Modeling the Java Bytecode Verifier

Mark C. Reynolds[1],[2]

Computer Science Department
Boston University
Boston, US

Abstract

The Java programming language has been widely described as secure by design. Nevertheless, a number of serious security vulnerabilities have been discovered in Java, particularly in the Bytecode Verifier, a critical component used to verify class semantics before loading is complete. This paper describes a method for representing Java security constraints using the Alloy modeling language. It further describes a system for performing a security analysis on any block of Java bytecodes by converting the bytes into relation initializers in Alloy. Any counterexamples found by the Alloy analyzer correspond directly to insecure code. Analysis of the approach in the context of known security exploits is provided. This type of analysis represents a significant departure from standard malware analysis methods based on signatures or anomaly detection.

Keywords: JVM bytecodes, Alloy, lightweight modeling, Java security.

1 Introduction

This paper will describe an analysis tool for verifying security constraints within Java bytecodes. This investigation was motivated by the continued appearance of malicious Java code that violates the security constraints imposed by the Java compiler, the Java Bytecode Verifier and the Java runtime. The analysis approach is based on the lightweight modeling language Alloy [1], [2]. This paper will describe the security verification approach taken by the Java Virtual Machine (JVM), and briefly enumerate some of the ways that it has been circumvented. This will be followed by a description of the design of the analysis tool and its implementation. Particular emphasis will be given to describing the Bytecode Verifier security constraints and the degree to which they are checked by the current model. The work described in this paper is a significant extension of [3]. Finally, a path toward future work will be described. The analysis tool has, in fact, proven to be a powerful approach to analyzing JVM security constraints, as it is capable of detecting several forms of malicious bytecodes.

[1] Thanks to Assaf Kfoury of the BU CS Dept
[2] Email: markreyn@cs.bu.edu

1.1 Background

The Java programming language has been touted as "secure by design" since its inception. However, attacks against Java security have been promulgated from the earliest days of Java. Felten discovered several weaknesses in the Java security model almost immediately, and his work on Java [4] contains an extensive list of early exploits. The development of Java malware has continued unabated up to the present. The Common Vulnerabilities and Exposures project [5] lists numerous Java bugs that can lead to privilege escalation, exfiltration of sensitive data, denial of service and other malicious outcomes. A recent announcement (November 2009) contained more than ninety previously unpublished Java vulnerabilities.

In order to understand how these security failures come about, it is first necessary to briefly review the Java security model. Java security is enforced in three ways. The Java compiler has a large number of rules that it enforces in order to ensure that the syntax and semantics of the Java language are satisfied, but also to prohibit certain actions that are known to be associated with malicious code. For example, the Java compiler will refuse to compile any program that contains a method that makes use of an uninitialized variable. The output of the Java compiler is a binary file known as a classfile. In order for a Java application or applet to use the methods provided by a class, it must load the classfile that contains that class into the Java execution environment. Loading is accomplished by a Java classloader. Whenever a class is loaded the Java Bytecode Verifier is invoked. The Bytecode Verifier checks that the contents of the classfile conform to the classfile format and also verifies a large number of security constraints before it will allow the classloader to complete loading of that classfile. Finally, the Java runtime performs array bounds checking, runtime type conversion checking and a number of other tests.

Almost all Java exploits to date have used weaknesses in the Bytecode Verifier. The Bytecode Verifier's rules are described in great detail in the JVM specification [8]. The Bytecode Verifier uses a constraint based approach in performing its analysis. For example, it checks that all local variables are written before being read, that each instruction receives precisely the set of operands that it is expecting, that the stack has the same depth at each program point regardless of execution path used to reach that program point, and many other constraints.

Our approach uses Alloy to perform constraint analysis on Java bytecodes. It attempts to emulate the constraint checking that is ostensibly being performed by the Bytecode Verifier. In Alloy it is very easy to express constraints in terms of formulas involving relations, and therefore it has proven to be a rich environment for checking Java security constraints. Previous efforts have been made to apply formal methods to Java bytecodes [9], [10], [11], among others, but these efforts have used a more heavyweight model checking approach that attempts to prove soundness, as opposed to Alloy's lightweight constraint based approach that converts assertions into Boolean formulas and then searches for satisfaction assignments or the existence of counterexamples.

