Programming With Information Flow-Control

Yannick Zakowski

September 3, 2012

Internship report, Magistère Informatique et Télécommunication, ENS Cachan/Rennes and University of Rennes 1

Internship performed from the 15th of May 2012 to the 25th of August 2012 in the *University of Chalmers*.

Supervisors : David Sands, Niklas Broberg, Bart van Delft.

Chalmers, Göteborg University.

Abstract

We informally introduce here the information flow-control domain. We then present two theoretical layouts for developping programming language with information flow-control. The approach is semantic-based and allows programs to be statically type-checked. We then consider its practical use, through the so-called Paragon language, extending Java in a user-friendly way. We'll be especially intersted in issues about its compiler, written in Haskell. Specifically, we'll take a close look at both the problem of infering non-specified policies, and the performance of a null-pointer analysis.

Keywords: Computer security, information flow-control, security-type system, implementation, paragon.

Introduction

Information security, as an economically and military sensible field, has been widely studied and developped. But it certainly is a vast domain, and one can think of it as being roughly partitioned in three big pictures. In order to illustrate it, let's draw a parallel with some highly confidential military intelligence and think of our expectations about its protection.

Obviously, the intelligence can't be readen by any non-expressively authorized person, and the guards will make it sure to stand! Tranposed to computer science, this leads to the *access control* field, which is well known and dealt with both academically and industrially. Its purpose is security at the system *border.*

Now and then, the intelligence may have to be communicated to an other base, and it must be ensured that it can't be intercepted nor modified! It obviously is the very object of *encryption,* dealing with security *outside* of the system.

But there remains a security issue, so obvious in the military case that one can easily miss it: what about all these expensive measures if the general, legitimately working on the intelligence, makes a draft of which he then casually gets rid in the main hall? This is all the point of *information flow-control,* ensuring security *inside* the system.

If the first two fields are quite old, the consciousness of the importance of the last issue, and its understanding, have been developped more recently. Especially, there are very few prototypes which could be considered in an industrial context. Niklas Broberg, in his thesis [7], has not only developped a new brillant approach, defining a powerful semantic based language, to specify the concepts underlying a program secured from a *information flow-control* point of view. He also has put it in practice by extending *Java* in order to obtain an actual programming language.

The object of this document is to introduce the mechanism behind this new language and expose my work realized on its compiler in order to address two specific subproblems.

The report is organized as follow: in a first section, we introduce informally the *information flow-control* issue; the second section presents *FlowLocks*, the first semantic based language developped by Broberg; the third one covers its extension, *ParaLocks*; the last section is dedicated to *Paragon*, and focus on my personal work by addressing both the policy inference issue and the null-pointer analysis topic.

1 Information Flow-Control

Speaking of *information flow-control* assumes that your system possesses different channels, which can be physically accessed by different entities. Therefore, the successive values taken by some variables will be readable by anyone, while others will remain secret for most. Supposing that we quite naturally want to use the first one to store our sensitive information, our purpose is to conceive a programming language in which one no information about a channel someone don't have access to can be infered by observing the other ones.

In order to give a feeling of how to put it in practice, let's consider the elementary case of two variables h, containing sensible information, and l, whose successive taken values are public. Figure 1 presents some programs that we would like to forbid.[1].

Assuming that the source code is freely available, you quite obviously do not want to allow the value of h to be assigned to l. We therefore want an assigment to be allowed if and only if the variable on the left-hand side is more "private", notion we obviously need to specify, than every single one appearing on the right-hand side.

But a leak can also arise indirectly, as illustrated in the second example. We therefore can't allow in the body of a conditionnal statement any assignment to a more public variable than

[1]If the reader wants to have the pleasure to discover by himself the basics, he can try to hack the successive type-system suggested here: http://ifc.hvergi.net/. It's both didactic and quite enjoyable :)

Figure 1: Examples of potentially leaking programs

$Ex1:$	$l \leftarrow h$	$Ex3:$	**while** $(h = 0)$ **do** ...
$Ex2:$	**if** $(h = 0)$ **then** $l \leftarrow 0$ **else** $l \leftarrow 1$	$Ex4:$	**if** $(h = 0)$ **then** $compute(100000!)$ **else** $skip$

the most private one used to compute the guard.

