Explicit information flow properties in JML

Christian Haack[1], Erik Poll[1], and Aleksy Schubert[2]

[1] Radboud University Nijmegen, the Netherlands
[2] Warsaw University, Poland

Abstract This paper considers how explicit information flow properties can be expressed and verified in a traditional program logic, using pre- and postconditions. As concrete specification language for expressing these properties we use the specification language JML for Java.
For this we propose a new classification of information flow properties, namely positive and negative properties, where the former are easier to describe. This also leads us to reconsider the duality between integrity and confidentiality in the light of the difference between explicit and implicit information flows.

1 Introduction

An important class of security requirements of programs are information flow properties, which can be about confidentiality or integrity of data. It is well-known that confidentiality and integrity are duals, and hence many accounts only consider one, typically confidentiality. However, as we will try to argue in this paper, there are limits to this duality, and by only thinking of one as the dual of the other one may miss interesting differences.

The motivation for this work were the security requirements encountered for case studies in the EU project Mobius[1]. These case studies include Java J2ME MIDP applications, i.e. Java applications that run on mobile phones. Typical security requirements for such applications include information flow requirements [14,13]. The Mobius project investigates the possibility of Proof-Carrying Code (PCC) to certify such security properties, and we are therefore interested in ways of expressing such properties in the program logic used in the PCC infrastructure.

There have been many efforts to express and verify information flow properties in more traditional program logics, e.g. [12,7,1,20,3]. In this paper we try to do the same for *explicit* information flow properties, as some of the requirements we encounter are about explicit information flow. Expressing explicit information flow in a traditional program logic turns out to be simpler than implicit information flow – as one might expect – but there are several ways to do this, each with its merits, as explained in Section 4.

Although taking into account implicit information flows is more challenging, and somehow less ad-hoc than only considering explicit flows, in many situations

[1] http://mobius.inria.fr

only explicit information flows matter, especially for integrity requirements. This led us to re-consider the duality between confidentiality and integrity, to understand why this should be the case.

The outline of the rest of this paper is as follows. We begin by reviewing the difference between explicit and implicit information flow in Section 2. Section 3 then introduces the distinction between positive and negative information flow properties, and Section 4 describes the ways in which such requirements – for explicit information flow – can be expressed in standard pre- and postconditions. We reconsider the (lack of) duality between confidentiality and integrity in Section 5, and conclude in Section 6.

2 Background: explicit vs implicit information flows

The distinction between *implicit* and *explicit* information flow dates back to [8] and is best explained with an example: the program

```
l := h;
```

has an explicit flow of information from `h` to `l`, whereas the program

```
if (h>0) then { l := 0; }
```

has an implicit flow of information from `h` to `l`.

In general, an explicit flow of information from `h` to `l` occurs by an assignment of the form `l :=` E`(h)` for some expression E`(h)` that contains `h`. An implicit flow of information from `h` to `l` is any flow of information that is not due to such an assignment, but is due to the surrounding control flow – control flow that depends on `h` and somehow influences the value of `l`.

To capture all flows, implicit flows should be taken into account. Still, just looking at explicit flows can be perfectly adequate in some practical situations. In particular, when looking at integrity properties, it is not so clear that implicit flows are an issue. For instance, if we want to avoid an untrusted value `h` contaminating the trusted value `l`, the first program should be rejected, but is not so clear that the second program should be rejected. (We get back to this issue in Section 5, when we discuss possible differences in attacker models for integrity and confidentiality.) Moreover, as we discuss at the end of Section 3, ignoring implicit flows, and even ignoring some explicit flows, may be 'sound' (or 'safe') in some situations.

The standard semantical characterisation of information flow – including implicit flows – uses the notion non-interference. By its very nature any characterisation of explicit flows only has to be more 'syntactical'. A natural way to express explicit information flow is through typing, i.e. by introducing special types to distinguish elements of type `T` that are confidential or (un)trusted.[2]. Implicit information flow properties can also be enforced through typing. The best

[2] In Java the meta-data mechanism could be used to express such additional type information. Indeed, the standard tags being considered by JSR305 [10] include a tag `@Tainted` to keep track of unvalidated – or tainted – inputs.

known example of this is the Jif type system for Java [15]. However, type systems that take implicit flows into account quickly become extremely restrictive.

