JBLIF, a Tool for Non-interference Analysis of Java and Java Bytecode Programs

Salvador V. Cavadini
Project EVEREST, Institut National de Recherche en Informatique et Automatique (INRIA)
Sophia-Antipolis, 06902, France
Salvador.Cavadini@sophia.inria.fr

Abstract

Protecting sensitive information has become an important facet of software development. One aspect of software security relies on information flow control (IFC), a technique for discovering information leaks in software. Despite the large body of work on language-based IFC, there are only few implementation of information flow analyzers for full-scale real programming languages. This lack signifies a gap between IFC theory and practice. This work introduces, a tool that helps to overpass this gap: JBLIF –acronym from Java Bytecode-Level Information Flow–, a tool capable of statically detect information leaks in systems coded in Java and/or Java bytecode.

Keywords: information flow control, non-interference, data security, software engineering.

1 Introduction

There is an urgent need for software applications with strong confidentiality guarantees. Protecting sensitive information –e.g. credit card data, personal medical information, military secrets– has become an important facet of software development. The problem is not new but it has acquired relevance due to ubiquity of computing systems.

One aspect of software security relies on *information flow control* (IFC), a technique for discovering information leaks in software. One of the two main tasks of IFC is guarantee that confidential data is not made public through public variables. Contemporary IFC use different kinds of program analysis to provide such a guarantee.

Language-based IFC uses the program code to discover security leaks. Most language-based IFC approaches uses non standard type systems where security levels are coded as special types for variables and the typing rules catch illegal flow of information [12].

Non-interference, a semantical condition on programs, ensures that high-security data will not be observable on low-security channels [3]. Despite the large body of work on language-based IFC, there are only few full-scale implementation of non-interference analyzers for real programming languages. This lack signifies a gap between theory and practice.

We have developed a tool that can help to overpass this gap: JBLIF –acronym from Java Bytecode-Level Information Flow–, is a tool capable of perform static non-interference analysis of software systems coded in Java and/or Java bytecode and supports security annotations at both levels: source and bytecode.

This paper is organized as follows. Next section explains how non-interference and program dependencies are related and why program slicing can be naturally used in non-interference analysis. Section 3 provides a description of JBLIF and section 4 shows two examples of non-interference analysis using JBLIF. Related work is discussed at section 5. Finally, the conclusions and future works.

2 Non-interference and Program Dependencies

Typically, a confidentiality policy labels certain variables as being secret to enforce the independence between the final value of non-secret –i.e. public– variables and the initial values of secret ones. This is semantically interpreted by *non-interference*: a program satisfy the confidentiality policy if every pair of computations, from a pair of initial states differing only in secret variables, leads to final states with identical public variables [3]. Non-interference generalizes to a security lattice with more than two elements but for the sake of the explanation two elements lattices are used.

The property of non-interference is naturally related with the dependencies between program statements, and expressions [1]. If statement y uses a variable defined at statement x, then y is *data dependent* on x. If the execution of statement y is controlled by the value of an expression x, then y is *control dependent* on x. The set of statements/expressions on whom y depends is called the *backward slice* of y [16]:

$$BS(y) = \{x | y \text{ depends on } x\}$$

From this definition, it is possible to conclude that if statement y directly or indirectly depends on statement x, then information could flow from x to y, noted $x \rightsquigarrow y$. If y do

not depends on x, then it is guaranteed that information can not flow from x to y, noted $x \not\rightsquigarrow y$ [14], symbolically:

$$x \notin BS(y) \implies x \not\rightsquigarrow y$$

2.1 Using Program Slicing in Non-interference Analysis

This last implication permits to develop a slicing-based non-interference analysis [5]. The idea is to mark certain selected statements as *providing* or *allowing* flows at certain security level. A *provided security level* specifies that a statement generates information at this security level. An *allowed security level* specifies that a statement accepts flows with a security level up to this security level.

Non-interference analysis consists in check that statements allowing security level l_1 do not depend on statements providing information at security level l_2 higher than l_1. More formally, program Prg is non-interferent iff

$$\forall a \in A : (\nexists p \in P \mid p \in BS(a) \text{ and } Allows(a) < Provides(p))$$

where A (P) is the set of statements in Prg with an allowed (provided) security level and $Allows(x)$ ($Provides(x)$) is the security level allowed (provided) by statement x.

