Jifclipse: Development Tools for Security-Typed Languages

Boniface Hicks Dave King Patrick McDaniel
Penn State
{phicks,dhking,mcdaniel}@cse.psu.edu

Abstract

Security-typed languages such as Jif require the programmer to label variables with information flow security policies as part of application development. The compiler then flags errors wherever information leaks may occur. Resolving these information leaks is a critical task in security-typed language application development. Unfortunately, because information flows can be quite subtle, simple error messages tend to be insufficient for finding and resolving the source of information leaks; more sophisticated development tools are needed for this task. To this end we provide a set of principles to guide the development of such tools. Furthermore, we implement a subset of these principles in an integrated development environment (IDE) for Jif, called Jifclipse, which is built on the Eclipse extensible development platform. Our plug-in provides a Jif programmer with additional tools to view hidden information generated by a Jif compilation, to suggest fixes for errors, and to get more specific information behind an error message. Better development tools are essential for making security-typed application development practical; Jifclipse is a first step in this process.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments—Programmer workbench; D.4.6 [*Operating Systems*]: Security and Protection—information flow controls

General Terms Security, Languages

Keywords security-typed languages, Jif, developer tools, Eclipse

1. Introduction

Security-typed languages such as Jif require the programmer to reason about security during the initial development process. Software that produces the correct results is no longer sufficient; it is also necessary to work out the information flows that will occur in the application. This requires the programmer to determine in advance 1) the entities (*principals*) that will handle data in the system, 2) the security policies (*labels*) that should govern individual program variables and 3) the interaction of variables throughout the application (*information flows*).

To make these ideas concrete, let us consider two examples. In a secure email system, 1) the principals could include all those who will send and receive emails, along with a policy specifying whom each principal trusts to receive his data. 2) Newly composed emails should be labeled with the principal of their sender and 3) they should only flow (i.e., be relabeled to) the principals of their receivers. The challenging subtlety comes in handling the network communication necessary for sending and receiving email. Because the Internet is public by nature, labeled emails should not be released unless they are first protected—by encryption, for example. Once the programmer specifies this in the application, the compiler can automatically ensure that email data is never leaked to principals who are not explicitly trusted by the sender.

As another example (we will use this as a running example throughout the paper), consider an application which simulates the administration of exams to students[1]. 1) The principals in this setting consist of students who can take the exam, along with the administrator who provides the exam. One policy is that none of the principals trust each other (to prevent all leakage between students). 2) The exam questions are labeled with the examiner's principal. The students' answers to the questions are labeled with both the student's own principal and the examiner's principal, since they must contain information from the student as well as information about the exam questions. 3) This application has some interesting information flow requirements. The simplest requirement is that students should not share information, so one student's answers should not be visible to another student. Of course, the application should also not leak the correct exam answers to any of the students. This leads to an interesting information flow, however, because ultimately, each student's answers must be compared against the correct answers and the result must be released to the student. This leaks information about the exam answers to the students and should only be allowed in a specific circumstance—after all students have turned in their exams.

While principal determination must occur largely in the design phase (and can involve complicated interactions between PKIs, system labels and other data classifications), labeling data and resolving information flows take place iteratively during the coding phase. Code that the programmer thought would respect secure data flows will often fail to compile and so the programmer must relabel variables as the compiler flags potential information leaks.

To complicate matters, the information flow violations can be quite subtle and are not easily expressed by simple error messages. Even though the message may accurately describe an information leak, it may not clearly illuminate the source of the error. This is only further complicated when the language handles not only confidentiality policies, but also integrity policies, not only explicit data flows but also implicit control flows. Inferred and default labels, while important to reducing the programmer's burden in providing labels, also cloud the process of tracking down label conflicts. It is our position that error resolution is one of the key issues currently facing security-typed language development: without better tools

PLAS'07 June 14, 2007, San Diego, California, USA.

[1] This example is derived from a class exercise used by Andrei Sabelfeld at Chalmers: `http://www.cs.chalmers.se/Cs/Grundutb/Kurser/lbs/JifLab2006/`

for finding and fixing errors, using these languages will remain impractical.

In this paper, we contribute a set of principles for designing tools to support application development in security-typed languages, based on our [13, 12, 14] and others' [1, 28] experience. The main focus of our work is on providing better tools which enable the programmer to understand and fix errors. To that end, we present Jifclipse, a plug-in for the Eclipse framework[2]. Our plug-in provides a Jif programmer with additional tools to view hidden information generated by a compilation, suggest fixes for errors, and get more specific information behind an error message.

The remainder of the paper is structured as follows. In Section 2, we describe some of the problems in security-typed application development in general and in Jif specifically. As we discuss these problems, we also propose some design principles for development environments for security-typed languages. In Section 3, we give some background on the Jif compiler and walk through an example of how to resolve information leaks in Jif (without the help of Jifclipse). In Section 4, we present Jifclipse and show how it fulfills some of our design principles. We then revisit our example of tracking down information leaks, this time with the help of Jifclipse. In Section 5, we discuss related work. Finally, we conclude in Section 6 and outline future work.

2. New Programming Challenges

Security-typed languages offer a new, unique set of programming challenges, because they require programmers to reason in advance about the information flows that should be allowed in a program. Mature language development tools should aid the programmer in these challenges as much as possible. In this section, we propose some principles that will be helpful guidelines for designing development tools for security-typed languages. We group these principles into three main areas, 1) principal determination, 2) labeling data and 3) identifying and resolving information flow conflicts. In the rest of the paper, we will focus primarily on the third area, because we believe it finds its most natural solution in an IDE and this is what we have provided for Jif with Jifclipse. We describe the other principles for the sake of completeness and as a target for future work. For each principle, we offer a key, motivating observation.

2.1 Principal determination

A critical and challenging problem in building secure systems is determining who the principals are in the system and how their information flows can interact. Clearly these principals need to be connected with some identity outside of the application since the sources and sinks of information flows are going to be outside of the application (through various I/O channels). In the first systems that have been built with security-typed languages, principals have been connected with a PKI for secure email [12], with access controls lists (ACLs) provided by the operating system attached to files [20], with labels from a Mandatory Access Control (MAC) system [14] and with login identities for two players in a game communicating through a serialized file channel [1].

This experience in building secure systems exposes the fact that *runtime principals* [29] are an essential feature of practical security-typed languages. Runtime principals are principals that have an identity during program execution and can be checked dynamically to uncover their policy (what information flows they allow). Since information flows originate outside an application and (if they are useful) end up outside an application, there must be some correspondence between principals outside the application (entities in the system) and principals within the application (controlled by the language). Furthermore, the relationships between these principals, including how they allow information to flow between them in ordinary and extraordinary (such as declassification) circumstances, is a critical part of a program's specification.

One solution to meeting this need is a high level policy system which supports the declaration of principals, the specification of their information flow policies and the way they relate to principals outside the application [13, 14]. This policy system should include tools which can implement these high-level policy constructs in a way that integrates smoothly with security-typed languages (e.g. generating code in a security-typed language). Ideally, such a policy system should also allow for safe dynamic updates of policy [15, 26] to support dynamic environments.

