Java Bytecode Verification by Model Checking*
System Abstract

David Basin[1], Stefan Friedrich[1], Joachim Posegga[2], and Harald Vogt[2]

[1] Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Universitätsgelände Flugplatz, D-79110 Freiburg i. Br. Germany.
{basin|friedric}@informatik.uni-freiburg.de

[2] Deutsche Telekom AG,
IT-Research Security (TZ/FE34)
D-64307 Darmstadt, Germany.
{posegga|vogth}@tzd.telekom.de

1 Motivation

Verification plays a central role in the security of Java bytecode: the Java bytecode verifier performs a static analysis to ensure that bytecode loaded over a network has certain security related properties. When this is the case, the bytecode can be efficiently interpreted without runtime security checks.

Our research concerns the theoretical foundations of bytecode verification and alternative approaches to specifying and checking security properties. This is important as currently the "security policy" for Java bytecode is given informally by a natural language document [LY96] and the bytecode verifier itself is a closed system (part of the Java virtual machine). We believe that there are advantages to more formal approaches to security. A formal approach can disambiguate the current policy and provide a basis for verification tools. It can also help expose bugs or weaknesses that can corrupt Java security [MF97]. Moreover, when the formal specification is realized in a logic and verification is based on a theorem prover, extensions become possible such as integrating the verification of security properties with other kinds of verification, e.g., proof-carrying code [NL96, NL98].

2 Approach

We provide a formal foundation to bytecode verification based on model checking. The idea, which has similarities with data flow analysis and abstract interpretation [Sch98], is as follows. The bytecode for a Java method M constitutes a state transition system where the states are defined by the states of the Java

* The opinions expressed in this paper are those of the authors and do not necessarily reflect the views of their respective employers.

Virtual Machine (JVM) running M, and the transitions are given by the semantics of the JVM instructions used in M. From M we can compute an abstraction M_{fin} that abstracts the state-transition system to a simpler one whose states are defined by the values of the JVM's program counter, the operand stack, a stack pointer, and the method's local variables. The actual values of the stack positions and local variables are abstracted away and simply represented by their type information. The transition rules of M_{fin} are defined likewise by the semantics of the JVM machine instructions with respect to our abstraction. Since only finitely many types appears in each method, the resulting abstraction M_{fin} is finite; the size of the state-space is exponential in the number of local variables and the maximal stack height.

After we can apply a model checker to M_{fin}. The properties that we model check correspond to the type safety checks performed by the Java bytecode verifier. For example, we specify that each transition in M_{fin} that represents a machine instruction in M finds appropriately typed data in the locations (stack or local variables) it uses. The model checker then either responds that the byte code is secure (with respect to these properties) or provides a counter-example to its security.

3 Architectural Description

The overall structure of our system is depicted in Figure 1. As input it takes a Java class file as well as a specification of an abstraction of the Java virtual machine. The specification defines the states of the abstract machine and how each bytecode instruction changes the machine's state. For each instruction, a precondition to its execution is given (e.g. that the operand-stack must contain enough operands of appropriate type) and also invariants are stated (e.g. that the stack may not exceed its maximal size). These are the properties that are model checked.

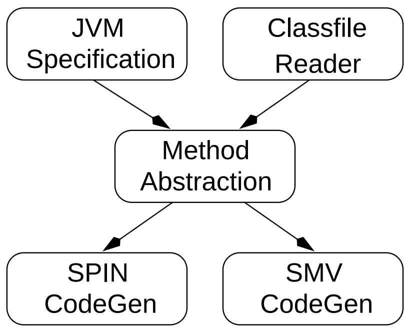

Fig. 1. Structure of the compiler

The core routine (method abstraction) translates bytecode into a finite state transition system using the specification of the abstract machine. Separating the machine specification from the translation gives us a modular system where we can easily change the virtual machine and the properties checked. Our system is also modular with respect to the model checker used. Currently we have implemented two different back-ends: one that compiles the transition system and properties to the input language of the SMV model checker and a second that generates output in the SPIN language Promela.

