APL Assignment
Information Flow in Jif

Niall Napier

March 18, 2008

1 Abstract

While programming languages provide support for encapsulation and basic access control there are rarely methods in place to mediate how data flows around an application. This is essential to ensuring that a system does not suffer from information leaks. The Decentralised Label Model and Java Information Flow framework approach this problem by providing a means of statically checking the flow of information around an application, based on policies associated with variables. Jif presents an efficient model for data flow control with a minimal runtime overhead. However, it lacks support for threading, an important programming feature, and a lack of accessible documentation make it unlikely to appeal to the mainstream programmer.

2 Introduction

Nowadays, almost all organisations rely on computer systems to store and retrieve personal information about their customers. Whether it be an address, account or even just a name this information must be kept secure under the terms of the Data Protection Act (1998). Under the Act, organisations storing personal information must ensure that "Appropriate technical and organisational measures shall be taken against unauthorised or unlawful processing of personal data and against accidental loss or destruction of, or damage to, personal data." [1] Consequently, they must provide a means to mediate access to any sensitive information contained in their systems. Conversely, the systems themselves must be protected from users. Just as customer information should never be exposed to unauthorised users, nor should information about the system itself. Information leaked about how the system works can provide clues allowing a malicious user to compromise the system.

The way that information moves through a program or system is often referred to as *information flow* and is defined by the variables and methods in the program. Each time a program produces an output there is the possibility that the information it contains includes data that was never supposed to be disclosed. To solve this problem it has been

suggested that programs should implement a permission based model to impose ownership on the variables in a program. This ensures that only the principals authorised to read and write variables can do so.

This report introduces Jif [4], an extension to the Java programming language that allows the programmer to control access to variables in a program. The following sections will detail the problems associated with information flow control, how Jif can be used as a solution and existing alternative solutions to the problem. To illustrate the concepts used in Jif, section 6 contains an example program.

3 Controlling Information Flow

Security covers a wide range of topics and programmers have long considered security implications while developing systems. Information security discusses how principals actions on objects are controlled by policies. A principal is a user, group or role that can affect an object in the system. For example a principal could read/write a value from/to and object. A policy is a set of rules which define what principals can perform what actions on a particular object. For the duration of this report consider Alice and Bob as principals. An example policy may state that Alice can read some value x but Bob cannot. The Java programming language provides an extensive set of security libraries covering domains such as encryption and authentication as well as security policies to limit the access a program has to the system. Using best practices [3] combined with these tools can improve the security of an application, however, it provides no *guarantee* the information will not be leaked from the system.

A leak of information occurs when a program sends output to an unintended recipient. For example, a user could be presented with another user's bank account details. Alternatively the information could be about the system, such as the location of a file reported during an exception. To guarantee that information cannot be leaked during the execution of a program requires that all variable assignments, operations and methods be checked to ensure that the information flow is permitted. This also requires that the system calculate the implicit permissions of any new object to ensure that restricted data is not copied to less-restrictive variables. Mandatory Access Control (MAC) is a mechanism whereby all operations on an object are checked against a central policy before they are executed. Unlike Discretionary Access Control (DAC), the access control model used in most UNIX filesystems, there is no way for the owner of an object to override the security policy. The system maintains a security object for each variable to keep track of each variable's permissions and uses them to compute new security objects for new variables when appropriate [4]. While the MAC model has seen some success in access controls for filesystems, coarse permissions granularity and runtime checking, and therefore significant overheads, have made it unsuitable for use in a programming language.

The Decentralised Label Model (DLM) stores the security policy of each variable as a label and allows principals to control the flow of information through the application [5]. By storing policies using labels the compiler can statically check the information flow

of the program removing the need to perform all checks at runtime. To ensure that the system remains flexible, it is still possible to declare runtime principals.