Several of the security constraints imposed by the JVM have already been mentioned. In general, we would like to focus on "high value" constraints. A constraint is said to be high value if known malware violates that constraint. Thus, this paper will focus on constraints associated with access to uninitialized or out-of-bounds

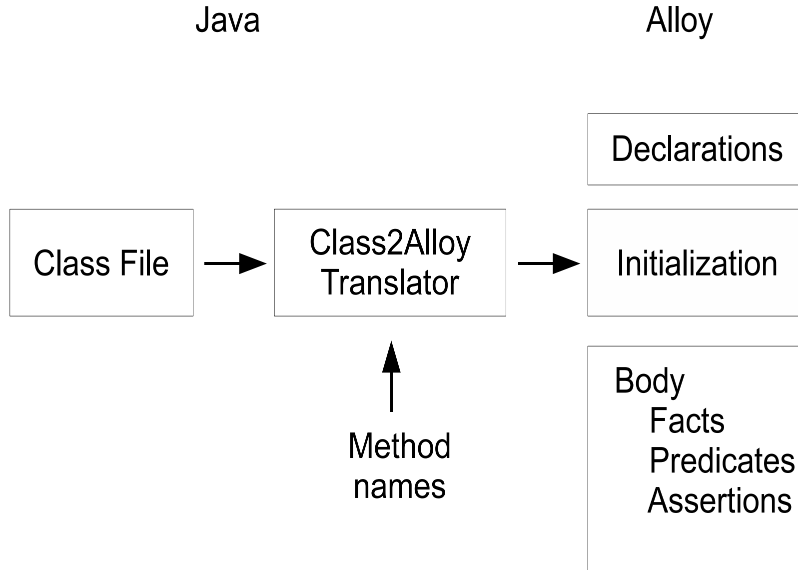
memory, as well as constraints associated with instruction transfer and method resolution. In order to create an extensible model for the operation of the Bytecode Verifier, a general framework for code analysis was created such that adding additional constraints would involve only incremental modifications, and not a complete restructuring of the model code. The current implementation concretely models several high value security constraints, as will be described below. The work to date strongly suggests that the current implementation can be readily adapted to additional constraints.

2 Design

Alloy is a lightweight modeling language that uses first order logic. Alloy is capable of analyzing assertions for satisfiability and also for the existence of counterexamples. A key observation is that the security constraints imposed by the JVM can be modeled as invariants, and thus can be analyzed by the Alloy Analyzer. Alloy is not a proof system, so the failure to find a counterexample to a constraint is not a proof that that constraint is always satisfied, only that the constraint is satisfied within the search space specified. If a counterexample is found, however, that does indicate that the invariant has been violated, and the Alloy Analyzer conveniently provides a graphical representation of that counterexample.

In the approach taken so far, the Alloy model is realized as a template containing a fixed set of relations, functions, facts, predicates and assertions. This model is then supplemented by relation initializers that are derived from particular JVM code. In this approach, the template portion of the Alloy model is completely independent of any choice of Java bytecodes, while the initializers depend only weakly on the detailed implementation of the template. Specifically, the initializers being generated only depend on the set of relations being initialized, and not on any specific way in which the constraints were realized in the model template. This decoupling between the "data" portion of the model and the "code" portion of the model makes it easy to extend with additional constraints. The model template can be further refined into three subcomponents: (1) the relation definitions; (2) the execution engine; and (3) the constraint assertions. The relation definitions are Alloy definitions of the top level signatures, as well as the definitions of the relations themselves. These relation definitions capture the static properties of JVM instructions, methods and classes, as well as capturing the JVM state as the execution engine executes. All other components of the Alloy model are logically dependent on the relation definitions.

The relation initializers are the initial values of the Alloy relations. They are generated from specific JVM code. Relation initializers need to be generated from one or more specified Java methods. Therefore, there needs to be an automatic way of converting the Java bytecodes in a set of methods into these relation initializers. To this end, a Java classfile parser was created to perform this conversion. The parser takes a Java classfile as input and produces an Alloy model fragment as output. When the model fragment is combined with the Alloy template, a complete Alloy model is produced, as is shown in Figure 1.