The advantage of this approach is that non-interference analysis depends solely on the soundness property of correct slices: program slices are computed conservatively thus they may contain too many statements but never too few. Another advantage is that analysis precision depends on the slicing algorithm precision, permitting fine tuning of the computational resources to be used in non-interference analysis.

3 Non-interference Analysis at Java and Bytecode Levels with JBLIF

In a context where the interest in the enforcement of software security properties increases and the access to technologies allowing easy of information exchange –such as Internet and mobile phones– is generalized, tools able to check the security of mobile or embedded code-based platforms are of capital importance.

These kind of applications are mainly developed in Java language and usually available as bytecode. That is why it is important to have a security enforcement tool capable to work on Java and bytecode programs. JBLIF is such a tool. It can be used to enforce non-interference while system development and also to check that after compilation to bytecode the system still satisfies non-interference –thus untrusted Java compilers can be used–. Because JBLIF can work directly on bytecode, non-interference can be also checked when original Java sources are not available, a common case when the program is downloaded from Internet.

JBLIF is based on the ideas described in section 2, i.e. it performs non-interference analysis using the different dependencies between program sentences. More specifically, in JBLIF, non-interference analysis relies on program slices computed at bytecode level.

JBLIF is coded in Java and uses the slicing library of Indus [10], a framework for analysis of full Java programs.[1] Indus provides a set of libraries that works on Jimple [15]

[1] With the exception of dynamic class loading, reflection, and native methods.

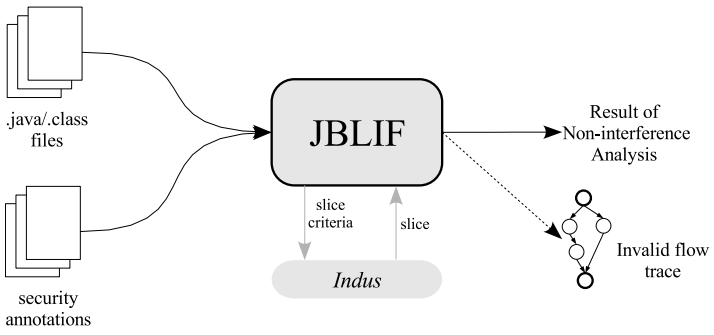

Figure 1: Highlevel architecture of JBLIF.

representation for Java bytecode. The slicing library gives access to functions computing highly customizable interprocedural context-sensitive slices in forward and backward direction.

In JBLIF, non-interference analysis is performed as follows –Figure 1–. First, the target system and the corresponding security annotations are loaded. Then, the tool computes the backward slice for each *allows-statement* and checks if some *provides-statement* in the slices has assigned a security level greater than the security level of the *allows-statement* being analyzed. If some invalid flow is detected, JBLIF informs it and generates a dependence graph that encodes the flows traces from the offending *provides-statement* to the *allows-statement*. The graph is the chop [7] between the two conflicting statements and it is generated in a format compatible with GraphViz [2].

3.1 Security Annotations in JBLIF

One characteristic that distinguish JBLIF from other non-interference analyzers is its ability to handle security annotations in both high level –Java– and low level –bytecode– programming languages. This way, users can add annotations to program components –variables, parameters, fields, statements, etc.– at the more convenient level.

Annotations are provided in separate files, one for each class in the system that the user needs to annotate and the same file can contain annotations at different levels. This scheme of annotations separated from the source code permits to check non-interference without modifying system files, thus preventing for unintentional program errors that could be introduced in the annotation process.

4 Examples

In this section, two small Java programs –listings 1 and 2– are used to illustrate how non-interference analysis is done with JBLIF.[2]

[2]Security annotations are given as program comments for the sake of clarity. As mentioned before, in JBLIF, security annotations are actually written in a separated text files.

Listing 1: A secure Java program.

```
1  public int secure (){
2
3      int secretInfo;
4      int publicInfo; // Provides: Public
5
6      secretInfo = System.in.read(); // Provides: Secret
7      if (secretInfo == 0){
8          publicInfo = 0
9      } else {
10         publicInfo = 1
11     };
12     publicInfo = 2;
13     System.out.print(publicInfo); // Allows: Public
14 }
```

The examples will also permit to emphasize that the dependence-based approach to non-interference analysis is more precise than the type system-based approach.

