Information Flow Analysis for Java Bytecode

Samir Genaim and Fausto Spoto

Dipartimento di Informatica, Università di Verona
Strada Le Grazie, 15, 37134 Verona, Italy
genaim@sci.univr.it, fausto.spoto@univr.it

Abstract. We present a context-sensitive compositional analysis of information flow for full (mono-threaded) Java bytecode. Our idea consists in transforming the Java bytecode into a control-flow graph of *basic blocks* of code such that the complex features of the Java bytecode made explicit. The analysis is based on modeling the information flow dependencies with Boolean functions which leads to an accurate analysis and efficient implementation which uses Binary Decision Diagrams to manipulate Boolean functions. To the best of our knowledge, it is the first implementation of an analysis of information flow for the full Java bytecode. The work is still in progress but it is already support a quite large portion of the Java bytecode which includes exceptions and the subroutine handling mechanism.

1 Introduction

Information flow analysis aims to infer (information) dependencies between the program variables, and it is usually used to verify that a program is free of undesired information flows, i.e that the program is secure with respect to a given security policy. A security policy can be defined as a complete lattice of security classes, where information is allowed to flow from variables of a specific security class, to variables of higher security classes [7]. Static analysis techniques have been used to check if programs meet their security policies. These techniques range from *data/control*-flow [4, 5, 12, 15, 14] to type-inference [16, 18, 3, 2, 11, 8]. In *data/control*-flow approaches, the analysis usually infers a (super-) set of all possible information flows, from which the security properties can be observed. In typed-based approaches, given a classification of the program variables into security classes, the analysis is designed in such a way that well-typed programs do not leak secrets.

In this work we develop an information flow analysis for the full (mono-threaded) Java bytecode. Type-Based information flow analysis for the Java bytecode was previously studied in [3, 2, 11]. In order to achieve both precision and efficiency we build on two ideas: (1) Using Boolean function to model information flow which allow us to use Binary Decision Diagrams; and (2) Transforming the Java bytecode into a control-flow graph of *basic blocks* of code such that the complex (control-flow) features of the Java bytecode made explicit in the graph. The work still in progress, but it is already support some complex feature of the Java bytecode such as exceptions and virtual method calls.

2 Modeling information flow with Boolean Functions

The idea of using Boolean functions to model information flow was introduced in [8] and demonstrated on a simple while language. For a given program P, the input/output information flow dependencies are described using a Boolean function φ_P such that its models describe the possible information flow scenarios of P. In what follows we demonstrate how to construct this Boolean functions for explicit and implicit flows.

Consider the single assignment $C \equiv x := y + z$. The information flow behavior of C is: (1) information may *flow* from y and z to x; and (2) the information that flow to y and z remain the same as before the execution since they are not updated. Now let $\check{x}$, $\check{y}$ and $\check{z}$ ($\hat{x}$, $\hat{y}$ and $\hat{z}$) be Boolean variables that correspond to the input (output) states of the corresponding program variables. Using these Boolean variables, the above information flow behavior can be expressed as follows:

$$\varphi = \underbrace{[\hat{x} \leftrightarrow (\check{y} \vee \check{z})]}_{(1)} \wedge \underbrace{[(\hat{z} \leftrightarrow \check{z}) \wedge (\hat{y} \leftrightarrow \check{y})]}_{(2)}$$

The models of φ, i.e. the assignments for which φ is satisfiable, describe all possible information flow behaviors for C. For example, the model $\{\check{y}, \hat{x}, \hat{y}\}$ of φ describes a possible information flow from y to x and y itself (since it is not updated). Similarly, the model $\{\check{z}, \hat{x}, \hat{z}\}$ describes a possible information flow from z to both x and z. In general a model m of φ includes two sets of Boolean variables: those that correspond to the *input* (e.g. $\check{y}$) and those that correspond to the *output* (e.g. $\hat{x}$), and it describes an information flow from the input to the output of the corresponding program variables.