Observation: The determination of principals in an application and the establishment of information flow policies between those principals is essential for building applications with security-typed languages.

Principle 1 (*Principal principle*): Development tools for security-typed languages should provide a means for specifying potentially dynamic principals, information flow policies between those principals and the relationships between those principals and entities in the operating system. These tools should also generate or link with the secure infrastructure that implements such policies. This might be done in the spirit of UML specification [11], compile-able policy [13], or some other high-level design tools.

2.2 Labeling data

After determining what principals can interact in a given application, the data handled by the application must be tagged with the appropriate principal. How to tag data depends partially on the program's specification. For example, a field that will contain an email server password should have restricted visibility by annotating it with the password owner's principal, while that user's name may be a public string. The exam questions should be secret (owned by the Examiner), while each student's answers should be labeled with both the student's principal and the Examiner's principal. These labeling decisions depend on the semantics of the application and must be provided by the programmer.

On the other hand, there are many other variables in a program that could be inferred by the compiler or IDE based on these "seed" labels provided by the programmer. For example, the local variables in a method will have labels that often follow from the parameters on the method. Consider an iterator used to loop through a secret array or a stream used to display exam questions to a student. Likewise, some fields in an object will depend on other fields. The label on the total number of correct answers a student achieves on an exam flows directly from the "seed" labels on the student's answer, the exam questions, and the exam answers. Inferring labels not only reduces the burden on the programmer with regard to providing labels, but is also more likely to reduce the number of information flow conflicts the programmer causes through mislabeling intermediate variables.

Observation: Some variables must be labeled by the programmer, because their labels are inherently part of the application's specification, but many more variables can be labeled by automatic inference or default.

Principle 2 (*Label inference principle*): Development tools for security-typed languages should provide as much label inference as possible, so that the application developer only needs to specify those labels which are inherently part of the program's specification (and thus impossible to infer with confidence).

[2] Eclipse is an extensible platform which provides facilities for developing IDEs, among other things (`www.eclipse.org`).

2.3 Fixing information flows

As a programmer specifies the labels on data, conflicts inevitably arise due to illegal information flows. The number of these conflicts may be reduced when fewer labels are specified and label inference is more robust, but it is unlikely they will be eliminated entirely. In fact, it is in this that security-typed languages provide a great benefit. They serve to expose interactions between data of different security levels even when these interactions may be subtle and far-reaching in the code, identifying potential information leaks.

Observation: A primary benefit of security-typed languages is that they expose information leaks in programs.

Principle 3 (*Information flow resolution principle*): Security-typed language tools should provide detailed information to programmers to help them identify and repair information flow violations.

Information flow leaks may have various causes, and helping the programmer to identify the cause and fix the broken information flow may be dependent on the language-specific implementation used for resolving security-type annotations. For example, using constraints to track information flows, allowing for declassification, and tracking implicit control flows, all complicate the process of resolving information flow conflicts in programs. For this reason, we break this principle down into a few language-specific sub-principles.

2.3.1 Constraints, declassification, and implicit flows

It is common to use constraints to abstract type system requirements, and existing security-typed languages have used this approach to model information flows based on security-type annotations. As previous experience (in security-typed languages as well as in other languages with complex type systems, such as Haskell or Standard ML) has shown, however, isolating an error in a constraint system can be challenging. It is not always obvious what the initial error which caused the constraints to fail to be satisfiable is. Since this is critical to the contribution of security-typed languages, some development tools are necessary to aid the programmer in this regard.

To start, good heuristics should determine which constraint is to blame. This helps the programmer focus in on the source of an illegal information flow. It would also help to suggest how to repair the information leak. Finally, when all else fails, the programmer should be able to explore all the constraints involved in the information leak and determine the source for himself.

Observation: Constraints complicate error reporting and make it difficult to determine what the source of the information flow may be.

Principle 3a (*Constraint resolution principle*): Security-typed language tools that use constraints to track information flows should provide detailed information to programmers to help them view all constraints in a program, especially focusing on constraints involved in a bad flow.

A related problem regards some situations when exceptional information flows, known as declassifications, may be needed. Recalling the example of the exam room, a declassification is needed to expose each student's results, but only after the exam is complete. This data exposure violates the information flow policy on the exam's questions and a security-typed language compiler will rightly flag this as an error. In the proper circumstances, however, it should be possible for a programmer to override such a violation—such as after all exams have been turned in. These declassification points must be chosen carefully, because they open up information leaks. An inference engine may be able to assist the programmer by suggesting places where introducing declassification can alleviate information flow conflicts. A more advanced inference engine may even be able to quantify the amount of information that could be leaked by adding such declassifications.

Observation: Declassifications are necessary in realistic applications, but because they open information leaks, they should be placed carefully.

Principle 3b (*Declassification principle*): A security-typed language that uses declassification should provide facilities for suggesting declassifications that can resolve information flow violations, while still allowing the programmer to make the ultimate decision based on the program's security specification.

Security-typed languages can track implicit control flows in addition to explicit, data flows. This is commonly done by introducing a PC label which is set to the level of any implicit flows at a given point in the code. This label then induces a constraint on any explicit flows at that point in the code. For example, the PC label is raised by a high-secure guard on a conditional and any explicit flows in the body of the conditional are tainted by the PC label. Consider the following code from the Exam Room application, which has an implicit flow from `qi` to `totalAlice`. When evaluating this code, the compiler raises the PC label by joining it with the label on `qi` and then joins the PC label to the right-hand side of the assignment, ultimately requiring that `totalAlice` be at least as secret as `qi`.

```
if (qi != null && qi.isCorrect(x))
  totalAlice = totalAlice + 1;
```

Preventing implicit flows makes for stronger security, but causes more troubles for the programmer. The trouble comes in the fact that the label from the guard taints the whole body of the conditional; this effect is not easy to keep track of. The problem is exacerbated when exceptions come into play, because an implicit flow can be caused by premature termination and taints the code following the place where an exception can be thrown.

Observation: Keeping track of implicit flows by hand can be complex.

Principle 3c (*Implicit flow principle*): If a security-typed language supports implicit flows, development tools should provide a mechanism to reveal the PC label at a given point in the code.

2.3.2 Relaxing security checking

Allowing for variable degrees of security checking can assist in more rapid prototyping of applications (and may improve a programmer's sanity when using security-typed languages). Furthermore, some dataflow analyses can help to determine when the compiler is being overly conservative and ruling out information flows that should be allowed. In this principle, we gather together three different areas that could accelerate a programmer's development time.

Alleviating overly conservative checking Because exceptions can cause information flows, Jif requires all exceptions to either be caught or explicitly thrown. This means that every use of an object reference must be wrapped in a **try/catch** block, because it could potentially throw a `NullPointerException`. The same holds for array accesses, arithmetic divisions and all other runtime exceptions that could remain unhandled in Java. Handling all these potential control flows becomes arduous and clutters the code. Some dataflow analyses can be helpful for determining when such exceptions are impossible (and thus not requiring them to be handled), and some facilities to automatically insert handling code would be beneficial for the programmer.