Type systems for information flow can be enforced statically or dynamically: types can be tracked at run-time, as happens with tainting in Perl [19], or at compile-time, as happens in various static analysis tools that look for missing input validation such as [9].

An alternative is to express the property in some program logic, and use a generic program verifier to check them. Because we are interested in certifying security requirements by means of Proof Carrying Code (PCC), this is the approach we will explore in Section 4.

3 Positive and Negative Information Flow Policies

As a general setting, we assume availability of an input channel `read` and an output channel `write` of some type `T`:

```
class In {
  static T read(){...}
}

class Out {
  static void write(T t) {...}
}
```

The input channel provides a *source* of data, the output channel a *sink*. Our information flow properties will be about (dis)allowed flows from source to sink. Such properties can be about integrity or about confidentiality.

Irrespective of whether implicit flows are taken into account, two converse types of information flow properties can be distinguished in our setting:

(P) Data flowing to `Out` should be obtained from `In`.
(N) Data obtained from `In` should *not* be fed into `Out`.

Property (P) allows certain information flows, implicitly disallowing all others; we call this a *positive property*. Property (N) forbids certain information flows, implicitly allowing all others; we call this a *negative property*.

An example of a positive property is a property limiting an application so that it only makes phone calls to numbers obtained from the user's address book; this property is concerned with the integrity of numbers used as telephone numbers.

An example of a negative property is a property forbidding numbers obtained for the user's address book to be included in text messages; this property is concerned with the confidentiality of numbers in the address book.

A positive property can also be about confidentiality, for instance if the only public data considered by the application is obtained from `In` and therefore only that data may be passed onto the public output channel `Out`. Dually, a negative property can also be about integrity, for instance if data from `In` is untrusted (e.g. when it is a user's input) and should not be passed to sensitive `Out` (e.g. when it is a database controller).

Generalisation: multiple sources and sinks Our setting could be generalised by having multiple sources and sinks. If we allow multiple sources and sinks, any negative property could be expressed as a positive one, and vice versa. After all, a positive property (P) for source `In` can also be seen as a negative property for all sources apart from `In`.

However, for practical purposes this can make a big difference: enumerating all the sources apart from `In` may be a lot of work. For instance, imagine having to take into account all sources of Strings in the Java API. So the effort involved in writing a positive property as a negative property can be huge. The property itself could become very long, unless the language for properties allows some abbreviation mechanism for expressing things like 'all other sources apart from `In`'.

Generalisation: additional operators Another way to generalise our setting is to take into account operations on type `T` and the dependencies these introduce. For instance, if `T` is the type of strings, information flow properties could take into account the concatenation of strings. More generally, if `T` is some object type, one might want to – or have to – take cloning of objects of type `T` into account.

Given that an operation such as cloning is an *explicit* way to leak information (after all, it is due to an actual operation, not some surrounding control flow), ignoring such an operation means ignoring some explicit flows. This means that there is not just a choice between information flow with or without implicit flows, but that explicit information flow comes in many flavours, depending on which operations we take into account.

If there are many operations this can become quite complicated. For instance, the Java API+ offers many operations for strings, and considering all of them would be a lot of work. (An advantage of implicit information flow properties is that these avoid this complication, by considering all dependencies irrespective of the concrete operations that could be used to introduce these dependencies.)

There is an important difference between positive and negative properties here. For a positive property ignoring some operation, say cloning, would be 'safe': the property may become more restrictive than it has to be, but it does not violate the restrictions on information flow we want to enforce. For a negative property, ignoring operations would be 'unsafe'. For example, if we are not allowed to pass data from `In` to `Out`, but we can pass this data to `Out` after cloning it, this provides a potentially harmful way to circumvent restrictions on information flow.

The same holds even for ignoring implicit flows: ignoring implicit flows is always safe for positive properties, but is potentially unsafe for negative properties. This is interesting, because taking implicit flows into account complicates any analysis; so for positive properties it may be possible to safely ignore this complication.

4 Expressing properties in pre- and postconditions using JML

We now consider ways in which explicit information flow properties such as (P) and (N) could be expressed in standard pre- and postconditions. We use JML syntax [4] to write these pre- and postconditions, but this is not really important (except when it comes to the various forms of specification-only fields – ghost and model fields – that JML provides).