The relation definitions and their initializers form a static representation of a set of properties of the Java methods being analyzed. In order to observe dynamic behavior, this static representation needed to be extended with model actions that would mimic the execution of the JVM itself, at least to the extent that the JVM's Bytecode Verifier synthetically executes method code in order to perform its constraint checking. Thus, an execution engine was needed. This execution engine represents the flow of execution through the medium of stateful relations. Alloy's "ordering" utility is used for representing this state. Execution could not be unbounded, of course, since Alloy only performs analysis over a finite set of states. It would have been possible to simply let Alloy "fall off the end" of execution, which is to say to allow the analyzer to perform an exhaustive analysis of all possible states in the state space. For both performance and structural reasons this was deemed to be an unacceptable solution. Therefore, the execution engine was designed such that certain JVM instructions are designated as terminal instructions. The execution engine was then implemented to recognize this condition and act on it so as to not create further unique states. Of course, this models the actual execution of the JVM itself. Certain instructions within a method are, in fact, terminal, in that they cause the method to be exited. One obvious question is the manner in which iterative constructs are handled by the execution engine. Would it provide better model fidelity to have the execution engine attempt to exactly mimic runtime execution, or would this lead to unacceptable performance penalties? In fact, the execution engine does not attempt to perform any branch prediction analysis in the model. The precise way in which this was handled, and its implications, will be explained in the **Implementation** section below.

Finally, the model must provide for a way in which each JVM security constraint is actually checked by Alloy. Formulating the security constraints as Alloy assertions proved to be straightforward once the model had been constructed to accurately

reflect the static and dynamic properties of the method code.

3 Implementation

The implementation of the JVM security constraints analyzer will now be described. First, the three components of the model template, namely the relation definitions, the execution engine, and the security constraint assertions, will be described. This will be done in parallel with descriptions of the ways in which malicious code would violate the constraints. The implementation of the Class2Alloy classfile parser which is used to generate the relation initializers will then be discussed.

3.1 Model Template

The model template employs five top level signatures, the *Global* signature, the *Clazz* signature, the *Method* signature, the *Instruction* signature and the *State* signature. The *Global* signature is a container for top level constants used in the various constraints. The *Clazz*, *Method* and *Instruction* signatures are used to represent the properties of the Java classes, methods and instructions that are being modeled. Each of these three signatures is made abstract in order that each of the individual classes, methods and instructions that make up the bytecodes being analyzed can be defined as concrete, atomic extensions of these abstract signatures. Intuitively this is reasonable because the properties (relations) of instructions (for example) vary from instruction to instruction, but are still static for any particular instruction. For example, the length of a given instruction in bytes is fixed for all time once the instruction is specified, but obviously varies between instructions. The *State* signature is derived from Alloy's ordering utility, which predefines certain relations such as *first*, *next* and *last*. The *State* signature is dynamic, and the values of its relations are updated by the execution engine as it executes during analysis. The Alloy definition of the four primary signatures is shown below.

```
abstract sig Instruction {
    map:         Int,      // offset of this instruction in bytes
    term:   lone Int,      // is this a terminal instruction?
    r:      set  Int,      // local variables read
    w:      set  Int,      // local variables written
    ubt:    lone Int,      // unconditional branch targets (goto)
    cbt:    set  Int,      // conditional branch targets
    jsr:    lone Int,      // unconditional branch target (jsr)
    iv:     lone Int,      // invokevirtual target
    inm:      Method,      // containing method
    smod:        Int,      // bytes pushed/popped onto stack
    len:         Int  }    // byte length of this instruction

abstract sig Method {
    framelen:    Int,      // number of loc vars avail for use
    cpindex:     Int,      // index in constant pool of method
    ordinal:     Int,      // unique identifier
```

```
    acc:     set Int,       // access modifiers, e.g. protected
    incl:      Clazz  }     // class of this method

abstract sig Clazz {
    cordinal:     Int,      // unique identifier
    parent:  lone Clazz }   // parent class or {} if Object

one sig JLO extends Clazz {} { no parent }

sig State {
    meth:        Method,        // currently executing method
    prog:        Instruction,   // currently executing instruction
    readers: set Int,           // set of local vars read
    writers: set Int,           // set of local vars written
    depth:       Int  }         // stack depth
```