The third example introduces in a rather extrem way a new notion, the use of a temporal dimension to infer a property: the value of h depends on the terminaison of the loop. The fourth one is a more subtle illustration to this idea that the time spent to execute a branch can be exploited to infer information on the value of the guard.

Those latters give a feeling of the diversity of sources of insecurity. A survey of the domain, realized by Sabelfeld and Myers [2], is quite astonishing, including probabilistic channels for instance. Each of these exotic ways to leak some information raising the necessity of a specific study, and adequate tools, the "secured" programming layout are legions, but deal with nearly as much different definitions of security. It raises an obvious but difficult necessity: a general declassification framework. Sabelfeld and Sands have given it a first precious attempt, classifying the basic goals according to the so-called *what, who, where and when* [3] parameters of the leak.

This complexity highlights the necessity of defining the capability of the attaquant that we consider, and what is a secured program according to such a threat.

What's more, we actually do want to allow some leaks. Otherwise, we couldn't even control the validity of a password, since answering whether it's the good one gives some information about its value! But we want to restrict these leaks to an explicitely specified area so that any one of them is consciously conceded. Therefore, we need a new construction, *declassify*, which allows us to ponctually get rid of the degree of privacy of a channel.

In the following sections, we will describe the language developped by Broberg in its thesis to specify *information flow policies*, its solution to the *flow-control* with *declassification* problem.

2 FlowLocks

The object of *flow security* is both specifying the notion of a "secured" program, and developing a process able to decide if a given program is actually one such. As we have seen in the previous section, it requires the introduction of new features in our language in order to solve a few challenges:

- modelling different levels of confidentiality;
- formalizing the notion of attaquant and consequently of a secured program;
- introducing a construction allowing us to *declassify* an information.

We present in this section *FlowLocks*, a semantic-based language answering these issues. The formalism being quite rich and our place quite restricted, we'll strive for giving a good understanding to the reader in a quite informal way.

2.1 Motivating example

Lets consider the present situation, quite artificial but for pedagogical purpose. A student is giving a talk in front of a jury composed by two members. We model it through:

- a variable *talk* used by the student to show its prestation: everyone can read it at any time;
- a variable $mark_i$ for each judge to store its secret grade;
- a variable *average* which will contains the average mark, and be readable by anyone, but only once the prestation is over!

Actually, the individual marks can't be completely secret since we want to use them together to compute the average one. We need a trusted agent that will be in our case the chairman of the jury.

The idea behind *FlowLock* is to assign to each variable a *policy* which regulates whose *actors* ase authorized to access it at a given time, time which is modeled by the state of some *locks*. We naturally grant our language with two constructions allowing to open or close a *lock*. Namely, in our example, where $`x$ is a special actor standing for "anyone":

- $talk : \{\emptyset \Rightarrow `x\}$
- $jury_i : \{\emptyset \Rightarrow Jury_i;\ \emptyset \Rightarrow Chairman\}$
- $average : \{\emptyset \Rightarrow Chairman; \{Finished\} \Rightarrow `x\}$

Therefore for instance, trying to assign the result of a computation depending on the variable *average* to a variable which would be public, *i.e.* attached to the *policy* $\{\emptyset \Rightarrow `x\}$, will be statically rejected by the type system, unless the *lock* $Finished$ is open.

2.2 FlowLock Security

As stated through the motivating example, we introduce two new sorts of entities, *actors* and *locks*. We then define a *clause*, and therefore a *policy*, as:

Definition 1 *A clause is a formula $\Sigma \Rightarrow \alpha$ where Σ is a set of* locks *required to be open for the* actor *α to be granted access to the data ruled by this policy.*

A policy *is defined as a set of clauses.*

Thanks to this *policy* mechanism, the *declassify* construction can be easily modeled by giving to the variable h a *policy* such as $\{\{Declassified_h\} \Rightarrow `x\}$. Then the construction **declassify** h **in** c would be encoded as:

$$\textbf{open}\ Declassified_h;\ c;\ \textbf{close}\ Declassified_h;$$

But this system is quite obviously much richer.

