simonet, "Information flow inference for ML," *ACM TOPLAS*, vol. 25, no. 1, pp. 117–158, Jan. 2003. Cited on page 11.

[35] R. Chapman and A. Hilton, "Enforcing security and safety models with an information flow analysis tool," *ACM SIGAda Ada Letters*, vol. 24, no. 4, pp. 39–46, 2004. Cited on page 11.

[36] 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*, vol. 8, no. 6, pp. 399–422, 2009. Cited on page 11.

[37] C. Hammer, "Experiences with pdg-based ifc," in *Engineering Secure Software and Systems, Second International Symposium*, 2010, pp. 44–60. Cited on page 11.

[38] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible dynamic information flow control in Haskell," in *Proceedings of the 4th ACM symposium on Haskell*, 2011. Cited on page 11.

[39] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell, "Disjunction category labels," in *16th Nordic Conference on Security IT Systems, NordSec*, ser. LNCS, vol. 7161. Springer, October 2011, pp. 223–239. Cited on page 11.

[40] 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. Cited on page 11.

[41] 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. Cited on page 11.

[42] P. Li and S. Zdancewic, "Arrows for secure information flow," *Theor. Comput. Sci*, vol. 411, no. 19, 2010. Cited on page 11.

[43] 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. Cited on page 11.

[44] 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. Cited on page 11.

[45] N. Swamy, B. Corcoran, and M. Hicks, "Fable: A language for enforcing user-defined security policies," in *Proc. IEEE Symp. on Security and Privacy*, 2008, pp. 369–383. Cited on page 11.

[46] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bharagavan, and J. Yang, "Secure distributed programming with value-dependent types," in *The 16th ACM SIGPLAN International Conference on Functional Programming*, 2011. Cited on page 11.

[47] N. Swamy, J. Chen, and R. Chugh, "Enforcing stateful authorization and information flow policies in fine," in *In Proceedings of the European Symposium on Programming (ESOP)*, 2010. Cited on page 11.

[48] J. Borgstrom, J. Chen, and N. Swamy, "Verifying stateful programs with substructural state and Hoare types," in *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, ser. PLPV '11, 2011. Cited on page 11.

[49] K. Bhargavan, C. Fournet, and A. D. Gordon, "Modular verification of security protocol code by typing," in *POPL*, 2010. Cited on page 11.