4.1 First Example: a simple Java program

This example exposes the main advantage of JBLIF non-interference analysis w.r.t. the type system based approach. Because type systems are usually flow insensitive they reject the program at Listing 1 considering that assignments at lines 8 and 10 are implicit flows from $secretInfo$ affecting the final value of $publicInfo$. By the contrary, JBLIF accepts the program because it can see that invalid implicit flows are killed by the assignment at line 12, thus the final value of $publicInfo$ is not related with $secretInfo$. More precisely, JBLIF computes the backward slice of sentence 13 –annotated as allowing up to *Public* level information flows–. This backward slice is the set of statements $\{4, 12, 13\}$ where only statement 4 is a *provides-statement* and generates a flow of *Public* information, thus the flow to statement 13 is valid.

If statement 12 is removed from the program, then the backward slice of 13 will result in the set $\{3, 6, 7, 8, 10\}$. Because statement 6 provides information at *Secret* level, JBLIF will be rejected the program as insecure.

4.2 Second Example: a more complex Java program

The program at Listing 2 –with a security lattice $High \rightarrow Low$– will permit to show how JBLIF deals with more complex features of Java language such as object sensitivity and dynamic dispatch. As explained before, JBLIF computes the backward slice for each one of the *allows-statements*, in this program, these statements are 18, 22, and 24. Then JBLIF checks each one of the slices looking for statements providing information at a security level higher than the allowed level:

Listing 2: A Java program with insecure flows (Taken from [4])

```
1  public class A {
2      int x;
3      void set(){x=0;}
4      void set(int i){x=i;}
5      int get() {return x;}
6  }
7  public class B extends A {
8      void set(){x=2;}
9  }
10 public class InfFlow {
11     public static void main(String[] a) {
12         int sec = 0; // Provides: High
13         int pub = 1; // Provides: Low
14         A o = new A();
15         o.set(sec);
16         o = new A();
17         o.set(pub);
18         outputInt(o.get()); // Allows: Low
19         if (sec==0 && a[0].equals("007"))
20             o = new B();
21         o.set();
22         System.out.println(o.get());// Allows: Low
23         o.set(42);
24         System.out.println(o instanceof B);// Allows: Low
25     }
26 }
```

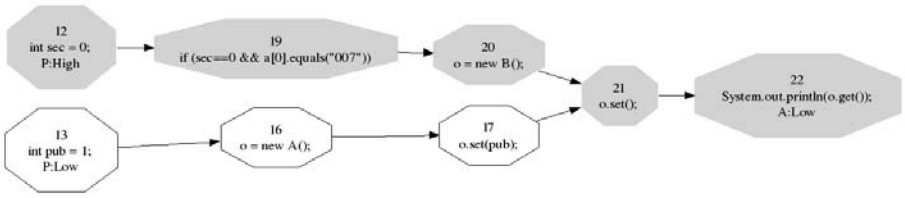

Figure 2: Partial backward slice, as is generated by JBLIF, from line 22 of Listing 2 highlighting –gray nodes– an invalid flow trace.

- The backward slice for statement 18 includes only one *provides-statement*: statement 13. Because 13 provides *Low* flows, the flow to 18 is safe. Notice that JBLIF is able to see that the object referenced at line 18, is in fact the object created at line 16 and set with a *Low* value –variable *pub*– and not the first created object –line 14– which was set with a $High$ value –variable *sec*–. This is possible because JBLIF has object sensitivity, something that is very hard to achieve with a type system.

- The backward slice for statement 22 includes statement 20, thus statements 19 and 12 are also included. Beacause the later provides a $High$ flow, JBLIF will indicate the existence of an invalid flow from 12 to 22, and the corresponding flow trace –Figure 2– is generated.

- The backward slice for statement 24, as the slice for 22 does, includes statement 12 because statements 19 and 20 are also included; JBLIF will highlight the invalid flow from 12 to 24 and generate the corresponding flow trace.

5 Related Work

Language-based information-flow security has a long, rich history with many –mostly theoretical– results [12]. Despite this large body of work, there are only few full language implementations of non-interference analyzers. In this section we provide a short overview of them and a comparison with JBLIF.

Jif is an information-flow typed extension of Java that builds upon the *decentralized label model* [9] and supports flexible and expressive information flow policies. Recently, Hicks et al. introduced FJifP, an extension of Jif that implements the *trusted declassification model* [6]. Flow Caml by Simonet et al. consists in an extension of the Objective Caml language with a type system tracing information flow [13].