Those tools being set, *FlockLock* models a perfect attacker, meaning with an infinite memory. Therefore, the intuitive idea is to define successively the following properties of an attacker:

- its capability, given by a set of *locks*. This capability allows it to see some of the outputs, depending on their policies;
- its knowledge is therefore the set of initial complete memory states from which ones the program execution would actually produce the observed outputs;
- and finally the property we require: an attacker whose capability includes the current *lockstate* at the time an output rises should learn nothing new thanks to this observation, meaning the set defining its knowledge shouldn't be reduced.

Consequently, typing the system from an initially empty *lockstate*, we won't allow any leaks, until a *lock* would be manually opened, and will restrain it to its minimum.

A different approach, although equivalent, is exposed by Askarov and Sabelfeld [5] as the idea of *gradual release*. It modeles the intuitive feeling that the uncertainty on the initial memory must be reduced by steps, when a declassification occurs, and only at these points.

Despite this particular subject won't be discuss here, the relations between programs protected relatively to different attaquants seem to be a promising area of research. Notably, Askarov and Chong [6] have recently presented a system in which one the security relatively to a perfect attacker is consequence of the one relatively to an attacker with memory of any size, but bounded.

2.3 Type System

Beyond its elegance and expressiveness, and therefore its practical convenience, an other *FlowLocks*'s asset is its sound type system. We make the arguable choice not to reproduce it there, but only to expose the main underlying reflexions, its understanding not being necessary for our purpose, but we can't help inviting the curious reader to seek for details in the already mentionned thesis.

For expressions, the idea is to simply follow the intuition we tried to develop so far. The information we are interested in is the so-called *read effect*, meaning a safe approximation of the degree of privacy the expression contains. It is therefore quite naturally its *policy* for a variable. For a composed expression, we simply take, thanks to the lattice structure, the join of the sub-expressions' *read effect*.

The statement judgment of c has the following form: $\Sigma \vdash c \rightsquigarrow p, \Sigma'$, where:

- Σ is the set of *locks* assumed to be open before execution of c.
- p is a *policy* referred as the *write effect* of c. It is the union of all variables whose content might be changed when executing the command.
- Σ' is a safe approximation of the set of *locks* open after the execution of the command.

Let's illustrate it through two examples.

$$\frac{\vdash e : r \qquad r(\Sigma) \sqsubseteq pol(x)}{\Sigma \vdash x := e \rightsquigarrow pol(x), \Sigma}$$

We get the *read effect* of the expression, and check that it is, in the current *lock* context, lower than the *policy* of x, *i.e.* "less confidential" that what someone allowed to read x should be able to know.

$$\frac{\vdash e : r \qquad \Sigma \vdash c_i \rightsquigarrow \omega_i, \Sigma_i \qquad r(\Sigma) \sqsubseteq \omega_1 \sqcap \omega_2}{\Sigma \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \rightsquigarrow \omega_1 \sqcap \omega_2, \Sigma_1 \cap \Sigma_2}$$

The conditionnal rule illustrates a lot of crucial ideas. The *read effect* of the guard illustrates the privacy of information we could infer simply by observing which path has been taken. Therefore, someone not allowed to know the value of e can't be allowed to see any affectation made in the conditionnal. That's the very purpose of the *write effect*: we make sure that no affectation in neither branch, through the meet, will be made to a variable "more public" than is the guard. Finally, we return only the *locks* that we are sure to be open no matter which branch has been taken as we want a safe approximation.

3 ParaLocks

Going back to our motivating example from section 2, we can't be satisfied with our $jury_i$ couple of actors. Indeed, not only is it dirty, but what if we don't know at run time how many judges there'll be? This question invites us to introduce the concept of role.

This new concept is kind of an abstraction of the *actor* one. Actually, it is quite natural to consider that the *policies* allowing someone to access to an information are more likely to be based on something like his job or his rank than his identity itself.

Note that if *roles* abstract *actors*, the lattest can hardly simply get rid of since depending on the context, we can both need to reason about status or individual. This is for instance illustrated if we enforce the fact that the grade given by a member of the jury is not visible by any other one.