Intuitively, the positive property (P) comes down to imposing a precondition on `write` that the argument must have come from `read`. For the negative property (N), we should express a precondition on `write` stating that the argument cannot have come from `read`. There are various ways to formalise this, as we will consider below.

4.1 Using sets

One approach to express the positive property (P) is to assume the existence of some set `TRUSTED` of objects of type `T` and then specifying

```
class In {
  //@ ensures result ∈ TRUSTED;
  static T read() {...}
}

class Out {
  //@ requires t ∈ TRUSTED;
  static void write(T t) {...}
}
```

Here `requires` and `ensures` are JML's notation for pre- and postconditions, respectively.

The two contracts above express property (P): if we verify some client code against these contracts, and we do not make any additional assumptions about the set `TRUSTED`, then the client code obeys property (P).

The name `TRUSTED` suggests we are dealing with an integrity property, but it could just as well be a confidentiality property, though then a name such as `NOT_CONFIDENTIAL` would be more intuitive.

This formalisation can easily accommodate additional operations to produce trusted elements. E.g., if `T` is the type `String`, then an assumption that the set `TRUSTED` is closed under concatenation could be expressed as a pre- and postcondition of the concatenation method

```
class String
...
 //@ requires this ∈ TRUSTED && str ∈ TRUSTED;
 //@ ensures result ∈ TRUSTED;
 String concat(String str)
```

or as some axiom

```
forall String t1,t2;  t1∈TRUSTED && t2∈TRUSTED ==> t1+t2∈TRUSTED
```

in the theorem prover used to verify the specifications. With these, the contracts for `read` and `write` express a more liberal policy, which also allows concatenation of trusted strings to be fed to `write`.

Trying to express the negative property (N) in a similar way, as shown below, fails.

```
class In {
  //@ ensures result ∈ TRUSTED;
  static T read() {...}
}

class Out {
  //@ requires t ∉ TRUSTED;
  static void write(T t) {...}
}
```

These contracts do indeed prevent any data being passed from `read` to `write`. The problem is that they do not allow any data to be passed to `write`: we have no way of knowing – let alone proving – that some element of type `T` is *not* in `TRUSTED`. To do this, we would need to annotate all other sources to specify that these produce data that is not in `TRUSTED`. But note that then:

– We are effectively specifying the information flow requirements as a positive policy.
– The specifications for the method `read` is superfluous. (In fact, it only introduces the danger of logical inconsistencies, as discussed in Section 4.4.)

So, in the end, any information flow requirement has to be specified as positive policy to apply this approach.

Remark
There are some technical issues with the use of the mathematical concept of sets in the specifications above. JML provides so-called model classes for this purpose, though there are several proposals to improve on this [11,6,2]. More generally, in any imperative programming language there is the issue that there are two notions of equality, in Java written as `==` and `.equals`. If `T` is an object type, one has to decide if set membership is defined modulo `.equals` or modulo

`==`. Intuitively, the natural choice would be the former, but if objects of type `T` are immutable, then also the possibility to use the latter is open.[3]

Using `.equals` as notion of equality makes the formalisation of (P) given above slightly more flexible than a type-based approach would be. For example, it would allow, in the most straightforward formulation, the program

```
m(T h){
  T l = In.read()
  if (l.equals(h)) {Out.write(h);}
}
```

whereas a type-based analysis would reject this. In some applications, this may be useful e.g. when the knowledge that some public information is secret is not essential. Still, we can exclude cases such as this by adding a precondition to `.equals` which imposes that both the method receiver and parameter are at the same time either trusted or not trusted. ∎

4.2 Using ghost fields

An alternative way to express information flow properties, which does not require the use of set-theoretic expressions in specifications, is to add some additional fields in objects to keep track of their security level, i.e. of whether they are confidential or (un)trusted. Clearly, this only works if `T` is a type of objects, and not it is as type of primitive values, such as `integer`. In spirit, this is very close to a type-based approach, in that the extra field just codes up type information.

JML offers the possibility of making this extra field a specification-only field, or *ghost* field. A ghost field could be added to class `T`, or it could be added to the `Object`, ensuring that all objects have the field.

For property (P) adding a boolean ghost field to class `T` would suffice:

```
class T {
   //@ ghost boolean isTrusted;
   ...
}
```

The name `isTrusted` is simply chosen to express the informal meaning of the property.