4 Decentralised Label Model

4.1 Principals

Principals are users, groups or roles that can affect some portion of the system, usually by reading or writing values in the program. Since principals are not limited to users, the provide a flexible means by which to manage access to variables. Jif implements the notion of a principal hierarchy through *acts-for* relationships. Any principal can authorise another to act on its behalf. This allows it to carry out any of the actions the original principal would be authorised to make in a program. Since principals can be roles, it is possible to implement a complex structure whereby an abstract *manager* can act for several employees, and a *person* can act for a manager, fulfilling their role.

To make the language more usable there are also limit principals $\top$ ($*$), which can act for any principal and $\bot$ (_) which can be acted for by any principal.

4.2 Policies

The DLM allows the programmer to specify policies using labels for each variable. This is done after the variable type declaration and is parenthesised by braces. The label is can contain a set of policies and where more than one policy is required, more can be defined by using semicolons to separate them. If there are no policies assigned to a variable it will infer one from its position in the program, though it is also possible to specify a public policy using the empty label {}. There are two main types of policy, one to specify who can read a variable and one to specify who can write to it.

Confidentiality policies describe which principals can *read* a variable and are described using the $\{o \rightarrow r\}$ notation where r is the reader and o is the owner of the policy. The owner of a policy can, implicitly, also read that variable. More readers can be added to the policy using a comma-seperated list, for example $\{o \rightarrow r_1, r_2\}$. This notation has changed with the development of Jif and is also referred to as $\{o : r\}$ and ->. The listings in this report use the -> notation as this is considered the most natural notation for programming.

Integrity policies describe which principals can *write* to a variable and are described using the $\{o \leftarrow w\}$ notation where w is the reader and o is the owner of the policy. Again, the owner of a policy can, implicitly, write to the variable it represents.

This is all very well, except that it requires the application trust all information it contains. Where the application allows data input this is not always appropriate. To solve this the DLM requires that for the policy to be upheld, the owner of the policy must act for the current principal. This models the relationship of trust between the current principal and the data. In the case that the owner does not act for the current principal, the current principal ignores the policy.

The limit principals can also be used to define highly or minimally restrictive policies. For example, the label $\{o \rightarrow *\}$ allows only the top principal to read the variable, and is effectively the same as having no readers at all.

Since labels contain sets of policies, the model allows for basic set operations. These can be used to calculate the intersections and unions of policies and therefore calculate the actual policy based on several distinct ones. This allows the programmer to think more abstractly about how policies affect one another and which is the most or least restrictive policy for a particular value.

To reduce the possibility of an information leak, data can only be transferred to variables or methods with at least as restrictive a policy as the original. For instance, new variables can only add owners or remove readers from policies. Since the transfer of information can only ever become more restrictive it is difficult or impossible for an information leak to occur.

5 Jif: Java Information Flow

While DLM forms the foundation of Jif, it is a technology model in itself. Jif takes the concepts and the language rules form DLM and expands them into a full language extension to Java. This section of the report explains some of the features of the language extension that will be covered in the example.

```
1 int{Alice->Bob; Alice<-Frank} x = 0;
2
3 declassify(x, {Alice->Bob; Alice<-Frank} to
4     {Alice->Bob, Frank; Alice<-Frank});
5
6 endorse(x, {Alice->Bob, Frank; Alice<-Frank} to
7     {Alice->Bob, Frank; Alice<-Frank, Bob});
```

Listing 1: Downgrading information policies.