Thinking about it through the *FlowLocks*'s formalism, we want to be able to state that a variable is accessible to an *actor a iif a* is a member of a role R, therefore we want a lock stating *"R(a) is open iif a is a member of R"*. But then to state the corresponding *policy*, we need to be able to quantify over actors in order to write $\forall x, R(x) \Rightarrow x$.

Thus, *ParaLocks* mainly extends *FlowLocks* with two new features:

- *Parameterised Locks: locks* can now be parameterised by *actors*, modelling the role concept.
- *Actor Polymorphism:* we allow quantification over all *actors*.

This way, we can combine fine grained *policies* with role-based ones.

Once again, having to aim at my actual work and lacking of place, we won't detail the formalism of *ParaLocks* in this document. But it is highly interesting to note that the few additions, despite seeming to fit fluently in the previous layout, actually imply consequent refactoring of the whole modelisation of a *safe program.* Indeed, this notably comes from the fact that *actors* and *locks* now have runtime representations. We therefore have to know if we are dealing with concrete entities, and a consequence is that they now also must have *policies*, since having a runtime representation implies to hold some *flow-sensitive information.*

A last interesting enhancement proposed by Broberg is offering the possibility for *clauses* to have *locks* on their right-hand side, and in particular to be recursive. The main interest is that it allows one to model easily properties on *locks*. For example, stating that a relation between two actors, modeled by a *lock* $R(a, b)$, is transitive can be written as:

$$\forall x, y, z R(x, y) \wedge R(y, z) \Rightarrow R(x, z)$$

Such special *clauses* are grouped as a *global policy* which modificates the lattice of *policies*.

This last feature raises issues of computability, as well as has consequences over the *policy* inference mechanism that we will cover in the following section.

4 Paragon

ParaLocks is a powerful language dedicated to the specification of *information flow policies*, but it's not a practical programming language. *Paragon*[2] fills the gap by integrating the *ParaLocks* paradigm to *Java,* in a user-friendly way.

The compiler, written in *Haskell*, aims at dealing with every feature of the language so that it could be adopted in industrial context. We won't cover in the present document the challenges, ranging from syntactic choices to computability issues, that the transition from *ParaLocks* to *Paragon* rises, but the curious reader is once more invited to refer to Broberg's thesis. [7]

More pragmatically, the following sections cover the jobs in which ones I actually did take an active part, namely *policy inference* and *null-pointer analysis.*

4.1 Policy inference

Aiming at actual industrial use, Paragon has to add as little extra work for the programmer as possible. In particular, we easily imagine that if we want to explicitely specify the *policy* for sensible variables, either containing classified information, either being public, we also would like to be authorized not to think of what should be the *policy* of some unremarkable local

[2] http://www.cse.chalmers.se/~d00nibro/paragon/

variables. All we want is to be granted by the system that there exists a *policy* which would respects every other ones.

More precisely, the type checking being modular, we consider the body of a single method. As we explained, an assigment such as $x \leftarrow y$ is correctly typed iif $\bar{y} \sqsubseteq \bar{x}$, with the convention that $\bar{x}$ designs the *policy* of x. Therefore, assuming that we now allow unspecified policies, the set of assigments in the body of a method will generate a set of constraints having the form :

$$\{p \sqsubseteq_{\mathcal{G},\mathcal{L}} q\}$$

Where $\mathcal{G}$ is the *global policy* and $\mathcal{L}$ the set of open *locks* standing at the assigment point. Each policy implied potentially containing unspecified ones, our goal is to infer from these constraints if there exists a set of solutions.

This feature was actually not handled in the implementation of Paragon when I began my internship. But the structure of lattice possessed by our set of *policies* seemed suitable for a constrained-based resolution, being even in the definite problem configuration covered by Rehof and Mogensen [10]. It actually appeared that a simpler solution could be used in our situation.

4.1.1 Without global policies

As a first step, I dealt without considering neither the *global policies* nor the *locks*. One can notice that our case actually possesses an additionnal property, being transitive, *i.e.*:

Property 1

$$(p \sqsubseteq q \wedge q \sqsubseteq r) \Rightarrow p \sqsubseteq r$$

But more accurately, we have:

Property 2 *if p and q don't depend on X,*

$$\exists X, (p \sqsubseteq X \wedge r(X) \sqsubseteq q) \Leftrightarrow r[X \leftarrow p] \sqsubseteq q$$

The next important point is to consider the form of the right-hand side of the constraints. Despite a significant case on which one we will come back later, it must be an atomic *policy*. Indeed, it has to be the *policy* associated with the left-hand side of an assigment, *i.e.* a variable, and therefore is either a concrete *policy* or reduced to an unspecified one.