An Alloy model is defined by its relations, so a careful description of each of the relations shown above will serve to illuminate the rest of the implementation. The *Instruction* signature is the basic unit of execution. As new states are synthesized by the execution engine, the current instruction advances and the currently executing method may change as well. In the *Instruction* signature the *map* relation defines the byte offset of the instruction from the beginning of the method (or other block of code) being analyzed; it is an integer. The *term* relation is a set of integers that is either empty, or contains a single value. If the set is nonempty and contains the value 1, then the instruction is a terminal instruction: it causes the execution engine to cease creating new states. The *r* and *w* relations model the sets of local variables read or written by the instruction, respectively. It is quite possible for an instruction to access more than one local variable, so these relations must be modeled as sets of integers. (The JVM itself also describes local variables in terms of integers.) The *ubt* and *jsr* relations name possible unconditional branch targets for the instruction, either as the result of a *goto* or a *jsr*, respectively. Most instructions do not have such targets, so the values of these relations is usually the empty set. An instruction can have at most one such target. If a *ubt* target exists, it is specified as a byte offset from the beginning of the method or code block, which is identical to the manner in which it is encoded in a classfile. If, however, a *jsr* target exists, it is specified as a relative byte offset from the current instruction. The *cbt* relation names a possible conditional branch target as an absolute address. Conditional branch targets occur with conditional instructions. An unconditional branch target represents a transfer of control that must be executed, while a conditional branch target represents one that might be executed. The *iv* target represents the argument for an *invokevirtual* instruction. This instruction is used in method resolution, using an algorithm that will be described in more detail below. The argument is an index into the constant pool of the current class; it is expected that this constant will contain a method reference. The *smod* relation models the number of bytes that the instruction modifies on the method stack. This can be a positive integer (item(s) are pushed onto the stack), a negative integer (item(s) are popped off the stack) or zero. The *inm* relation captures the method containing the instruction

in question. Finally, the *len* relation models the length of the instruction in bytes. Note that *len* and *map* contain redundant information, in that it should always be the case that $next.map = current.map + current.len$. This redundancy was introduced deliberately as an additional way of validating the internal consistency of the model.

The *Method* signature is used to describe the properties of a method. It contains a relation that indicate the size of its frame (*framelen*) which, in the JVM, is the total number of local variables that method is permitted to address. It also contains relations that gives its constant pool index (*cpindex*), unique ordinal (*ordinal*) and a relation that holds its containing class (*incl*). Finally, it contains a relation *acc* that is a set of accessor bits for this method. Each of the possible accessor types (public, protected, final, etc.) is encoded as a bit value in the *Global* signature. The *acc* relation must be a set, of course, since a method may have more than one such attribute.

The *Clazz* signature is used to describe the properties of a Java class. For the purposes of the constraint checking that will be described below, it is sufficient to assign each class a unique *cordinal* ordinal and also indicate the *parent* class of the given class. The special signature *JLO* corresponds to the java.lang.Object class, which is the root of the Java class hierarchy.

3.2 Execution Engine

The *State* signature represents the dynamic execution state. Its *prog* relation models the current instruction being executed; its *meth* relation models the current method; its *readers* and *writers* relations model the current set of local variables that have been read or written up to the current program point, respectively, and its *depth* relation models the depth of the stack at the current program point. As the execution engine processes the instruction initializers, it effectively creates new *State* atoms representing the execution state after the effects of the current instruction have been applied.

The execution engine contains the Alloy code associated with State initialization, State sequencing, and execution termination. State initialization code is fixed within the model template. The State initialization code creates an initial state *s0*, sets the *readers* and *writers* relations of *s0* to be empty, sets the *depth* relation of *s0* to be 0, and sets the *prog* relation of *s0* to be the special *startup* instruction. Note that here is no actual JVM instruction named *startup*. However, when the JVM invokes a method it performs certain very specific startup actions (the method prologue) before the first instruction of that method is executed. The pseudo-instruction *startup* captures these actions. Specifically, when the JVM enters an instance method it will set the value of the local variable 0 to be Java's *this* object; if a class method is being called there is no *this* and so 0 is not used for that purpose. If the method has arguments, these arguments are placed in local variables starting at the first unused index. Note also that the *startup* instruction will always have a *depth* relation will be 0. Finally, the *meth* relation of *s0* will be set to the special method *jthis*. This method name is an alias for the method in which execution begins. The initializer for the *startup* instruction must be generated by the classfile parser. By convention, the *startup* instruction is located at a *map* value of -1, and

has length 1.