The requirements on `In` and `Out` can now be expressed by the following contracts

```
class In {
  //@ ensures \result.isTrusted;
  static T read() {...}
}
```

[3] Note that except from the immutability you need also the control over the creation of objects to exclude the double creation of the same object.

```
class Out {
  //@ requires t.isTrusted;
  static void write(T t) {...}
}
```

Together, the two contracts above express the property (P): Verification of some client code would ensure it obeys the property (P).

Note the similarity with the approach using sets. Effectively, all we are doing is using this boolean field to record membership of the set `TRUSTED`. This approach seems simpler in that it does not require the use of set theory in specifications. For negative properties we run into exactly the same issue as when using sets as in Section 4.1: we are forced to specify it as a positive one.

A difference between this approach with the earlier approach using sets is that this approach does not work if objects of type `T` are mutable. For instance, if `T` is the array type `int[]` or the type `StringBuffer`, both of which are mutable in Java, then changing the content of the array or the stringbuffer would leave the `isTrusted` field unchanged, introducing a loophole in the formalisation. The approach using sets does not have this problems, assuming membership of sets is defined up to `.equals()`.

4.3 Using model fields

JML offers two kinds of specification-only fields, namely ghost and model fields. Instead of ghost fields we could also use model fields to express the security level of objects.

A ghost field is just like a regular field, except that it is only to be used in specification. Adding a ghost field to a class introduces a new piece of state for all objects of that class. A model field, on the other hand, is more like an abstract value for abstract data type: it does not introduce new state, but provides some abstraction of the existing state. The value of the abstract model field should be definable in terms of the values of the concrete field by a representation relation, by specifying an abstraction relation (or function). So a model field of an object is similar to an abstract value for an abstract data type.

In particular, this means that a model field may change its value whenever the 'state' of the associated object changes. We can use this property to our advantage, when it comes to dealing with mutable objects. If the relation between the value of an abstract model field and the concrete state of an object is left unspecified, then the value of the model field may change in an unspecified way whenever the state of the object changes.

So if a model field is used rather than a ghost field to capture the security level of an object, then mutable objects can be dealt with. A disadvantage of using a model field is that it is a more complicated concept that any program verifier we have to use will have to support.

We will not spell out the example using model fields here. Essentially, the only difference would be replacing the keyword `ghost` with the keyword `model`. For a detailed discussion of the difference between ghost and model fields we refer to [5].

4.4 Comparison

All the three approaches have the limitation that information flow requirements will have to be expressed as positive properties.

Using an extra (ghost or model) field to track the confidentiality or integrity of data is in spirit very close to having a type system. A limitation of the approach using ghost or model fields is that it can only deal with information flow requirements for objects, not for primitive values such as integers.

A further limitation of using ghost fields is that it only works for immutable objects, such as Java Strings, and not for mutable objects, such as arrays. However, as many information flow requirements seem to be about Java strings, this limitation may not be important in practice. [4] The table below summarises this:

Approach	is possible for:		
using	primitive types	immutable objects	mutable objects
sets	yes	yes	yes
ghost fields	no	yes	no
model fields	no	yes	yes

There is a possible danger with the set approach if there are both positive and negative assumptions about data are specified: there is then a risk of introducing logical inconsistencies. For example, suppose we have

```
//@ ensures result ∈ TRUSTED
static String source1() {...}

//@ ensures result ∉ TRUSTED
static String source2() {...}
```

Now if `source1` and `source2` produce the same string at some stage, we have a logical inconsistency. The approach using model field suffers from the same problem. With ghost fields we do not have this problem, as two copies of the same string could have different values for their tainting field. (Of course, there one has to be careful with the meaning of `.equals()`, and whether identical strings with different integrity/confidentiality levels are to be regarded as `.equals()` or not.)

If we stick to only expressing only positive information flow properties, there is no danger of running into inconsistencies: there will then only be assumptions about data being in `TRUSTED`, and no assumptions about data not being in `TRUSTED`. It will then not be possible to derive any contradiction from these assumption – assuming the program logic we use to reason about program is sound, of course.

[4] It is of course no coincidence that security-sensitive arguments tend to be immutable objects. If untrusted and trusted code exchange data it introduces a security risk if this data is mutable. This is the reason why Strings *must* be immutable in Java. Still, the Java API has some surprises here; for instance, URIs are immutable, but URLs are not.