On the other side of the constraint, the important point is that no meets can be encountered.

It explains that the condition in property 2 is suitable since if p depends on X, then it is more restrictive than X and the constraints can be dismissed, being trivial. And if q depends on X, then it resumes to X and the constraint is trivial.

All these remarks lead to the quite straightforward algorithm 1. The instruction $cstrs[X \leftarrow p]$ returns the set of constraints in which one X has been substituted by p, but also the unmodified ones since X is likely to be on the right-hand side of more than one inequality. Consequently, the function *check* first gets rid of the constraints still containing any variable so that it only have to check the transitive closure.

The implementation itself went quite easily once I had learnt the Haskell basis, since it worked with very few interaction with the compiler.

4.1.2 Open locks and global policies

We now take into consideration both the global *policies* and the open *locks*. We actually can't anymore directly go on with the substitutions as we did in the algorithm 1 since all the constraints are not meant to be satisfied on the same lattice. Indeed, opening a lock may kind of merge two points of our lattice into the same one, and closing it will separate them. So property 2 is no longer true. But we have an additionnal property which saves the situation:

Algorithm 1 Algorithm for policy inference without global policies and locks issues

Input: $cstrs$
$cstrsVar \leftarrow \{p \sqsubseteq q \text{ s.t. } q \text{ is a variable}\}$
for each variable X **do**
  for each p s.t. $p \sqsubseteq X$ is in $cstrs$ **do**
    $cstrs \leftarrow cstrs[X \leftarrow p]$
  end for
end for
check($cstrs$)

Property 3

$$(p \sqcup q \sqsubseteq_{\mathcal{G},\mathcal{L}} r) \Leftrightarrow (p \sqsubseteq_{\mathcal{G},\mathcal{L}} r \wedge q \sqsubseteq_{\mathcal{G},\mathcal{L}} r)$$

Thanks to property 3, we can assume that we have atomic constraints also on the left-hand side of our inequalities. We then get back a new version of the transitive property:

Property 4 *if p and q are atomic* policies,

$$\exists X, (p \sqsubseteq_{\mathcal{G},\mathcal{L}} X \wedge X \sqsubseteq_{\mathcal{G},\mathcal{L}'} q) \Leftrightarrow p \sqsubseteq_{\mathcal{G},\mathcal{L}\cup\mathcal{L}'} q$$

The resulting algorithm is very similar to the first one, but is internally based on a more evolved *lower than* relationship, depending on the *lock* state and taking the *global policy* into account.

Algorithm 2 Algorithm for policy inference without global policies and locks issues

Input: $cstrs$
$cstrs \leftarrow$split($cstrs$)
$cstrsVar \leftarrow \{p \sqsubseteq_{\mathcal{G},\mathcal{L}} q \text{ s.t. } q \text{ is a variable}\}$
for each variable X **do**
  for each p s.t. $p \sqsubseteq_{\mathcal{G},\mathcal{L}} X$ is in $cstrs$ **do**
    $cstrs \leftarrow cstrs[X \leftarrow p]$
  end for
end for
check($cstrs$)

The primitive *split*, following the property 3, flattens our input into elementary constraints, namely inequalities between atomic *policies*. The substitution mechanism now works according to property 4.

The implementation has been a little bit harder since it has been coupled on a refactoring of the data structure used to represent *policies*, and therefore forced me to enter deeper in the compiler.

4.1.3 Filling the remaining gap ?

So far, we assumed that we did not have any meet *policies* on the left-hand side of a constraint, and therefore could split them the way we did, neither join ones on the right-hand side. This necessity comes from the fact that both the following implications are true, but the reciprocals are not.