Let us consider another kind of information flow which stems from guards of conditional statements. For example, in the following statement:

$$C_1 \equiv \text{if } (w = 0) \text{ then } x := y \text{ else } z := y.$$

The information flow behavior of C_1 consists of: (1) *explicit* information flow that stems from the "*then*" or the "*else*" branches; and (2) *implicit* flow from the guard's variables, i.e. w, to all variables that *might* be updated during the executions, i.e. x and z, because by watching the values of x and z we *may* learn whether $w = 0$ or $w \neq 0$. Suppose we already have constructed Boolean functions φ_1 and φ_2 that describe the information flow in the "*then*" and "*else*" branches respectively, in order to construct a Boolean function that describes the information flow of C_1 we need to add the implicit flows (from w to x and z) to $\varphi_1 \vee \varphi_2$, one way for doing this is as follows:

$$\varphi_3 = \underbrace{(\varphi_1 \vee \varphi_2)}_{(1)} \wedge \underbrace{(\hat{x}' \leftrightarrow (\hat{x} \vee \check{w})) \wedge (\hat{z}' \leftrightarrow (\hat{z} \vee \check{w}))}_{(2)}$$

The idea is to define new output variables $\hat{x}'$ and $\hat{z}'$, that consider the old information flows to $\hat{x}$ and $\hat{z}$ (of $\varphi_1 \vee \varphi_2$) as well as the new information flow from $\check{w}$. We could also eliminate the *old* output Boolean variables $\hat{x}$ and $\hat{z}$ from φ_3 and

then rename $\hat{x}'$ and $\hat{z}'$ to $\hat{x}$ and $\hat{z}$ respectively in order to keep the representation uniform. You can verify that the models of this φ_3, indeed represents all possible information flow caused by C_1.

3 Information Flow Analysis For the Java Bytecode

In this section we describe briefly how we translate a Java bytecode program into a graph of *basic blocks* of code such that the complex (control-flow) features of the Java bytecode made explicit, and how we translate this graph into an equations system of Boolean function such that its least solution approximates that information flow of the corresponding program. At the moment, for simplicity, we ignore implicit flows and later we describe how to handle it.

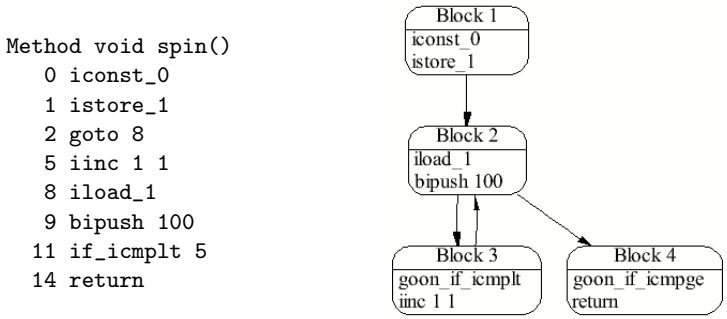

Fig. 1. A bytecode and its graph of basic blocks.

In order to generate the graph we use a technique already applied in [1] to other languages. It consists in splitting the code into chunks of contiguous bytecodes, called *basic blocks*. Jumps of control can only occur at the end of a chunk and the target of a jump can only be the first bytecode of a chunk. Basic blocks are linked through edges representing transfer of control. Consider for instance the bytecode in Figure 1, taken from [13]. It is transformed into the graph of basic blocks shown in the same figure. The two `goon` bytecodes are new *filter* bytecodes which select the right branch of execution. The `goto` at line 2 has now become an arrow between blocks 1 and 2. Moreover, the loop from line 5 to 11 is apparent in Figure 1 since blocks 2 and 3 call each other.

The graph of basic blocks such as that in Figure 1 is used to compute the meaning or *denotation* of a method through a fixpoint computation, local to the method. For more details, see [17]. Here we just note that the bytecodes contained in this graph are always *state transformers i.e.,* their semantics is a map from an input *state* to an output *state*.

A Java bytecode b can be seen as (*denoted* by) a state transfer function $[\![b]\!]$ over states, where a state is a triple $\langle l, s, \mu \rangle$, where l is an array of local variables (indexed from 0), s an operand stack (whose deepest element is indexed with 0)

and μ the memory or heap of the system. For instance,

$$\begin{aligned}
[\![\texttt{bipush } i]\!] &= \lambda\langle l, s, \mu\rangle.\langle l, s :: i, \mu\rangle \\
[\![\texttt{idiv}]\!] &= \lambda\langle l, s :: i_1 :: i_2, \mu\rangle.\langle l, s :: i_1/i_2, \mu\rangle \\
[\![\texttt{aload } n]\!] &= \lambda\langle l, s, \mu\rangle.\langle l, s :: l(n), \mu\rangle \\
[\![\texttt{astore } n]\!] &= \lambda\langle l, s :: v, \mu\rangle.\langle l[n \mapsto v], s, \mu\rangle \\
[\![\texttt{goon_if_icmplt}]\!] &= \lambda\langle l, s{::}i_1{::}i_2, \mu\rangle.\begin{cases} \langle l, s, \mu\rangle & \text{if } i_1 < i_2 \\ \textit{undefined} & \text{otherwise} \end{cases}
\end{aligned} \tag{1}$$