Integrating with non-security-typed languages Integration with a non-security-typed language can be helpful in cases when low-level subroutines must be called. Also, to facilitate the gradual migration to fully security-typed applications, the ability to reuse existing, but unannotated library code (for encryption, for example) is extremely important. Compiler facilities that enable the method headers of these modules to be annotated without requiring complete annotation and checking of the bodies is a great help. Jif already provides a mechanism for incorporating existing Java libraries with minimal annotation (the annotations are called Jif signatures), but an even deeper integration of Java and Jif development environments for this purpose would be helpful.

Weakening information flow checking Finally, some control over the amount of security enforced by the compiler can be a useful switch for more general usage of security-typed languages. Although weakening the information flow checking also weakens the guarantees produced by the compiler (and compiled applications should be signed with a manifest indicating the guarantees they provide), it can be effective for more rapid prototyping and may be sufficient in certain situations. To be specific, security-type annotations can be used to enforce confidentiality and integrity. Furthermore, as previously discussed, some additional constraints can be added to prevent implicit flows along with the usual explicit flows. Although not currently implemented in any security-typed language, it is possible to imagine that other covert channels such as timing and termination flows could also be prevented. Finally, various models of declassification have been proposed in the literature [24]; the programmer may wish to use different ones for different applications.

Observation: All security guarantees are not needed for every application. Application developers may benefit from starting with weaker security checking and adding guarantees incrementally as they progress through the development process.

Principle 4 (*Rapid prototyping principle*): Security-typed language development tools should aid the programmer in rapid prototyping of applications by allowing the temporary weakening of security enforced by the compiler and also enabling the integration of existing, non-security-typed libraries into security-typed applications. Furthermore, it is beneficial to include dataflow analyses which can prevent the compiler from being overly conservative.

3. Resolving Information Flow Conflicts in Jif

For the remainder of the paper, we focus our attention on the third set of principles, which pertain to fixing information flows. We give some background on Jif and the Jif compiler. We then look at a particular example and the challenges faced by the programmer in using the Jif compiler to find and resolve an information flow. This example will be revisited in Section 4 after we describe the advantages offered by Jifclipse, our main contribution.

3.1 The Jif compiler

The Jif compiler is an extension module for the Polyglot extensible compiler framework [21]. Jif implements a superset of the JFlow language [19], but has recently been expanded with support for, among other things, integrity policies and meet labels [4, 20]. The phases of the compiler which are important for our purposes are the *type-checking pass* and the *label-checking pass*.

The type-checking pass verifies that the program passes Java type-safety; e.g, there are no unsafe assignments or improperly-invoked methods. Jif takes a further step in forcing runtime exceptions (such as null pointer and class cast exceptions), to be either caught or explicitly thrown, as these may cause implicit flows.

The label-checking pass is where Jif verifies that a program satisfies certain security requirements. The compiler generates constraints that force these security requirements to hold throughout each section of the program. For example, for an assignment statement to be secure, the security level of the variable being assigned to must be more restrictive than the security level of the data being stored to it. The compiler generates the constraint $\texttt{L1} \leq \texttt{L2}$, where `L1` is the label on the data being assigned and `L2` is the label on the variable being assigned to. After generating all of the constraints for a method[3], the constraint solver verifies that these constraints are satisfied and thus that the desired security properties hold.

We describe here some Jif syntax which will be used later in this section. Jif uses the decentralized label model (DLM) [18] to specify security labels. Labels in Jif consist of confidentiality policies and integrity policies. In the DLM, the join of two labels `L1` and `L2` is written {`L1` ; `L2`}. If `P` is a principal, then the label with confidentiality policy {`P:`} indicates data that can only flow to other data labeled `L` where $\{\texttt{P:}\} \leq \texttt{L}$, while the label with integrity policy {`P!:`} indicates data that can only flow to data labeled with `L` where $\{\texttt{P!:}\} \leq \texttt{L}$. Principals can be arranged in a hierarchy which induces an ordering on labels, such that $\{\texttt{P:}\} \leq \{\texttt{Q:}\}$ iff $\texttt{P} \leq \texttt{Q}$. The least confidential (i.e. public) principal is `_` and the least confidential label is {`_:_`}, while the most confidential principal is $*$. The DLM also allows for reader and writer lists on labels, but they are not needed to understand the examples in this paper.

Jif allows classes to be parameterized with a principal or a label to provide additional static checking on mutable fields. For example, a `Student` object containing secret fields annotated with {`Alice:`} is instantiated as `Student[Alice]`. Another important note for our examples is that arrays of parameterized classes will have two pairs of square braces and two labels. The first pair of square braces signifies the a security annotation parameter and the second indicates the type is an array. Consider an array of `Question`'s, `Question[Examiner]{Examiner:}[]{_:_}`. The structure of this array is public (i.e. anyone can see the maximum length of the array), but each element is parameterized with `Examiner` and the existence of each array element (i.e., that it is not **null**) is considered {`Examiner:`}-level information.

Jif also uses a special label variable `caller_pc` to represent the level of the program counter at the time that a method was called. It is implicitly joined to local variables in the method. This can be a frequent source of errors for beginning Jif programmers [28].

3.2 Fixing information flows

In this section, we describe a series of errors as viewed by a programmer of a Jif class. We use the Exam Room application as described previously; it has the benefit of being a simple example of security-typed language programming while also not being tailor-made to show off the features of Jifclipse.

The primary objects that interact in this example are an `Exam` object and a `Student` object (in general, there could be many student objects, but we consider only one for this example). Furthermore, there are two principals, `Alice` and `Examiner`, which are used to restrict the information flows on the object fields. The challenge of the security-typed application programmer is to properly label the fields and method headers for the classes.

The method `Exam.runExam`, displayed in Figure 1, asks each of the students the questions in the exam, tallies the number of answers that they answered correctly, and then reports that value back to the student. The programmer has labeled the lower bound of side effects for `runExam` as being at level {`Examiner:`} (this lower bound is called the *begin-label*), thinking that only the examiner should observe an effect of this method. He has labeled the total

[3] A compiler option allows checking to be done per class instead.

```
1  public void runExam{Examiner:}() where caller(Examiner){
2      IStudent[Alice] alice = this.alice;
3      Question[{Examiner:}]{Examiner:}[]{Examiner:} pool =
4          questionPool();
5      int{Examiner:} nq = pool != null ? pool.length : 0;
6      int{Alice: } totalAlice = 0;
7
8      for (int{Examiner:} i = 0; i < nq; i++) {
9          Question[{Examiner:}]{Examiner:} qi = null;
10         try {
11                 qi = pool != null ? pool[i] : null;
12         } catch (ArrayIndexOutOfBoundsException e) {}
13
14         String qtxt = qi != null ? qi.getText() : null;
15         String{Examiner:}[]{Examiner:} qvars =
16                         qi != null ?
17                         qi.getVariants() :
18                         null;
19
20         int x = alice != null ? alice.getAnswer(qtxt, qvars) : -1;
21         if (qi != null && qi.isCorrect(x))
22             totalAlice++;
23     }
24     if (alice != null) alice.passResult(totalAlice);
25 }
```

Figure 1. Jif code for method `Exam.runExam`.

number of questions answered correctly by Alice, `totalAlice` as having {`Alice:`}-level security. The error that he receives when first compiling the code is:

```
src/Exam.jif:45: The actual argument is more
restrictive than the formal argument.
    if (alice != null) alice.passResult(totalAlice);
                                        ^--------^
```

Checking the method `Student.passResult`, the programmer confirms that this method, when instantiated for `Alice`, is labeled such that it may take an integer at level `Alice`:

public void `passResult{Alice:}(int{Alice:} x)`

This error, however, is telling him that on line 24, he cannot pass `totalAlice`, which he has labeled at level {`Alice:`} (line 6), to a function that takes exactly that security level. Using the `-explain` flag to generate a more detailed error message is a little better, but can still be confusing. This message can be seen in Figure 2.