Property 5

$$p \sqcap q \sqsubseteq_{\mathcal{G},\mathcal{L}} r \Leftarrow p \sqsubseteq_{\mathcal{G},\mathcal{L}} r \wedge q \sqsubseteq_{\mathcal{G},\mathcal{L}} r$$

$$p \sqsubseteq_{\mathcal{G},\mathcal{L}} q \sqcup r \Leftarrow p \sqsubseteq_{\mathcal{G},\mathcal{L}} q \wedge p \sqsubseteq_{\mathcal{G},\mathcal{L}} r$$

But actually, both these bad cases can occur.

Especially, since *Paragon* technically allows the programmer, when specifying the writing *policy* of a method, to express it in function of the possibly non-specified ones of the arguments, and in particular as a meet, the second case might rise. It's actually a quite pathological case that we could simply forbid, but the same issue can also come from an imbrication of two exceptions, the resulting *write effect*, *i.e.* the flow-sensitive *policy* context in which one we are, beeing a meet.

Expecting to find a way to be complete, we tried to take another point of view. We see a *policy* as the set of *actors* it allows and our set of constraints becomes a problem of set theory inclusions. Such a problem can be solved in a complete way, as described by Omedeo, Ferro and Schwartz in [8].

But if it solves our remaining issues in the easy context we treated in subsection 4.1.1, it does not seem to be compatible with *locks*. Consequently, at this point, the *Paragon* compiler simply asks to the programmer to manually specify the problematic *policies* if it falls in such a case.

4.2 Null-pointer analysis

4.2.1 Leaks through exception

Paragon inherits from *ParaLocks* of the choice of being *termination insensitive*, meaning we allow the last step of computation to reveal the existence of an observable output. This concession is pretty much a practical necessity, due to the non-computability of the halting problem. Consequently, we could assume to pay no attention to exceptions, considering them as potential premature terminations. But the *catch* feature makes it being flow control based, and therefore needs to be prevented.

This aspect was already handled for most of exceptions as *Java*'s methods have to declare if they might raise such one. We can therefore enforce in *Paragon* the systematic checking of them and work consequently in the adequate context. [7]

But some exceptions actually can't be dealt with this way. Indeed, subclasses of RuntimeException are unchecked. This concerns in particular the infamous NullPointerException and therefore highlights the need for a *null-pointer analysis*. The point of it is infering through a static analysis the existence of a run raising such an exception. It is known to be non-computable in a complete and correct form, and we will therefore naturally need a correct one for our purpose.

If the domain is quite covered in academical researches, our context is incompatible with most of them. Especially many assume, and in particular do the first papers I've investigated [12], to be working on *Java Bytecode*. Indeed, this is clever since it gets rid of many ultimately redundant features of the language, and evolves far less through the years. But sadely, this is not an option for us.

The second constraint is the fact that we need to perform this analysis in parallel with our current one, so that we can use the detection of a potential null-pointer error as do when crossing a **Raise** statement. This dismissed the second promising paper I considered [9], since this one would first analyse the whole program to generate a set of constraints, and then solve it.

4.2.2 Our analysis

I based my work on the method suggested by Summers and Müller [1]. The idea is to add two additionnal informations to types :

- a non-null type in {*MaybeNull, NotNull*} indicating whether we actually know that the expression can't be *null*;
- an initialisation modifier, taking its value in {*Commited, Free, Unclassified*}, assuring whether an object is deeply initialized when accessed.

The object initialisation is dealt with in various way in the literature. The idea is that the runtime system initialises all fields of a new object with zero-equivalent values. So even fields declared as non-null start out being null. To adress this issue, we need a way to track the point at which one we know the initialisation to be done.

The point is depicted as the *commitment point*, and an object must have, before it, a *free* modifier so that an access to a field, even *NotNull*, will end with a *MaybeNull* type. Once the commitment point is reach, the type system enforce the object to remain deeply initialized.

The non-null type is more intuitive. The idea is to consider any field being initially *MaybeNull*, unless explicitely specified. We then track affectations, as well as non-nullity tests in guards of conditionnals or loop. We also allow the programmer to explicitely defer its non-nullity checking in the body of a method, or before its call, as arguments are assumed *MaybeNull* by default, unless specified otherwise.