State transitions and also execution termination are handled by an Alloy fact known as *stateTransition*:

```
fact stateTransition {
    all s: State - ord/last |
        let s' = ord/next[s] |
            ( some t: s.prog.term | t = 1 ) =>
                sameState[s, s'] else
                nextState[s, s'] }
```

The model of execution is that the *nextState* predicate is executed for each nonterminal state. This predicate is responsible for updating the execution state relations (*readers*, *writers* and *depth*) and advancing the instruction (and possibly method) state. This predicate calls the *nextInstruction* predicate, which updates the value of the current instruction and method for *s'*. It updates the *reader* and *writer* relations for the new state *s'* by calling predicates that take the unions of the corresponding *r* and *w* sets from the current instruction *s.prog* with the values of *readers* and *writers* from the current state *s*, respectively. Finally, it updates the *depth* relation for *s'* by adding the *smod* value of the current instruction to the *depth* in the current state *s*.

3.3 Security Constraints

The Alloy model template attempts to capture some of the high value JVM security constraints checked by the Bytecode Verifier. Thus we must now define what constitutes a high value constraint. There are many avenues of attack that can be used by malicious code. Common attack methods include the use of uninitialized memory, accessing out-of-bounds memory, functional redirection (causing the code to execute a method other than the one that was specified) and instruction redirection (causing a transfer to control to any location other than the beginning of a valid, in-scope instruction). In the JVM, these abstract attack methods can be expressed in terms of concrete constraints, and from there can be written as Alloy assertions. Specifically, it should never be possible to read a local variable that has not been written; it should never be possible to access a local variable with an index less than zero or greater than the frame length of the current method. The first case would correspond to accessing uninitialized memory, while the second and third cases would correspond to accessing out-of-bounds memory. The following Alloy assertions are used to verify these constraints:

```
assert LocalVar { all s: State | s.readers in s.writers }

assert rInRangelow { all s: State, u: s.readers | gte[u, 0] }

assert rInRangehigh { all s: State, u: s.readers |
                      lte[u, s.meth.framelen] }

assert wInRangelow { all s: State, u: s.writers | gte[u, 0] }
```

```
assert wInRangehigh { all s: State, u: s.writers |
                      lte[u, s.meth.framelen] }
```

It is straightforward to interpret these constraints. The *LocalVar* constraint, for example, says that for all States, the set of local variables read must be a subset of the set of local variables written. The *rInRangelow* constraint says that for all states and for all local variables indices in the *readers* relation of that state, it must be the case that each index is greater than or equal to 0. (Of course if the *LocalVar* constraint is satisfied the subsequent constraints on the readers are subsumed in the corresponding constraints on the writers.)

It should also be possible to make similar assertions about the stack. In the JVM the stack is not used to carry method arguments or return values, so that stack is, in fact, purely local to a method. It must always be the case that the depth of the stack is non-negative. Further, it must always be the case that the stack depth at any program point is invariant, regardless of how that program point is reached. The notorious BlackBox applet, for example, violates this latter stack depth invariance constraint as part of its exploitation methodology. These two stack constraints can be expressed in Alloy as:

```
assert StackGTE { all s: State | gte[s.depth, 0] }

assert StackDepth {
        all s, s' : State | (s.prog.map = s'.prog.map) =>
                            (s.depth = s'.depth) }
```

Instruction transfer constraints insure that when the program flow of control is changed by the execution of a conditional or unconditional branch, the program counter is changed to a byte offset that matches the beginning of an instruction. If a transfer could be arranged into the middle of an instruction, or into uninitialized or out-of-bounds memory, then arbitrary code could be executable. This is a security defect that is often exploitable. In the current model of an *Instruction* there are three possible branching relations: *ubt*, *jsr* and *cbt*. Note that *ubt* and *cbt* name absolute offsets, while *jsr* names a relative offset. We must therefore write Alloy constraints that insure that the computed target locations correspond to the byte offset of exactly one *Instruction*. These constraints are written as follows:

```
assert InstructionTransfer_abs {
       all ins : Instruction, u: ins.ubt |
          one bti: Instruction { bti.map = u } }

assert InstructionTransfer_cabs {
       all ins : Instruction, c: ins.cbt |
          one bti: Instruction { bti.map = c } }

assert InstructionTransfer_rel {
       all ins : Instruction, u: ins.jsr |
          one bti: Instruction { bti.map = add[ins.map, u] } }
```