5 Integrity vs Confidentiality

Looking at typical security requirements for information flow, the duality between confidentiality and integrity appears to break down when we consider the difference between explicit and implicit flows: somehow implicit flows seem to be less of an issue for integrity requirements.

Our distinction between positive and negative properties explains why implicit flows may be safely ignored in some cases, namely for positive properties. However, this does not explain why implicit flows do not seem to matter for many integrity properties. There is no reason to expect that integrity properties are more likely to be positive properties. In fact, one would expect that most information flow requirements are most naturally expressed as negative properties, irrespective of whether they are about confidentiality or integrity.

An essential difference between confidentiality and integrity is noted in [16]: "a computing system can damage integrity without any interaction with the external world, simply by computing data incorrectly." We do not agree that integrity requirements are therefore harder to enforce, as [16] goes on to claim; rather, integrity requirements seem to be easier to enforce, as implicit flows usually do not seem to matter for most integrity requirements.

The difference between integrity and confidentiality noted above highlights that there may be subtly different attacker models. If an application A executes on a platform P and communicates with the outside world W, then for an attack by A on the integrity of P no communication with the outside world is needed. However, most approaches aimed at detecting security flaws that damage integrity, say detecting SQL injection vulnerabilities in web-applications, are not concerned with direct attacks by A on the integrity of P, but are only concerned by attacks in which A is tricked into attacking P by the outside world W, by means of some malicious input. In other words, A is trusted to some degree, and only W is completely untrusted.

This goes some way towards explaining why implicit flows are not considered a problem for most integrity requirements: we want to rule out direct information flows from W to P via A as this would allow W to attack P, but we do allow implicit information flows from W to P via A, because we trust A enough not do harm us with such indirect flows.

However, in principle, one could use the same reasoning to discount implicit flows when considering confidentiality requirements.

Two notions of integrity?

A more fundamental difference between confidentiality and integrity is that there seem to be two, fundamentally different, notions of integrity:

- A *flow-based notion*, which is concerned with preventing flows from untrusted input channels or untrusted data to trusted output channels or trusted data. For such a flow-based notion, there is a liberal variant that prohibits explicit flows only, and a strict variant that also prohibits implicit flows.

– A *format-based notion*, which is not concerned with the origin of data, but only with the 'format': the aim is to prevent malformed data from being sent to some trusted output channels (or critical API call) or from being used as part of trusted data. For example, a string used as phone number should only contain numbers, and should maybe start with a certain prefix.

The latter notion can be characterised by a predicate on the data type in question. The former notion cannot; for this we need to look at program traces or – as is done with non-interference – at multiple executions.

Most input validation routines involve some check on the format of input (e.g. strings should not contain semicolons or other dangerous characters), possibly in combination with some operation to make potentially harmful data harmless (e.g., by 'escaping' dangerous characters). Clearly such input validation routines are concerned with the latter notion of integrity. An input validation routine that checks digital signatures would address the former notion of integrity.

For the format-based notion of integrity there is no obvious dual. Also, note that *output* validation can be used to ensure constraints on formats just as well as *input* validation.

The two notions of integrity are related in that unwanted flows may cause malformed data: an attacker will try to exploit unwanted flows from input channels to output channels to create malformed data that is harmful. Exploiting explicit flows to do this is easier than implicit flows. If we have some confidence (or trust) in a given application, but no confidence in the people supplying inputs to that application, then explicit flows are clearly the first thing to worry about.

However, not all explicit flows are harmful in such a situation, as an input validation routine that checks for a correct format may well leave an explicit flow intact. For instance, the check on the format of an untrusted phone number in

```
if phone_number.startsWith("0800") { dial(phone_no); }
```

does not remove the explicit flow to the method call `dial`.

6 Conclusions

We have shown that *explicit* information flow properties can be expressed in pre- and postconditions, and that there are several ways of doing this in JML, as shown in Section 4. This means that a standard program logic to could be used to verify such properties, and a standard PCC infrastructure based on such a program logic could certify them.

For this we introduced a distinction between *positive* and *negative* information flow properties in Section 3. To express information flow properties in pre- and postconditions they have to be formulated as *positive* properties. Although in theory any negative property can be expressed as positive one (and vice versa), there can be a big difference in the practical effort required to express a given property in positive or in negative form, and comparable difference in the size of the resulting specifications (in the number of pre- and postconditions). Some automation in providing the specification or some abbreviation mechanism may be required in practice to deal with properties that are most naturally described as a negative information flow property.