Unexpectedly, this work has been really hard for me, and took me a lot of time. The theory is quite not trivial to handle in details, and switching from the toy language described to *Java* implied some work, but my greatest difficulty has been practical. This time, interventions at nearly every single level of the compiler was necessary to carry the extra informations needed, and make the desired checkings. It therefore took me a huge amount of time between the moment I decided which paper to use and studied it, and my first step in its implementation.

It surely has not been some waste time, as it taught me lots of stuff about Haskell, Paragon itself, compilation and simply how to handle a huge project, but it led to the deceiving situation being that I did not finished the job. The initialisation modifiers handling is missing. Nevertheless, most of the type system is implemented, being successfully tested on various examples. And maybe more important, the core necessary enhancements of the data structures are in, so that I have great hope that it can easily be completed by my successor.

Conclusion

Niklas Broberg has developped a new platform for information theory in the presence of declassification [7], namely *ParaLocks*, a rich policy specification language accompanied by a a sound type system. He has incorporated it to an actual programming language, having chosen for this purpose *Java*, mainly in order to compare itself with the only major existing alternative, *Jif* [11, 4], resulting in the so-called *Paragon*. My internship took place in this context, a practical compiler, written in *Haskell*, about to be complete, missing only very few features.

It has therefore been a great opportunity to be introduced to the *information flow-control* field, and discover in a deeper way some of its aspects. I furthermore learned the *Haskell* language in order to be able to work on the compiler.

These requirements being fullfiled, I worked on the *policy inference* issue. I implemented a correct method, based on transitive closure computation, able to infer most of unspecified policies through a method body. We then investigated an idea, taking a set-theoretic point of view, which led to a dead-end.

The second part of my work has been dedicated to the *null-pointer analysis*. Information leaks through uncaught exception being undesirable, the *Java* RunTimeException need special

intention. I partially implemented a correct, and therefore non-complete, such analysis. It is mainly based on the enrichment of the type system with both a non-nullity tracker and an initialisation one.

Please don't hesitate to give a try to the *Paragon* compiler, and through it to my fractional contribution, and program in a secure way!

Acknowledgment

I would like to thanks all the people who allowed this intership not only to happen, but to be both a pleasure and an highly rewarding experience.

In order to do so, I'm extremely grateful to David Sands for having so warmly welcomed me in the security department.

I'm also grateful to each of its member, since no matter how intimidated I've been, they all have been incredibly kind and friendly. It's an impressively and beautifully cosmopolitan team. It has made this journey for me as rich in lessons from a social point of view than a scientific one.

And my last words here go to the two people who gave me a lot of their time and kindness to answer to every one of my questions. So, Bart van Delft and Niklas Broberg, thank you both!

References

[1] Peter Müller Alexander J. Summers. Freedom before commitment: a lightweight type system for object initialisation. *OOPSLA*, pages 1013–1032, 2011.

[2] Andrew C. Myers Andrei Sabelfeld. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[3] David Sands Andrei Sabelfeld. Declassification: Dimensions and principles. *Journal of Computer Security*, 2009.

[4] Barbara Liskov Andrew C. Myers. A decentralized model for information flow control. *Symposium on Operating Systems Principles*, 1997.

[5] Andrei Sabelfeld Aslan Askarov. Gradual release: Unifying declassification, encryption and key realease policies. *IEEE Symposium on Security and Privacy*, 2007.

[6] Stephen Chong Aslan Askarov. Learning is change in knowledge: Knowledge-based security for dynamic policies. *CSF*, 2012.

[7] Niklas Broberg. *Practical, Flexible Programming with Information Flow Control.* PhD thesis, Chalmers University of Technology, 2011.

[8] Omodeo E. G. Ferro A. and Schwartz J. T. Decision procedures for elementary sublanguages of set theory. i. multi-level syllogistic and some extensions., 1980.

[9] Laurent Hubert, Thomas Jensen, and David Pichardie. Semantic foundations and inference of non-null annotations. In *Proceedings of the 10th IFIP WG 6.1 international conference on Formal Methods for Open Object-Based Distributed Systems*, FMOODS '08, pages 132–149, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] Torben Ae. Mogensen Jakob Rehof. Tractable constraints in finite semilattices, 1996.

[11] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, 1999.

[12] Fausto Spoto. Precise null-pointer analysis. *Softw. Syst. Model.*, 10(2):219–252, May 2011.