Lightweight Modeling of Java Virtual Machine Security Constraints

Mark C. Reynolds

Boston University
Department of Computer Science
markreyn@cs.bu.edu

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 component known as the Bytecode Verifier. 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 a real world malicious applet is given to demonstrate the efficacy of the approach. This type of analysis represents a significant departure from standard malware detection methods based on signatures or anomaly detection.

Key words: Alloy, JVM, 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. A review of the top level goals of this work will then be presented, followed by a description of the design of the analysis tool and its implementation. Results will then be presented in detail using a real world example of malicious code: the BlackBox applet. 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. The approach of applying lightweight modeling as a means to check JVM security constraints appears to be a novel approach.

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 [3] 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 [4] lists numerous Java bugs that can lead to privilege escalation, sensitive data exfiltration, denial of service and other malicious outcomes. Of particular note is the BlackBox malicious Java applet [5], [6]. This applet exploits a number of Java security weaknesses, and was widely deployed, infecting thousands of machines. The BlackBox applet not only breaks out of the supposedly inescapable sandbox that the Java applet runtime imposes, it also manages to escalate its privilege to the highest possible level. The BlackBox applet can be easily customized to download any program to the infected machine and then run it. This applet is thus not only an exploit in itself; it is also a delivery vehicle for an arbitrary malicious payload. The BlackBox applet will be analyzed using the methodology presented in this paper.

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 succeed. 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 [7]. 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 [8], [9], [10], 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.

1.2 Goals

This work described in this paper has three goals: (1) to provide an extensible framework for modeling security constraints imposed by the JVM's Bytecode Verifier; (2) to provide a concrete model for meaningful, high value security constraints, and (3) to demonstrate that the analysis tool does check them correctly.

It would be straightforward to use Alloy to create a model for a specific block of Java bytecode. While this might serve as the demonstration of the applicability of Alloy to security analysis of the JVM, this would have little value in analyzing compliance with the JVM security constraints as a whole. Therefore, it is desirable to have an extensible model. In this context "extensible" means that the model must have the ability to be applied to any block of JVM code and to perform analysis on that code against a specified set of constraints. In the **Design** section it will be shown how this goal was realized.

Several of the security constraints imposed by the JVM have already been mentioned. In general, most constraints are independent of one another, although there are some functional overlaps, as will be demonstrated below. In order to prove the soundness of the basic concept, it was deemed prudent to select a realistic subset of the total set of JVM security constraints and begin with a simple model that would encompass that reduced subset of constraints. With the extensibility goal in mind, 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 a small, but critical, set of security constraints. 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.

The initial design problem was to find an "implementation" of the Alloy model that would capture the invariants of interest abstractly, independent of

any actual JVM code, but would then permit the model to be run against any concrete realization of such JVM code. Initial experimentation with Alloy suggested two possible approaches: automatically generate Alloy functions, facts or predicates based on the JVM code to be analyzed, or automatically generate Alloy statements that initialize relations based on the JVM code to be analyzed. In order to realize a classical code/data separation, it was decided to use the latter approach. Thus, the Alloy model would be realized as a template containing a fixed set of relations, functions, facts, predicates and assertions. This model would then be supplemented by relation initializers that would be derived from particular JVM code. In this approach, the template portion of the Alloy model would be completely independent of any choice of Java bytecodes, while the initializers would depend only weakly on the detailed implementation of the template. Specifically, the initializers being generated would 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 is the means by which the stated extensibility goal has been achieved.

Further requirements analysis revealed that these two top level components, the model template and the initializers, could be further refined into four subcomponents: (1) the relation definitions; (2) the relation initializers; (3) the execution engine; and (4) the constraint assertions. The relation definitions, execution engine and constraint assertions are all part of the Alloy model template. The relation definitions are Alloy definitions of the top level signatures, which contain relations, as well as the definitions of the relations themselves. These relation definitions capture the static properties of individual JVM instructions, 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, and vary from one invocation of the model to the next. An initial design decision was made to capture JVM code at the method level. This, of course, is a trade off between performance and granularity. It is certainly possible to model multiple methods within a single model. However, the time that Alloy takes to analyze a particular model is strongly dependent on the number of (program execution) states, which, in turn is strongly dependent on the size of the relation initializers. As will be seen below, the actual Alloy model template is quite suited to analyzing code blocks within a method, and could be extended to handle multiple methods. Relation initializers need to be generated from specific Java methods. Therefore, there needs to be an automatic way of converting the Java bytecodes in a method 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.

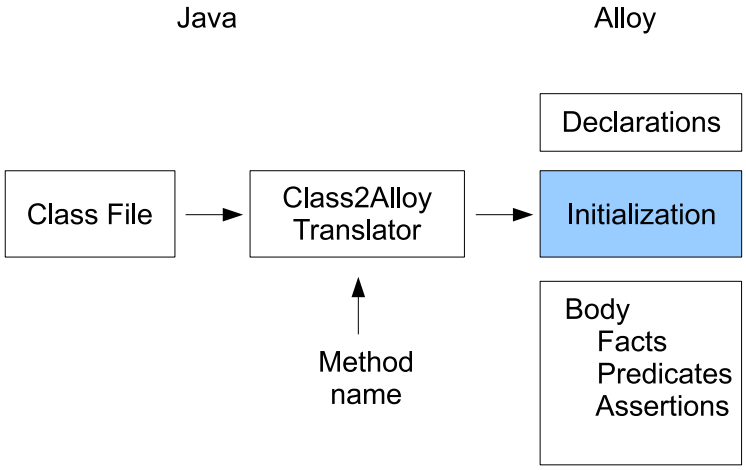

Fig. 1. Constructing a complete Alloy model using the classfile parser

The relation definitions and their initializers form a static representation of a set of properties of the Java method 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 would represent 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. (Any type of *return* instruction would be terminal, for example.) The execution engine was then implemented to recognize this condition and act on it in such a way as to create no 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 pre-

cise 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 be described in three subsections. In the first subsection, the three components of the model template, namely the relation definitions, the execution engine, and the security constraint assertions, will be described. In the second subsection, the implementation of the Class2Alloy classfile parser which is used to generate the relation initializers will be discussed. In the third subsection a concrete example will be dissected, including a description of the parser invocation and subsequent model analysis. The example in question is a reduced form of the BlackBox applet.