Section 4.2 detailed how policies restrict information flow, however, this does not make for a usable programming language. Jif allows the security of information to be downgraded by explicitly specifying a new policy label for an expression. The method `declassify(e,` L_1 `to` L_2`)` downgrades the security of the read set by replacing label L_1 with L_2 for expression e. The writer set of L_2 must be at least as restrictive as L_1. To downgrade the writer set, the programmer must instead use `endorse(e,` L_1 `to` L_2`)`. This time, the writer set is relaxed to the policies of L_2, however, the reader set must remain at least as restrictive as L_1. This is illustrated in listing 1 though it is worth noting that Jif 3.1.1 is inconsistent with this notation and instead only uses the expression and the new label in declassification and endorsements. By forcing these downgrades to be performed independently of one another there is never the chance that the programmer can accidentally relax

the policy for the wrong reader or writer set.

```
l = false;
if (h) {
    l = true;
}
```

Listing 2: Implicit flow [2]

So far the report has only discussed explicit assignments information flow, however, there are many ways that information can pass through an application without being explicitly transferred. Listing 2 shows a simple program taken from the Jif reference manual [2] whereby private information stored in variable h could be propagated to public variable l. To combat this, Jif records the program counter during compilation to check that any reads or assignments are to variables with policies at least as restrictive as those associated with the branch variable, in this case h. If l were public, it would be less restrictive than the branch condition h and the program would not compile.

```
class Train authority(driver) {

    void start() where authority(driver) {...}

    void stop() where caller(driver, emergencyBrake){...}

    void {driver<-* meet waitstaff<-*}doTeaService(){...}
    ...
}
```

Listing 3: Class authority.

To explicitly state that an application runs as a particular principal, the class can be marked with the authority of that principle. Principals can also be specified in the method signature to limit the principals that can call a particular method. The caller condition only allows principals listed in parenthesis to execute the method. More flexibly, a policy label can be specified before the method name as a precondition to running the method. These are illustrated in listing 3.

While Jif provides a comprehensive way to improve the security of information in Java programs, there are limitations to the solution. Several Java programming features are not supported by the language including nested classes, initialiser blocks and threads [2]. Since it is possible to use these features as covert channels for data propagation they cannot be ignored by Jif and due to their complex nature are difficult to provide solutions for.

By their very nature, data policies must be verbose to ensure that they correctly define the restrictions on data, however, from the view of the programmer they require a significant amount of typing and repetition to perform the simplest of tasks. `System.out.println()`

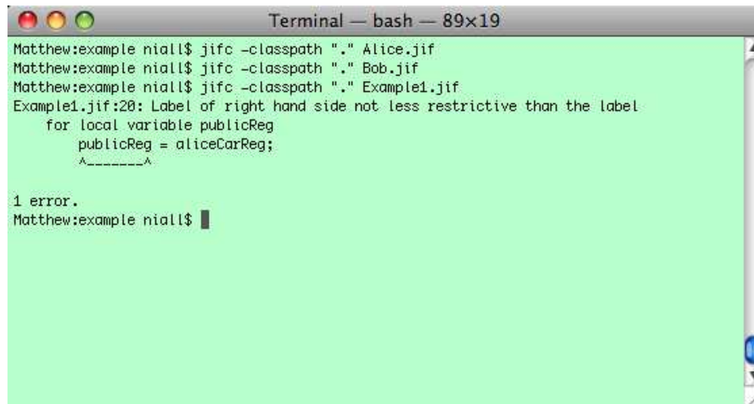

Figure 1: Compiling the first example.

requires a multiline initialisation to ensure that the output channel is secure before it can be used to print data to the screen. It can also quickly render code less-readable as programming logic becomes cluttered with security information. There are also some mismatches between the Jif logic and common understanding of logic symbols. Where regular expressions use * to represent *anything*, Jif uses it to model the most restrictive permission type $\top$. While this is a small issue, it is important to understand that the point of the technology is to limit information leaks and such discrepancies are likely to cause confusion and produce insecure systems.

Most notably, Jif is not part of the Java programming language and as such, Jif programs must be pre-compiled to Java before they can be compiled and run. This is taken care of in one step by the Jif compiler, however, it means that Jif classes are notably distinct from regular Java classes. Jif can make use of regular Java classes and be inter-operable but it isn't seamless.