4 Example Output

As a simple example (even here we must elide details) we give (a) a Java program, (b) the corresponding bytecode, and (c) the output of our system, which is input for the SPIN model checker.

```
public static int fac(int a){
       if (a==0)
         return 1;
       else
         return a*fac(a-1);}
```

(a) Java Code

```
.method public static fac(I)I
    .limit stack 3
    .limit locals 1
.line 8
    iload_0
    ifne Label1
.line 9
    iconst_1
    ireturn
.line 11
Label1:
    iload_0
    iload_0
    iconst_1
    isub
    invokestatic Sample/fac(I)I
    imul
    ireturn
.end method
```

(b) Bytecode

```
#define pc_is_1 (pc == 1)
#define pc_is_2 (pc == 2)

/* Conditions to be checked */
#define cond_1 (locals[0] == INT)
#define cond_2 (st[stp_st - 1] == INT)

[...]

/* State of the abstract machine       */
byte pc;           /* program counter  */
byte st[3];        /* operand stack    */
byte stp_st;       /* stack pointer    */
byte locals[1]     /* local variables  */

/* Process that watches if the conditions hold */
proctype asrt_fac() {
  assert( ( !pc_is_1 || cond_1) &&  [...])}

/* Process that models the transition system */
proctype meth_fac() {
  do
     /* iload_0 */
  :: pc_is_1 -> atomic {
                  pc = pc + 1;
                  st[stp_st] = locals[0];
                  stp_st = stp_st + 1 };

     /* ifne Label1 */
  :: pc_is_2 -> atomic {
                  if
                  :: pc = pc + 5;
                  :: pc = pc + 3
                  fi;
                  stp_st = stp_st - 1 };

 [...]

od }

/* Initialization of the abstract machine */
init {
atomic {
  pc = 1; stp_st = 0; locals[0] = INT;
  run meth_fac(); run asrt_fac() } }
```

(c) fac_{fin} and Properties

The Java program and the bytecode should be clear. We have added by hand some comments to (c). In the process `meth_fac`, the transitions of the method `fac` are modeled. For example, the first instruction of the method `iload0` loads an integer value from a local variable on the stack; the corresponding condition to be checked (`cond_1`) requires that the respective variable contains an integer value. The instruction `ifne` performs a conditional branch, which is modeled by nondeterministically assigning a new value to the program counter. The process `asrt_fac` runs in parallel to the process `meth_fac` and checks if all conditions (preconditions and invariants) are fulfilled. SPIN checks this in negligible time (0.03 seconds).

5 Future Work

We have completed Version 1 of the system. This formalizes and model-checks the JavaCard subset of Java, which is used for smartcards [Sun98]. We have chosen this particular instance of Java for three reasons: first, JavaCard does not allow for dynamic class loading, therefore there are no "real-time" requirements for bytecode verification. Second, the bytecode verifier for JavaCard lives outside the client platform, so it can easily be replaced/extended without modifying the platform itself. Finally, our aproach can contribute to meeting the high security requirements that smartcard applications usually have.

In a future release we plan to extend this version to the full JVM instruction set. The only significant problems that might occur are run time requirements for the model checker (defined by the time a user is willing to wait when loading a class) and multi-threading, which is not possible in JavaCard and could increase the model checker's search space.

References

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[MF97] G. McGraw and E.W. Felten. *Java Security: Hostile Applets, Holes, and Antidotes.* Wiley, 1997.

[NL96] G. Necula and P. Lee. Proof-Carrying Code. Technical Report CMU-CS-96-165, Carnegie Mellon University, School of Computer Science, Pittsburg, PA, September 1996.

[NL98] G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 61–91. Springer, 1998.

[Sch98] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Conference Record of POPL'98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 38–48, San Diego, California, January 19–21, 1998.

[Sun98] Sun Microsystems, Inc. Java Card 2.1 Language Subset and Virtual Machine Specification. http://java.sun.com:80/products/javacard/, 1998.