From this information, the programmer can determine that the left-hand side of the constraint (the label on the actual argument) contains an {`Examiner:`} policy which is not on the right-hand side (in the label on the formal argument). This may come as a surprise, because the programmer annotated the actual argument, `totalAlice`, with the label {`Alice:`} (line 6), but the error message asserts that the label of the first argument passed to the function is *not* {`Alice:`}. Instead, the label on `totalAlice` is some join of {`Alice:`} with the `caller_pc` along with some other innocuous confidentiality and integrity policies.

At this point, the programmer must figure out that the mismatch is caused because the method `runExam` could be called from a site with a program counter that is not less restrictive than the begin-label on `passResult` (i.e. the callee). Seeing this, the programmer realizes that `runExam` may have side effects visible to principals other than `Examiner`, and so adds a meet-label to the method header (line 1) indicating that its side effects may be visible either to `Examiner` or to `Alice`.

However, a second error occurs after the program is recompiled. In this case, the compiler highlights the point where `totalAlice` is incremented after Alice gets a question correct, indicating that the program counter is too restrictive at that point. The relevant code (lines 21–22) is

```
src/Exam.jif:45: Unsatisfiable constraint:
actual_arg_1 <= formal_arg_1
{caller_pc; Alice: ; _!: _; _: _; *!: } <= {Alice: }
in environment
[{this} <= {caller_pc}]

Label Descriptions
------------------
- actual_arg_1 = the label of the 1st actual argument
- actual_arg_1 = {caller_pc; Alice: ; _!: _; _: _; *!: }
- formal_arg_1 = the upper bound of the formal argument x
- formal_arg_1 = {Alice: }
- caller_pc = The pc at the call site of this method
(bounded above by {Examiner: })
- this = label of the special variable "this"

The label of the actual argument, actual_arg_1, is
more restrictive than the label of the formal argument,
formal_arg_1.

    if (alice != null) alice.passResult(totalAlice);
                                        ^--------^
```

Figure 2. A detailed error message provided by the Jif compiler for information leak.

```
// qi is a Question owned by the Examiner
...
if (qi != null && qi.isCorrect(x))
        totalAlice++;
```

This violation occurs because `totalAlice` gets incremented for each question Alice answers correctly. This implicitly leaks information about the questions (i.e. {`Examiner:`}-level information, as labeled on line 9) into `totalAlice`. The actual policy on the variable after this assignment is a join of {`Alice:`} and {`Examiner:`}. To reflect this, the programmer needs to change the declared label on `totalAlice` (line 6). Once he modifies `totalAlice` to being at level {`Examiner::Alice:`}, this error goes away, but one final error occurs.

Now the formal parameter of `passResult` is not restrictive enough, because it expects only {`Alice:`}-level information. There are two ways to handle this. The programmer may decide that the implicit flow from the question into `totalAlice` is insignificant (it only says how many she got right in total) and add a declassifier. Alternatively, stricter confidentiality could be maintained by restricting the formal parameter on `passResult` to reflect that it carries {`Alice::Examiner:`}-level information now. After either of these changes is made, the program successfully compiles.

It is obvious that writing security-typed language code without a precise understanding of the data and the interfaces involved can cause major difficulties. In the worst case, it is sometimes a challenge just to understand an error. Sometimes, changing the declared label on a variable or on a method header can fix a conflict. Other times, a declassifier is the right solution for a problem. Better tools can aid the programmer in finding and repairing errors in their code.

4. Jifclipse

In this section, we describe the implementation of features in Jifclipse which correspond to the problems and design principles that we have identified in previous sections. Jifclipse has several additional features which we do not highlight in this section; our primary contribution is in providing tools for programmers to understand and resolve security errors in their applications.

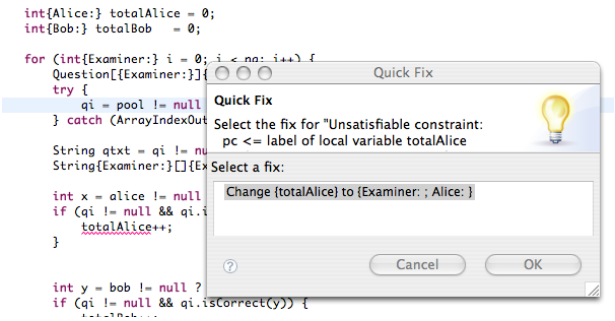

Figure 3. Jifclipse suggests changing the declared label on a variable.

4.1 Fixing information flows

Quick fixes From our experience, many label errors in Jif result from the programmer incorrectly declaring the label on a variable. This is especially true for the declared labels on the upper bound of method arguments and the begin label. If a Jif program fails to compile, then we examine each of the variables involved with the broken constraint and attempt to determine, using our inference framework, what its label should be. Though this is a fairly simple approach, it gives good results in practice and can be augmented with more advanced techniques in the future.

We implement this using the "Quick Fix" framework in Eclipse. Figure 3 shows Jifclipse suggesting that the programmer change the explicit label on a variable declaration. In the code above, the programmer has incorrectly declared `totalAlice`, a variable keeping track of how many answers the student Alice has gotten correct, as having {`Alice:`} level security, when in fact this value needs to be modified using information that the Examiner has. Jifclipse suggests raising the variable `totalAlice` to its correct security level.

Jifclipse can also suggest modifications to a method's begin label or labels on method arguments. To accomplish this, however, we needed to improve the algorithm for the constraint solver. Our solver allows us to infer optimal upper bounds for procedure arguments. We briefly describe the differences between the solver provided by the Jif compiler and the solver we implemented for Jifclipse in the following section.

Label inference The Jif compiler already contains a simpler label inference engine for determining the lower bounds of label variables [23]; programmers do not need to explicitly give the types of local variables when writing their programs (though giving them makes code more explicit and easy to understand). Label variables are also used as a shortcut to represent longer types in equations; for example, the local variable declaration **int**{`Alice:; Bob:`} `i = 5` produces an equality constraint {`i`} == {`Alice:; Bob:`}. Afterwards, when the label on the variable `i` is used in other constraints, the label variable {`i`} is used instead of the explicitly declared label on the originally declared variable.