One interesting observation about positive information flow properties is that for these properties is safe (albeit overly restrictive) to disregard some information flows – notably implicit flows. This can offer a way to avoid the complication of having to take implicit flows into account: if an information flow requirement is expressed as a positive property, than any analysis that ignores implicit flows is sound but not complete. This is interesting, because taking implicit flows into account is so much complicated than only dealing with explicit flows. That it can be sound to disregard implicit flows for certain policies is somewhat counter-intuitive.

Only in retrospect did we realise that positive policies correspond with white-listing and negative policies with blacklisting. It is a standard recommendation to use whitelisting rather than blacklisting. The rationale for this is that having an incomplete blacklist is insecure, whereas an incomplete whitelist is not. This would also be a reason to prefer positive policies over negative ones.

Finally, we note that the duality between integrity and confidentiality appears to break down when one considers typical information flow requirements: for confidentiality requirements it is natural to worry about implicit flows of information, but for most integrity requirements it seems natural to only consider explicit flows. Indeed, there appears to be a dichotomy in research into information flow: people either investigate confidentiality requirements and include implicit flows (using non-interference as notion of information flow) or they investigate integrity requirements and ignore implicit flows. (A comprehensive overview of the former line of work is given in [16]. Examples of the latter line of work are the taint mode in Perl [19] and Ruby [18] and approaches to statically or dynamically detect security flaws due to missing input validation, e.g. [9,17].) Section 5 makes some attempt at understanding why implicit flows do not seem to matter for integrity, but it is hard to put a finger on this. Whatever the cause, this difference between integrity and confidentiality suggests that any approach based on non-interference will be unsuitable for many integrity requirements. Expressing a rich set of information flow requirements may ultimately require support for two notions of information flow, one including and one excluding implicit flows.

Acknowledgements This work is supported by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the Mobius project (IST-2005-015905).

References

1. Ádám Darvas, Reiner Hähnle, and Dave Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors,

Proc. 2nd International Conference on Security in Pervasive Computing, volume 3450 of *LNCS*, pages 193–209. Springer, 2005.

2. Ádám Darvas and Peter Müller. Faithful mapping of model classes to mathematical structures. In *Specification and Verification of Component-Based Systems (SAVCBS)*. ACM Press, 2007.
3. Lennart Beringer and Martin Hofmann. Secure information flow and program logics. In *Computer Security Foundations Symposium*, pages 233–248. IEEE, 2007.
4. L. Burdy, Y. Cheon, D.C. Cok, M.R. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.
5. Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *FMCO'2005*, volume 4111 of *LNCS*, pages 342–363. Springer, 2005.
6. Julien Charles. Adding native specifications to JML. In *ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP'2006)*, 2006.
7. Pedro D'Argenio, Gilles Barthe, and Tamara Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Computer Security Foundations Workshop*, pages 100–114. IEEE Press, 2004.
8. Dorothy. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, July 1977.
9. David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.
10. JSR 305 Expert Group. Annotations for software defect detection. Technical report, Java Community Process, 2006.
11. Thierry Hubert and Claude Marché. A case study of C source code verification: the Schorr-Waite algorithm. In *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, Pune, India, 2006. IEEE Comp. Soc. Press.
12. R. Joshi and K.R.M Leino. A semantic approach to secure information flow. *Science of Comput. Progr.*, 37(1-3):113–138, 2000.
13. The Mobius Consortium. Report on framework-specific and application-specific security. Deliverable D1.2 produced as part of the EU IST project Mobius, `http://mobius.inria.fr`, September 2006.
14. The Mobius Consortium. Report on resource and information flow security requirements. Deliverable D1.1 produced as part of the EU IST project Mobius, `http://mobius.inria.fr`, March 2006.
15. A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241. ACM, 1999. Ongoing development at `http://www.cs.cornell.edu/jif/`.
16. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
17. Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, pages 201–220, 2001.
18. [8] Dave Thomas and Andy Hunt. *Programming Ruby: A Pragmatic Programmers Guide*. Addison-Wesley, 2000.
19. Larry Wall and Mike Loukides. *Programming Perl*. OReilly, 2000.
20. Martijn Warnier. *Language Based Security for Java and JML*. PhD thesis, Radboud University, Nijmegen, The Netherlands, 2006.