The main advantage of JBLIF over these tools is that non-interference is enforced without rewriting the system in a new language. Moreover, the original source code of the system is not needed because JBLIF works at bytecode level, thus only `.class` files

are necessary. Other distinctive characteristic of JBLIF w.r.t. the above mentioned tools is related with the analysis approach: while JBLIF uses a dependence-based technique, Jif, FJifP, and Flow Caml use type systems; thus JBLIF analyses are more precise and, as mentioned previously, less false alarms are generated.

Recently, in [8], Li and Zdancewic presented an embedded security sublanguage of Haskell for enforcing information-flow policies in the standard Haskell programming language. Their approach has the advantage, over other systems like Jif, that the information-flow type system encoding is done using general features of Haskell without the need of a new language. Anyway, parts of the system where non-interference is to be enforced must be recoded, something that is not needed with JBLIF.

In [4], Hammer et al. present a dependence graphs-based system to check intransitive non-interference [11]. They inform that the system was implemented as an Eclipse plug-in that handles full Java. As far as we know, this tool and JBLIF are the sole tools implementing non-interference analysis for full Java language. The principal differences between JBLIF and this system are:

1. JBLIF works at both Java and bytecode level while Hammer's system works only at Java level,

2. JBLIF is not able to handle intentional information declassification as Hammer's system does,

3. In JBLIF, annotations are made in separated files and source code remains untouched. In Hammer's system, annotations are made in the Java source files,

4. JBLIF is available upon request to the authors.

6 Conclusions and Future Work

Despite the urgent need for strong confidentiality guarantees, the large body of literature, and considerable attention from the research community, information-flow based enforcement mechanisms are not widely used. One reason for this is the lack of a full-scale implementation of non-interference analysis for popular programming languages. JBLIF helps to overpass this gap between research and practice by making available non-interference analysis for full Java and also for Java bytecode. Moreover, JBLIF is the first tool to provide a flexible annotation mechanism allowing annotations at both Java, and bytecode level in separated files keeping source code untouched.

We have planned to extend JBLIF as a plug-in for the Eclipse IDE to facilitate the use of JBLIF in real cases of confidential Java programs development. Future work also includes adding the possibility of information declassification. Many realistic systems need to declassify some kind of confidential information as part of their normal behavior. The actual challenge is to differentiate between proper and improper declassification of confidential information. This problem will be the object of our coming research efforts.

JBLIF is available upon request to the authors.

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, pages 147–160, New York, NY, USA, 1999. ACM Press.

[2] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[3] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[4] Christian Hammer, Jens Krinke, and Frank Nodes. Intransitive noninterference in dependence graphs. In *Second International Symposium on Leveraging Application of Formal Methods, Verification and Validation (ISoLA 2006)*, pages 136–145, 2006.

[5] Christian Hammer, Jens Krinke, and Gregor Snelting. Information flow control for java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering*, 2006.

[6] Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. Trusted declassification:: high-level policy for a security-typed language. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 65–74, New York, NY, USA, 2006. ACM Press.

[7] Daniel Jackson and Eugene J. Rollins. A new model of program dependences for reverse engineering. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 2–10, New York, NY, USA, 1994. ACM Press.

[8] Peng Li and Steve Zdancewic. Encoding information flow in haskell. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, page 16, Washington, DC, USA, 2006. IEEE Computer Society.

[9] Myers and Liskov. Complete, safe information flow with decentralized labels. In *RSP: 19th IEEE Computer Society Symposium on Research in Security and Privacy*, 1998.

[10] V. P. Ranganath and J. Hatcliff. An overview of the indus framework for analysis and slicing of concurrent java software (keynote talk - extended abstract). pages 3–7, 2006.

[11] A. W. Roscoe and Michael Goldsmith. What is intransitive noninterference? In *PCSFW: Proceedings of The 12th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1999.

[12] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications, 21(1)*, 2003.

[13] V. Simonet. Flow caml in a nutshell. In *Proceedings of the first APPSEM-II workshop*, pages 152–165, 2003.

[14] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient path conditions in dependence graphs for software safety analysis. 15(4):410–457, October 2006.

[15] Raja Vallee-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations, 1998.

[16] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.