The final set of high value constraints is associated with virtual method invo-

cation. In Java virtual method invocation involves the execution of an instance method in which the dispatch is based on the Java class that contains an actual matching implementation of the method. In the JVM instruction set this is handled by the *invokevirtual* instruction. The *invokevirtual* instruction takes a index into the constant pool of the current class as its argument. The index must be a valid index into the constant pool, and must refer to a fully qualified method. If the method reference's class contains a resolvable method of that name and signature, or a compatible signature that can be reached by the usual rules of widening, boxing and unboxing, then the method lookup procedure terminates. If one or more of these conditions is not realized, then the superclass of the method reference's class is consulted recursively until resolution terminates successfully; otherwise a Java exception is raised. The resolved method must not be an initializer (constructor) or a class initializer, and must not be static or abstract. Finally, if the method in question has the protected attribute, it must be in the same class as the class containing the method which called *invokevirtual.* Insuring that JVM bytecodes satisfy all the stated constraints is critical for security. If *invokevirtual* can be coerced into executing a method with inappropriate attributes, for example, this could lead to the execution of arbitrary bytes, information disclosure (by virtue of exposing private methods or variables) or denial of service (by causing a JVM crash through the execution of non-viable code). We separate the constraint checking into two separate Alloy assertions:

```
assert invokevirtual {
    all i: Instruction, iv: i.iv | one m: Method {
        m.cpindex = iv && m.acc - Globals.ACC_STATIC = m.acc &&
                          m.acc - Globals.ACC_ABSTRACT = m.acc &&
        m.ordinal != init__.ordinal &&
        m.ordinal != clinit__.ordinal }  }

assert invokevirtual2 {
    all i: Instruction, iv: i.iv | one m: Method {
        (m.acc - Globals.ACC_PROTECTED != m.acc) =>
         m.incl = i.inm.incl } }
```

The first constraint asserts that for all *Instructions*, if the *iv* relation of that instruction is non-empty, then there is exactly one *Method* such that the constant pool index of that method is the same as the argument to the instruction (which must, perforce, be the *invokevirtual* instruction), the *acc* accessor bitfield of that method contains neither the **static** attribute nor the **abstract** attribute, and the ordinal for that method does not match the ordinal for an initializer or class initializer. The second constraints covers the case of the **protected** attribute. It states that if the matching method has the protected attribute, then the containing class of the matching method must be the same as the containing class of the method that contains the *invokevirtual* instruction being analyzed.

3.4 Class2Alloy Classfile Parser

The model template is not a complete Alloy model in that it does not encode any property information of actual JVM instructions, methods or classes. That encoding is handled by the relation initializers, which must initialize all the instruction, method and class relations based on the bytecodes of a specified methods. The initialization must also handle the creation of concrete signatures that extend the abstract *Instruction*, *Method* and *Clazz* signatures. These concrete signatures are based on exactly those instructions that are in the specified methods, their containing classes, and the class ancestors of those containing classes.

A classfile parser, known as Class2Alloy, was written to generate these Alloy relation initializers given a Java classfile and also a list of method names. Class2Alloy was implemented in Java using the Byte Code Engineering Library, BCEL [12]. BCEL is an extremely powerful classfile analysis library that provides ready access to the instruction stream in Java classes. BCEL makes it straightforward to extract the requisite properties for each instruction, method or class under consideration.