Using the above transfer function, we can translate each rule an abstract rule which capture the information flow in terms of a Boolean functions, for example the abstract denotation for $[\![\texttt{aload } n]\!]$ which load the n^{th} local variable into the top of the stack can be defined as:

$$[\![\texttt{aload } n]\!] = (\check{l}_n \leftrightarrow \hat{s}_{top}) \wedge \psi(lc) \wedge \xi(top)$$

where the Boolean functions $\psi(lc)$ and $\xi(top)$ express respectively the fact that the local variables and the stack elements (except the *top*) keep their information flow properties unchanged. Similarly we can translate all the transfer rules. Then using this rules we can generate from the control flow graph an equations system of Boolean function such that its least solution approximates the information flows in the program.

In the above example we handled the stack element s_{top} as it were a local variables, this is possible due to an important feature (in this context) of the Java bytecode which requires the hight of the stack at each program point to be the same in all executions. Due to this feature information flow cannot stem from observing the stack hight, and hence simplify the analysis.

Other commands of the Java bytecode are handled in a similar (but more technical) way, and complex features of the Java bytecode such as exceptions and virtual method calls are compiled into the control flow graph explicitly, namely for a given virtual method call we have branches for all possible methods that might be called at that point, this is obtained by applying a pre-analysis which approximate the set of possible types (classes) for each program variables at each program point. Currently fields are not supported by our analysis and they are simply approximated by $\top$.

In our graph representation, implicit flow arises every time there is a branch in the graph of basic blocks, depending on the outcome of some test. Hence, to spot the sources of implicit flows it is enough to look in the graph for those blocks with more successor. In addition another two problems must be solved: (1) we must compute the scopes of the choice points in the graph of basic blocks so we know exactly which commands are affected by the implicit flows; and (2) we must add to the Boolean functions of these scopes the implicit information flow. The first problem is a classical graph theory problem and solved using graph algorithms, the idea is to find the meeting point of all branches, and the second problem is solved by introducing a context Boolean variables for each basic block through which we add the implicit flows.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles Techniques and Tools*. Addison Wesley Publishing Company, 1986.
2. Gilles Barthe, Amitabh Basu, and Tamara Rezk. Security types preserving compilation. In *VMCAI'04*, volume 2937 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
3. Gilles Barthe and Tamara Rezk. Secure information flow for a sequential java virtual machine. Unpublished.
4. C. Bodei, P. Degano, F. Nielson, and H.R. Nielson. Static analysis for secrecy and non-interference in networks of processes. In *Proc. of PaCT'01*, volume 2127 of *Lecture Notes in Computer Science*, pages 27–41. Springer-Verlag, 2001.
5. D. Clark, C. Hankin, and S. Hunt. Information flow for algol-like languages. *Computer Languages*, 28(1):3–28, April 2002.
6. JFlow: Practical Mostly-Static Information Flow Control. Andrew c. myers. In *26th ACM Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, Texas, 1999.
7. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, 1976.
8. Samir Genaim, Roberto Giacobazzi, and Isabella Mastroeni. Modeling secure information flow with boolean functions. In Peter Ryan, editor, *WITS'04*, April 2004.
9. R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *The 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04).*, January 2004. to appear.
10. J. A. Gougen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, 1982.
11. Naoki Kobayashi and Keita Shirane. Type-based information flow analysis for low-level languages. In *3rd Asian Workshop on Programming Languages and Systems*, 2002.
12. P. Laud. Semantics and program analysis of computationally secure information flow. In *In Programming Languages and Systems, 10th European Symposium On Programming, ESOP*, volume 2028 of *Lecture Notes in Computer Science*, pages 77–91. Springer-Verlag, 2001.
13. T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Java^{TM} Series. Addison-Wesley, 1999.
14. M. Mizuno. A least fixed point approach to inter-procedural information flow control. In *Proc. 12th NIST-NCSC National Computer Security Conference*, pages 558–570, 1989.
15. A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, 2001.
16. C. Skalka and S. Smith. Static enforcement of security with types. In *ICFP'00*, pages 254–267. ACM press, 2000.
17. F. Spoto. Focused Static Analyses for the Java Bytecode. Submitted for publication, 2004.
18. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.