The Jif solver initially assigns all variables to $\top$, the highest security level, and then lowers these bounds down until the constraint equations are satisfied. If a solution exists, this method is guaranteed to produce one (in fact, the least restrictive solution).

However, in order to infer what the optimal upper bounds for procedure arguments are (where "optimal" means the highest possible upper bound, such that the method checks), we need to implement a more powerful solver that can fix both upper bounds and lower bounds for a label variable. For this, we use bounds consistency solving algorithm [17]. We briefly summarize the implementation of this algorithm here. All variables begin with upper bound $\top$ and lower bound $\bot$; we then adjust their bounds from the label constraints as generated by the Jif compiler. For example, the equation {`i`} <= {`Alice:`} fixes the upper bound of the label variable {`i`} to be the label {`Alice:`}. For the equation {`i`} <= {`j`}, we adjust the upper bound of {`i`} to be the meet of its current upper bound and the upper bound of {`j`} (since the label of {`i`} cannot be above the label of {`j`}).

In the event that applying the constraints to the label variables results in an undetermined domain (one where not every label variable has the same label as its upper and lower bound), then the solver restricts one of the variables to a specific label and applies the constraints again. A label variable is restricted to a label depending on whether it is a "lower" or "upper" variable label. For example, local variables, return labels, return value labels, and exception labels are all "lower" labels. In contrast, method arguments and PC bounds are "upper" labels. When a label variable is chosen for restriction, it is set to its current lower or upper bound depending on whether it is a lower or upper label, respectively.

The algorithm continues in this way until the solution is completely specified. If during this it discovers an unsatisfiable equation, we backtrack and choose a different restriction. While in the worst case, this is exponential in the number of variables, we only call the inference solver with very few variables at a time while generating a fix for a broken label constraint. We plan to use the label solver to greater effect in the future; for example, adding a feature to infer all of the labels at once for a method or class.

4.1.1 Constraint resolution principle

The most important service provided by security-typed languages is discovering information flows that violate the application's information flow policy. This advantage is severely reduced, however, if it is not possible for the programmer to leverage the compiler to track down and fix these information flows. This principle can be in tension with our inference principle. While the inference principle seeks to hide details from the programmer, in some cases, it is necessary also to reveal more information to the programmer in order to fix errors. Jif has already laid the foundations for this, by generating and storing detailed information about the constraints needed for type checking. We have expanded the information contained in these constraints and exposed the information to the programmer through Eclipse Views. Since the Views need not be opened or examined, we have addressed the tension between hiding information from the programmer in general and revealing more information only when an information leak needs to be fixed.

Outline View If a Jif programmer does not give explicit security labels for data in his code, a default value is used. (The exception is local variables; as mentioned above, these are already inferred by the compiler.) Method headers have three important labels associated with them; the return value label, the return label, and the begin label. If not explicitly annotated, the return value and return labels default to the $\bot$ label, while the begin label defaults to the $\top$ label. Method arguments, if unlabeled, default to the most restrictive security level.

In order to reveal these implicit labelings, we have implemented an outline viewer for Jif. In Java, the outline viewer shows the imports in a file, the classes defined within a file, and the methods defined within a class. We implement this functionality as well as explicitly showing the programmer what the implicit labels are on their fields and methods. Figure 4 shows the outline view of a given class, with the labels for method headers made explicit.

Constraints View By default, the Jif label checker checks sets of constraints one method at a time; the average method can have anywhere between five and a hundred constraints which need to be satisfied. How these constraints are generated and how they interact with one another is sometimes subtle: when a Jif program fails to

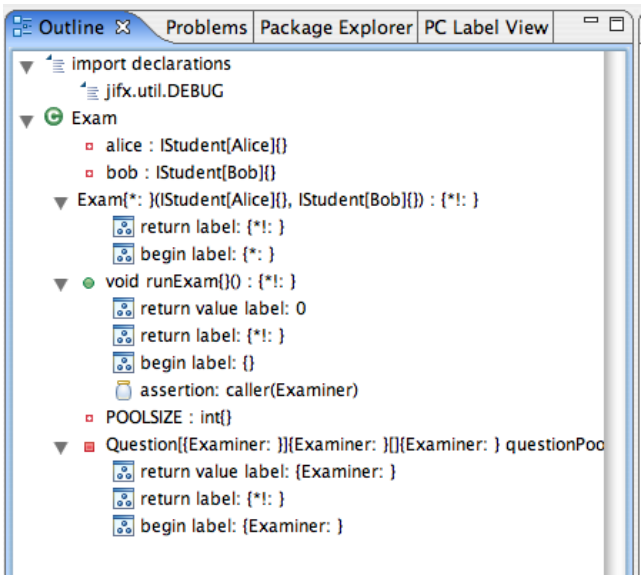

Figure 4. The Outline View, similar to the outline view in Eclipse, quickly provides information about the methods in a file.

compile, it only reports one unsatisfiable constraint, which may or may not be the original cause of the error.

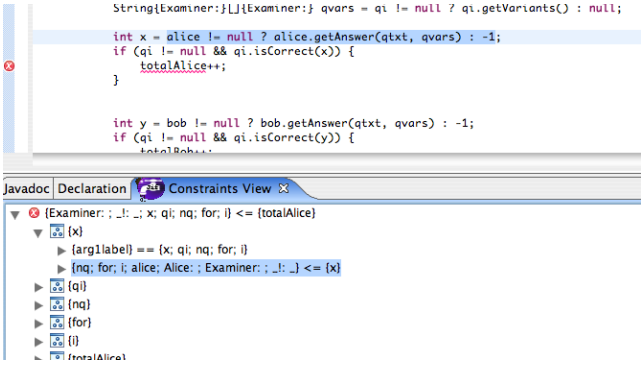

Figure 5. The programmer can use the Constraints View to view all of the constraints generated by a line of code and navigate through related constraints.

The Constraints View exposes all of the constraints associated with a piece of code and all of the constraints that share variables with that constraint. In the Figure, the top half is the program code, the bottom half is a window containing the Constraint View. By using a hot key, the constraints for the cursor position in the program code are revealed in the view (the view in Figure 5 was generated by pressing the hot key while the cursor was on the error `totalAlice++`). The subtrees from a constraint are all the constraint variables that occur in the constraint. The subtrees on a constraint variable are all the constraints that involve that variable. (Clearly, these trees may involve cycles, but the on-demand generation of subtrees avoids infinite loops.) In practice, it is only valuable to traverse a few levels of these constraint trees to uncover relevant constraint relationships.

The Constraints View can also reveal some additional information about constraint variables by double-clicking them, namely, the label inferred for a particular constraint variable and some additional explanation provided by the compiler for how the constraint got that label. Finally, by highlighting a constraint, Eclipse automatically highlights the expression in the program code that caused that constraint (in the Figure, the highlighted constraint was generated by the initialization of the variable `x`, which is highlighted).