6 Example

Since Jif statically checks the information flow in programs it is difficult to provide a full example which displays all the features of the programming language. Instead, small examples have been provided which include deliberate errors in the program to show how Jif detects and reports them.

The principal files Alice.jif and Bob.jif are taken from the Jif package test directory at `$JIF/tests/jif/principals/`. To avoid changing your classpath they should be copied to the same directory as the example files.

```
1 import jif.principals.*;
2
3 public class Example1
4 {
```

```
    public static void main{}(principal{} p, String[]{} args) {

    // Construct the principals
    final principal Alice = new Alice();
    final principal Bob = new Bob();

    // Set up private data, notice the fields only
    // have owners
    String{Alice:} aliceCarReg = "XYZ";
    String{Bob:} bobCarReg = "ABC";

    String{} publicReg = "";

    // attempt to set the public field with alice's
    // private car registration
    publicReg = aliceCarReg;
    }
}
```

Listing 4: Example 1.

As can be seen in figure 1 this example does not compile. Alice's car registration can only be read by Alice and therefore cannot be copied to the publicRef variable. One way around this would be to add a reader for aliceCarReg.

```
...
    String{Alice<-_} aliceCarReg = "XYZ";
...
```

Listing 5: Example 2.

However, this would allow the variable to be read across the program. More preferable would be to allow the car number-plate only to be made public when absolutely necessary. In order to do this, the program must declassify aliceCarReg before it is used.

```
...
    String{Alice:} aliceCarReg = "XYZ";
    String{Bob:} bobCarReg = "ABC";

    String{} publicReg = "";

    // attempt to set the public field with alice's
    // private car registration
    aliceCarReg = declassify(aliceCarReg, {Alice:; Alice<-_});
```

```
10      publicReg = aliceCarReg;
11      }
12  }
```

Listing 6: Example 3.

7 Resources

- **Jif Website** - *http://www.cs.cornell.edu/jif/*, Perhaps the most useful resource, the Jif website is where users can download the Jif compiler and libraries to experiment with the language. There are also many links to academic papers on both the language and DLM.
- **Jif Reference Manual** - *http://www.cs.cornell.edu/jif/doc/jif-3.1.1/manual.html*, Linked to from the Jif website is the Jif Reference Manual. This is perhaps the best way to get a grasp on the fundamentals of DLM as applied to the Jif programming language though sometime the text can be highly theoretical for a programming manual.
- **Jifclipse** - *http://siis.cse.psu.edu/jifclipse/index.html*, Jifclipse is a plugin for the Eclipse IDE and is designed to aid development of Jif programs. The plugin does not seem to be up-to-date with the latest version of Jif, however, the website does provide links to tools for compiling and constructing Jif policies.

8 Conclusion

Jif presents an approach to securely typed languages that claims to be usable, flexible and with significantly lower overhead than other proposed models for access control. It manages something unique in that it enforces control over the flow of information throughout the life-cycle of a program. Unlike the existing access controls in mainstream languages the mediation does not begin and end when the information is retrieved. However, there are several caveats to using the language for real-world development. Perhaps most importantly, Jif does not support threads. This to many is a crucial feature of the Java language and one that cannot be discarded lightly. Another point is that Jif is a Java language extension and requires its own compiler to develop applications. This limits its use in existing IDEs and forces the developer into using a jif-specific plug-in.

While there are clear benefits to using a securely typed language, many of which Jif implements, there is currently a lack of documentation to explain these concepts in practice and without support for threading Jif just isn't ready for the mainstream.

References

[1] Uk data protection act 1998. 1998.

[2] Stephen Chong, Andrew C. Myers, K. Vikram, and Lantian Zheng. *Jif Reference Manual*. Cornell University, Ithaca, NY, USA, http://www.cs.cornell.edu/jif/doc/jif-3.1.1/manual.html [last checked 10th March 2008], June 2006.

[3] Sun Microsystems. Secure coding guidelines for the java programming language. 2007.

[4] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[5] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.