Class2Alloy is implemented in two Java files, Class2Alloy.java and AlloyString.java. Class2Alloy.java contains the main analysis routines, while AlloyString.java is a utility class that handles the specific Alloy syntax needed to generate syntactically correct relation initializers. The operation of the parser is as follows. The *main* method receives three arguments: the name of a classfile, which must be in the classpath, a list of method names, and the name of an output file. The *main* method creates a Class2Alloy instance; the Class2Alloy constructor creates a set of empty AlloyStrings, one for each relation to be initialized, along with an empty AlloyString that will hold the signature information. BCEL is then used to load the classfile, enumerate its methods, and search for the named methods in the array of methods; on success an array of BCEL *Method* objects is obtained. Class2Alloy then parses these *Method* objects to obtain a list of instructions contained within the methods. For each instruction, it then queries that instruction for those properties that need to be initialized in the Alloy model, namely its byte offset from the beginning of the method, its byte length, the sets of local variables that it reads or writes, the set of possible conditional or unconditional branches that it can take, and also the number of bytes that it adds or removes from the stack. If an *invokevirtual* instruction is encountered, a method lookup is performed on the constant pool index it names. This may cause other classfiles to be loaded recursively in order to obtain the properties for the referenced methods and classes. Once the instruction analysis is complete, each AlloyString prints itself to the output file. The AlloyString class handles the details of generating syntactically correct Alloy output for each of the relation initializers, as well as generating the appropriate extension signatures for each instruction, method and class being analyzed.

Once this output file is combined with the model template, a complete model specialized for the methods under analysis is obtained. The Alloy analyzer is then run on that model, and each of the constraint assertions is invoked to determine the presence of counterexamples. Any counterexample represents a violation of one or more constraints. Any constraint violation is an indication of insecure, and potentially exploitable, JVM bytecodes. The approach in question has been tested extensive and produces no false positives. In addition, when run against known ma-

licious applets, such as the BlackBox applet, counterexamples are detected. Several malicious applets were synthesized to illustrate invalid memory accesses, instruction diversion and function diversion. All were detected.

4 Future Work

There are several areas in which the JVM security analysis approach described in this paper can be extended and improved. The most obvious, and certainly the most important, is to add constraint checking for additional constraints. The opcode argument constraint, which states that each JVM instruction is invoked with the correct number of type conforming arguments, is of particular importance. The JVM instruction set contains three other instructions involved in method invocation, namely *invokestatic*, *invokespecial* and *invokeinterface*. The constraint checking shown above for *invokevirtual* should be extended to handle these instructions as well.

The current model does not completely handle exceptions. In particular, only a single exception block per method is currently modeled, while actual bytecode can employ multiple (nested) exception blocks. Adding full exception handling to the model has high priority. This is an ongoing area of research.

Finally, the JVM is not the only bytecode interpreted language that could be subjected to this form of security analysis. CIL and Flash are obvious candidates for constraint based model checking for security purposes.

5 Conclusion

This paper has demonstrated that Alloy is an extremely powerful tool for performing security constraint analysis on Java bytecodes. Even at this stage of development, meaningful results have been obtained. Extensions to this work are ongoing, with the goal of increasing the scope of constraint checking and further refining and improving the analysis process. Extensions to other languages are also in work.

References

[1] Alloy website, http://alloy.mit.edu

[2] Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT Press, Cambridge (2006)

[3] Reynolds, M.: Lightweight Modeling of Java Virtual Machine Security Constraints. ABZ2010 conference, to appear.

[4] McGraw, G., Felten, E.: Securing Java: Getting Down to Business with Mobile Code 2nd Edition. Wiley, New York (1999)

[5] Common Vulnerabilities and Exposures, http://cve.mitre.org

[6] BlackBox Security Advisory, http://www.ca.com/us/securityadvisor/virusinfo/virus.aspx?ID=36725

[7] Java and Java Virtual Machine security vulnerabilities and their exploitation techniques, http://www.blackhat.com/presentations/bh-asia-02/LSD/bh-asia-02- lsd.pdf

[8] Lindholm, T., Yellin, F.: The Java Virtual Machine Specification Second Edition. Addison Wesley, Boston (2003)

[9] Xu, H.: Java Security Model and Bytecode Verification, http://www.cis.umassd.edu/ hxu/Papers/UIC/JavaSecurity.PDF

[10] Posegga, J., Vogt, H.: Java bytecode verification using model checking, http://eprints.kfupm.edu.sa/47269

[11] Leroy, X.: Java Bytecode Verification: An Overview. Proceedings of the Thirteenth International Conference on Computer Aided Verification, p. 265–285, 2001.

[12] Jakarta BCEL, http://jakarta.apache.org/bcel