4.1.2 Declassifier inference principle

Sometimes a security constraint is unsatisfiable because of a fundamental conflict between security requirements. In this case, Jif allows adding a declassifier to explicitly lower the sensitivity of a security level. To aid the programmer in this, we provide a feature to automatically declassify an expression and the current program counter, if necessary.

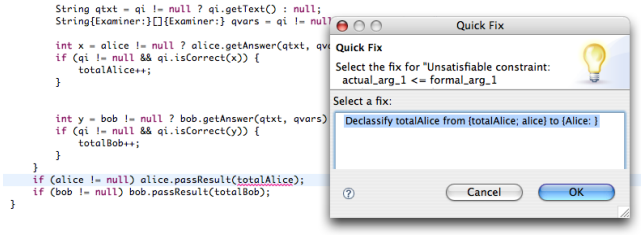

Figure 6. Jifclipse suggests adding a declassifier to resolve a problem.

In Figure 6, the programmer has written code that he believes will tell Alice her result, stored in `totalAlice` and declared to have security label {`Alice:`; `Examiner:`}. However, the Student interface (as presently written), when instantiated for Alice, expects that the result told to the student is owned wholly by the student, having label {`Alice:`} in this case. This causes a security violation, as we cannot lower the security label on `totalAlice` when passing it to a function. Jifclipse suggests adding an explicit declassifier to lower `totalAlice` to the expected security level of {`Alice:`}. Other fixes suggested by the IDE may explicitly declassify the program counter as well or instead of the data, as necessary.

Due to some limitations with using Polyglot as a backend for an Eclipse plugin, Jifclipse currently only supports adding declassifiers for a few classes of errors. A topic of future work is modifying the backend so that automatically adding declassifiers can be done more easily. Additionally, the programmer could benefit from some extra analysis to ensure that a minimal number of declassifiers can be added to fix an error.

4.1.3 Implicit flow principle

The program counter, which taints the labels of expressions with the label of information required to evaluate that expression in the code, serves to disallow implicit flows in a security-typed language. However, as mentioned previously, the value of the program counter changes at different points in the code in ways that may not be obvious to the programmer.

The PC Label View shows the different values that the program counter takes on at different points in the code. By walking down the PC Label View, the corresponding program code that caused the PC label to change is highlighted. If the programmer double-clicks on a label in the View, the system reports exactly why the program counter has been changed to this new value.

Figure 7 shows the PC Label View. In this example, the programmer has mislabeled `totalAlice` as having {`Alice:`} level secrecy as in a previous example. The PC Label View for the if statement indicates that the program counter is {`Examiner:`; `_!:_`; `x`; `qi`; `nq`; **`for`**; `i`}. Prior to this it was {`nq`; **`for`**; `i`}. Double-clicking on the label, we can see that the change was based on the use of the Examiner's `qi` object.

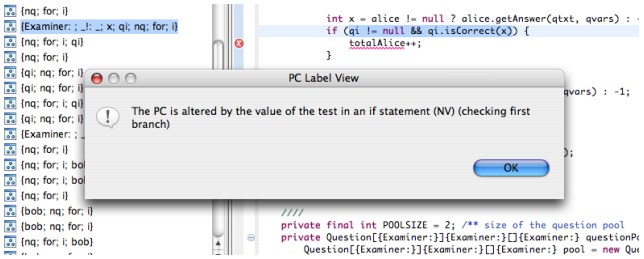

Figure 7. The PC Label View highlights which lines of the code modify the program counter.

Currently, this View is too reflective of the underlying compiler framework, presenting the PC label at each change. One solution would be to integrate it more tightly with the Constraints View, since the PC Label can be seen as just another constraint. Another possibility is integrating the information on the PC label into the program code more tightly, perhaps using different-colored highlights for different label changes, and perhaps allowing hovertexts to show the PC label at different points in the code.

4.2 Rapid prototyping principle

We have augmented the compiler to limit the security checking it requires for successful compilation in various cases. In particular, the addition of integrity in Jif 3.0 broke existing applications which compiled under Jif 2.0. By adding a `−nointegrity` flag to the compiler, we limit the power of its security checking, but increase backward compatibility and enable programmers to build applications more rapidly. A similar switch was added for confidentiality to handle applications for which only integrity is important. We leave it to future work to provide hooks in the compiler for allowing the programmer to specify even more fine-grained security specifications.

4.3 Ease-of-Use modifications

To supplement Jif's null pointer analysis for handling runtime exceptions, we also provide a quick fix to automatically insert **try/catch** blocks for catching exceptions. Handling every runtime exception is an exceedingly time-consuming part of converting Java code to Jif code. This quick fix enables the programmer to prototype an application more rapidly.

We have also added some simple, helpful configuration tools for setting up applications to leverage Java code and Jif signatures. These project setup and configuration tools enable programmers to move more rapidly from startup to development.

4.4 Exam room revisited

Here we review how our tools help in resolving the information flows presented in the example in Section 3.2. With the first error, the programmer discovers that there are in fact three problematic constraints (two are related to the begin-label and the third is regarding the error that we handle last). The first broken constraint is very clear; {`Examiner:`} <= {`Alice:`} does not hold. Double-clicking on this constraint reveals the reason it was added:

```
The PC before evaluating the call must be less
restrictive than the callee's begin label.
```

Furthermore, it explains that the begin label of the method's side-effects is {`Alice:`} and the PC of the call site is {`Examiner:`}. This leads to the conclusion that the begin-level on `runExam` should be lowered with a meet-label.

For the next error, the explain message gives the broken constraint:

{`Examiner:` ; `_!:_` ; `Alice:` ; `_!:_` ; `_:_` ; `∗!:`} <= {`Alice:`}

The Constraint View helps the programmer by keeping track of what variables those labels originated in and allows the programmer to view the label on each variable as a subtree of this constraint.

{`Examiner:` ; `_!:_` ; `x`; `qi`; `nq`; **for**; `i`} <= {`Alice:`}

More importantly, however, Jifclipse suggests the correct solution through a quick fix: to relabel `totalAlice` as {`Examiner:`; `Alice:`}.

The final error is that `totalAlice` is too restrictively labeled to be an argument to `passResult`. A look at the Constraints View reveals all the variables involved in this constraint, including explicit flows from `totalAlice` and implicit flows from checking whether `alice` is **null**. Jifclipse proposes a declassification as a quick fix. The programmer may decide that he wants to maintain the stricter confidentiality and change the method labels on `passResult`. This is ultimately a decision that must be made based on the application specification. The quick fix suggestion at least gives the programmer an intuition about what will fix the constraint.

5. Related Work

As programming languages add more power to their static analysis, the error messages returned by the compiler become more difficult to understand. This makes it harder for the programmer, as the list of things that the programmer needs to keep in mind while debugging type error increase: language syntax, typing rules, subtyping rules, polymorphism, checked exceptions, and on. The security-typing rules of Jif, when added to the already complex Java language, make it quite difficult for beginning Jif programmers to parse some of the more subtle error messages produced by the compiler.

Improving error messages There is a large body of work on improving compilation error messages in order to have them make more sense for the programmer, much of it from the functional language community about reporting error messages in the presence of type polymorphism. In a recent survey [10], Heeren identifies two different directions this work has taken. One strand, beginning with Wand [30] seeks to trace *everything* which is involved with an error. The other strand which began in the same conference with Johnson and Walz [16] has sought to pinpoint the most likely mistake which led to an error. These two strands have a common theme with our work, which seeks to accomplish both tasks: giving all the information related to an error *and* suggesting the most likely cause along with a way to fix it.

We highlight a few papers relevant to displaying hidden compiler information to the programmer. Wand [30] modifies ML's unification algorithm to tag type variables with the expressions that caused them to become bound. When a type error is reported, the inference algorithm provides several possible explanations, leaving the programmer to decide which one caused it. Beaven et al. [2] extend this by developing an explanation-based approach for an ML compiler, allowing the programmer to ask both `Why` questions about why an expression has been given a particular type and `How` questions about how a type variable has come to have been bound. Chitil [3] develops a type system which provides *compositional* explanations, i.e. explanations that contain unique sub-explanations. Reading the explanation of a type error involves walking through the graph of explanations built during type-checking. Two other graphical engines for aiding in understanding constraints are given by Foster [7] for the type-based static analysis tool, CQual, and by Flanagan, et al. [6] for ML.

A primary difference between ML and Jif is that, in ML, once the cause of a type error has been determined, the course of action is generally clear. For example, the ML code (`fun f −> f 5 25`) (`fun (x,y) −> x + y`) raises a type error because `f`,

which expects a pair, is passed into a context where a function that takes non-paired arguments is expected. Tupling the arguments to `f`, resulting in the code `(fun f -> f (5,25)) (fun (x,y) -> x + y)`, does not involve system-wide corectness concerns.

On the other hand, resolving errors in security-typed languages can be complicated. Adding a declassifier to code has system-wide security concerns. Existing work has suggested a number of programming patterns for resolving errors in Jif [1].

Deng and Smith [5] present an approach to improving security-type error messages in a simple language with arrays. Their approach is necessarily language-specific; rather than using constraints to express label dependencies between program code, their solver keeps track of all of the variables which influenced the type of an expression and their histories. For more complicated languages, we believe that developer tools, rather than just improved error messages, are essential for effective program development.

Flow Caml Flow Caml is an implementation of information-flow ML [22] that extends a subset of the Caml language with security typing. Along with security typing, the language features ML polymorphism, mutable state, exceptions, datatypes, and pattern matching. Its main advantage is its inference engine [25], which automatically assigns a security typing to programs. Its policy language is simpler than the decentralized label model used by Jif for its label syntax, featuring only joins of principals. It currently contains no features for runtime principals or declassification.

An interesting feature that Flow Caml implements is a graphical representation of types. Quicksort is a commonly used sorting algorithm which uses a filter function to quickly sort in $O(n \log n)$ expected time. Its type in Flow Caml is given below.

```
val qsort : ('a int, 'b) list ->
            ('a int, 'b) list
            with 'a < 'b
```

A version of Quicksort that leaks information about the list being sorted to standard output (in Flow Caml, this is represented by the principal `!stdout`) has, as we should expect, a different security type.

```
val qsort_leaker :
       ('a int, 'b) list -{!stdout ||}->
       ('a int, 'c) list
       with 'a, 'b < 'c
       and  'b < !stdout
       and  'a < !stdout
```

The above function type explicitly reveals that there is a side effect viewable by the stdout channel, and that this function can only operate on lists which allow information flows to that channel.

Figure 8 displays the graphical representation of the types of the two different quicksort functions. *Green* variables are contravariant (inputs), while *Red* variables are covariant (outputs). A dashed arrow represents a subtyping constraint on the security levels in that line. Note that the graphical type makes it clear that the leaking version gives information to standard output.

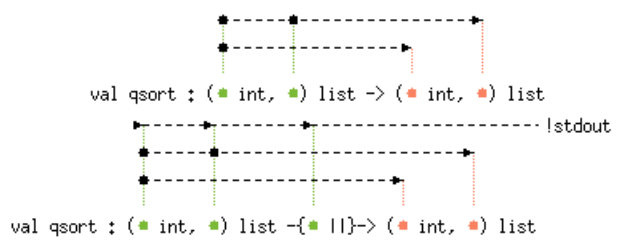

Figure 8. Graphical information in Flow Caml

More complicated code examples (especially higher-order ones) result in more complicated graphs. For example, the type of the recursive higher-order `map` function has six variables and four label constraints, while the graph displays three type subtyping constraints and three label subtyping constraints, pointing both right and left. This speaks to the inherent complexity of the security information in this function.

Eclipse development environments Eclipse provides a platform for building integrated development environments (IDEs) for programming languages. For example, the Eclipse JDT [27] provides features for Java programmers to automatically add new classes, change variable names, refactor code into its own method. Additionally, the programmer can see compilation errors while he types, which allows him to quickly correct common mistakes.

There are a number of language-specific Eclipse plug-ins available, though the level of additional functionality that they provide varies. Guyon et al. [9] integrate TOM, an extension of Java incorporating algebraic types and pattern-matching expressions, with Eclipse and give commentary on the details of the process. More generally, the SAFARI project [8] aims to provide a way to easily build Eclipse IDEs for new languages. For Jifclipse, we have primarily focused on adding functionality to the Jif compiler to support easier resolution of error messages. Future work involves integration of Jifclipse with releases of SAFARI.

An early version of Jifclipse was used during development of previous work [12]. This bare-bones plug-in automatically compiled files, highlighed special keywords, and displayed error messages generated by the Jif compiler with a pointer for the correct line. It contained all of the basics required to do application development, as well as an early version of the outline view. All of the other features described in Section 4 are new to this paper and should provide programmers with powerful new tools in writing secure applications.

6. Conclusion and Future Work

Security-typed languages introduce new programming challenges, because they open up a new dimension to system construction, requiring programmers to reason in advance about information flows. These challenges include 1) principal determination, 2) data labeling and 3) resolving information flow conflicts. Principal determination and labeling data cannot be solved entirely through development tools; they are inherent to an application's specification. On the other hand tasks such as information flow conflict resolution and a substantial amount of data labeling, can be aided through visualization and inference tools, respectively.

In this work, we have sought to capture the unique programming challenges faced by the security-typed language programmer and we have set forth principles for developing tools to support the programmer in these challenges. We have shown how these principles can be met through a cooperation of the compiler and a development environment by implementing an Eclipse IDE for Jif. One conclusion we draw is that while the compiler is most helpful in assisting with data labeling (through inference and default labels), a development environment is essential for simplifying the task of identifying and resolving information flow conflicts.

There are a few areas of future work. There is a clear need to build better tools for integrating principals with entities outside the application and for improving label inference. For example, is there some way to facilitate the connection of application principals with a PKI or operating system identities or even an application's own user database? This might be specified as a compile-able policy [13] or designed as a UML specification [11], for example.

A second category of future work involves using Eclipse as a platform for future projects in security-typed languages. As

security-typed languages involve both code and policy, tools developed for them necessitate greater programmer interaction than "normal" languages such as Java. Eclipse allows for a rich amount of interaction with the programmer; for example, it could be used as a platform to guide automatic program transformation from Java into Jif, similar in flavor to an algorithmic type debugger [3].

A third area of future work is improving Jifclipse. Though the plugin currently assists Jif programmers in many ways, it can be improved further, in terms of its label suggestions, declassification points, and more helpful error messages. There is a vast amount of work in improving compiler errors for functional languages such as ML; there may be security-specific techniques that can better identify the source of Jif errors.

Finally, some basic features of Eclipse IDEs which allow for simpler traversal of large code bases, for example, would be most welcome. We expect that these will be provided as part of the SAFARI project [8], at which point we hope to integrate them with the security-typed language-specific tools. At that point it will be clearer how to specialize these standard IDE features for use with a security-typed language.

Acknowledgments

We are indebted to Kiyan Ahmadizadeh for his assistance in implementing an early version of Jifclipse. We are also grateful to Mike Hicks, Stephen Tse and the anonymous reviewers for their valuable editorial comments on this paper. We thank Steve Chong for his endless patience with and prompt responses to our questions about Jif. This research was supported in part by NSF grant CCF-0524132, "Flexible, Decentralized Information-flow Control for Dynamic Environments" and in part by Motorola through the Software Engineering Research Consortium (SERC).

References

[1] ASKAROV, A., AND SABELFELD, A. Secure implementation of cryptographic protocols: A case study of mutual distrust. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS '05)* (Milan, Italy, September 2005), LNCS, Springer-Verlag.

[2] BEAVEN, M., AND STANSIFER, R. Explaining type errors in polymorphic languages. *ACM Lett. Program. Lang. Syst. 2*, 1-4 (1993), 17–30.

[3] CHITIL, O. Compositional explanation of types and algorithmic debugging of type errors. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2001), ACM Press, pp. 193–204.

[4] CHONG, S., AND MYERS, A. C. Decentralized robustness. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW)* (Venice, July 2006). to appear.

[5] DENG, Z., AND SMITH, G. Type inference and informative error reporting for secure information flow. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference* (New York, NY, USA, 2006), ACM Press, pp. 543–548.

[6] FLANAGAN, C., FLATT, M., KRISHNAMURTHI, S., WEIRICH, S., AND FELLEISEN, M. Catching bugs in the web of program invariants. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation* (New York, NY, USA, 1996), ACM Press, pp. 23–32.

[7] FOSTER, J. S., JOHNSON, R., KODUMAL, J., AND AIKEN, A. Flow-insensitive type qualifiers. *ACM Trans. Program. Lang. Syst. (TOPLAS) 28*, 6 (2006), 1035–1087.

[8] FUHRER, R. M., CHARLES, P., SUTTON JR., S. M., AND LAFFRA, C. SAFARI: A meta-tooling platform for creating language-specific IDEs. EclipseCon 2007 Talk.

[9] GUYON, J., MOREAU, P.-E., AND REILLES, A. An integrated development environment for pattern matching programming. In *2nd Eclipse Technology eXchange Workshop - eTX'2004* (2004).

[10] HEEREN, B. J. *Top Quality Type Error Messages*. PhD thesis, Departement Informatica, Universiteit Utrecht, 2005.

[11] HELDAL, R., AND HULTIN, F. Bridging model-based and language-based security. In *ESORICS* (2003), E. Snekkenes and D. Gollmann, Eds., vol. 2808 of *Lecture Notes in Computer Science*, Springer, pp. 235–252.

[12] HICKS, B., AHMADIZADEH, K., AND MCDANIEL, P. Understanding practical application development in security-typed languages. In *22st Annual Computer Security Applications Conference (ACSAC)* (Miami, Fl, December 2006).

[13] HICKS, B., KING, D., MCDANIEL, P., AND HICKS, M. Trusted declassification: High-level policy for a security-typed language. In *Proceedings of the 1st ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '06)* (Ottawa, Canada, June 10 2006), ACM Press.

[14] HICKS, B., RUEDA, S., JAEGER, T., AND MCDANIEL, P. From trusted to secure: Building and executing applications that enforce system security. In *Proceedings of the USENIX Annual Technical Conference* (Santa Clara, CA, USA, June 2007). To appear.

[15] HICKS, M., TSE, S., HICKS, B., AND ZDANCEWIC, S. Dynamic updating of information-flow policies. In *Proceedings of the Foundations of Computer Security Workshop (FCS '05)* (March 2005).

[16] JOHNSON, G. F., AND WALZ, J. A. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (New York, NY, USA, 1986), ACM Press, pp. 44–57.

[17] MARRIOT, K., AND STUCKEY, P. J. *Programming with Constraints: an Introduction*. MIT Press, 1998.

[18] MYERS, AND LISKOV. Complete, safe information flow with decentralized labels. In *RSP: 19th IEEE Computer Society Symposium on Research in Security and Privacy* (1998).

[19] MYERS, A. C. JFlow: Practical mostly-static information flow control. In *POPL* (January 1999), pp. 228–241.

[20] MYERS, A. C., NYSTROM, N., ZHENG, L., AND ZDANCEWIC, S. Jif: Java + information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[21] NYSTROM, N., CLARKSON, M., AND MYERS, A. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction* (2003).

[22] POTTIER, F., AND SIMONET, V. Information flow inference for ml. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2002), ACM Press, pp. 319–330.

[23] REHOF, J., AND MOGENSEN, T. A. Tractable constraints in finite semilattices. *Science of Computer Programming 35*, 2–3 (1999), 191–221.

[24] SABELFELD, A., AND SANDS, D. Dimensions and principles of declassification. In *Proceedings of the IEEE Computer Security Foundations Workshop* (Aix-en-Provence, France, June 2005).

[25] SIMONET, V. Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS'03)* (Beijing, China, Nov. 2003), A. Ohori, Ed., vol. 2895 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 283–302. ©Springer-Verlag.

[26] SWAMY, N., HICKS, M., TSE, S., AND ZDANCEWIC, S. Managing policy updates in security-typed languages. In *CSFW '06: Proceedings of the 19th IEEE Computer Security Foundations Workshop* (2006).

[27] THE ECLIPSE FOUNDATION. Eclipse java development tools (JDT) subproject. http://www.eclipse.org/jdt/.

[28] TSE, S., AND WASHBURN, G. Cryptographic programming in Jif. Project report for CIS-670, Spring 2003, Feb. 11 2004. `http://www.cis.upenn.edu/~stse/bank/main.pdf`.

[29] TSE, S., AND ZDANCEWIC, S. A design for a security-typed language with certificate-based declassification. In *Proceedings of the 10th European Symposium on Programming, ESOP 2005* (2005), Lecture Notes in Computer Science.

[30] WAND, M. Finding the source of type errors. In *POPL '86: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (New York, NY, USA, 1986), ACM Press, pp. 38–43.