3.1 Model Template

The model template employs two top level signatures, an *Instruction* signature and a *State* signature. The *Instruction* signature is made abstract in order that each of the individual instructions that make up a method can be defined as concrete, atomic extensions of this abstract signature. Intuitively, this is reasonable because the properties (relations) of instructions 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 these two 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 by this instruction
    w:      set  Int,       // local variables written by this instruction
    ubt:    lone Int,       // unconditional branch targets
    cbt:    lone Int,       // conditional branch targets
    smod:        Int,       // bytes pushed/popped onto the stack
    len:         Int  }     // byte length of this instruction

sig State {
    prog:        Instruction,
```

```
    readers: set Int,
    writers: set Int,
    depth:       Int  }
```

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. 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* relation names a possible unconditional branch target for the instruction. Most instructions do not have such a target, so the value of this relation is usually the empty set. An instruction can have at most one such target. If such a 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. The *cbt* relation names a possible conditional branch target. 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. Note that in the JVM it is possible for a conditional branch instruction to have multiple targets, but for simplicity this is not currently modeled. 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. 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, as will be described shortly. The output of the translator acting on a simple method is shown below.

```
one sig startup, iload_1_1, bipush_2, if_icmpge_3, iload_1_4,
        iconst_5_5, imul_6, istore_2_7, goto_8, iload_1_9,
        istore_2_10, iload_2_11, ireturn_12 extends Instruction {}

fact maps { map = startup->(-1) + iload_1_1->0 + bipush_2->1 +
        if_icmpge_3->3 + iload_1_4->6 + iconst_5_5->7 + imul_6->8 +
        istore_2_7->9 + goto_8->10 + iload_1_9->13 +
        istore_2_10->14 + iload_2_11->15 + ireturn_12->16 }

fact lens { len = startup->1 + iload_1_1->1 + bipush_2->2 +
        if_icmpge_3->3 + iload_1_4->1 + iconst_5_5->1 + imul_6->1 +
        istore_2_7->1 + goto_8->3 + iload_1_9->1 + istore_2_10->1 +
```

```
        iload_2_11->1 + ireturn_12->1 }

fact rs { r = iload_1_1->1 + iload_1_4->1 + iload_1_9->1 +
        iload_2_11->2 }

fact ws { w = startup->0 + startup->1 + istore_2_7->2 +
        istore_2_10->2 }

fact ubts { ubt = goto_8->15 }

fact cbts { cbt = if_icmpge_3->13 }

fact terms { term = ireturn_12->1 }

fact smods { smod = startup->0 + iload_1_1->1 + bipush_2->1 +
        if_icmpge_3->(-2) + iload_1_4->1 + iconst_5_5->1 +
        imul_6->(-1) + istore_2_7->(-1) + goto_8->0 + iload_1_9->1 +
        istore_2_10->(-1) + iload_2_11->1 + ireturn_12->(-1) }
```

The *State* signature represents the dynamic execution state. Its *prog* relation models the current instruction being executed; 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 a method it will set the value of the local variable 0 to be Java's *this* object. If the method has arguments, these arguments are placed in local variables starting at index 1. Thus, the *startup* instruction will always have a nonempty value for its w relation; its r relation will be empty and the *depth* relation will be 0. 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. The *nextState* predicate is shown below. This predicate is responsible for updating the execution state relations (*readers*, *writers* and *depth*) and advancing the instruction state. This predicate calls the *nextInstruction* predicate, which updates the value of the current instruction 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*.

```
pred nextState[s, s': State] {
    nextInstruction[s.prog, s'.prog]
    nextReader[s.prog, s.readers, s'.readers]
    nextWriter[s.prog, s.writers, s'.writers]
    (s'.depth = add[s.depth, s.prog.smod]) }
```

The *nextInstruction* predicate calculates the next instruction for the state *s'* as follows. If the current instruction has an unconditional branch target, as indicated by the fact that the current instruction's *ubt* relation is not empty, then the unconditional branch is taken. The next instruction is the one whose *map* value (byte offset) matches the value of the *ubt* for the current instruction. This raises the interesting possibility that the JVM bytecodes might be sufficiently damaged that the *ubt* relation pointed to a *map* value that was not represented by any instruction, e.g. that the *ubt* pointed to the middle of an instruction, or outside the method entirely. This internal consistency constraint is checked by the *Terminates* predicate described below.

If there was no unconditional branch target, but there was a conditional branch target, then Alloy can choose to take that branch, or it can instead simply go to the next instruction by adding the current value of the *map* relation to the length of the current instruction. It will also perform this latter action in case there are no branches of either type. Note that the current model of exception handling treats an exception as a possible conditional branch for the entire range of instructions protected by a particular handler; the conditional branch target is the beginning of the handler code.

```
pred nextInstruction[from, to: Instruction] {
    some from.ubt =>
        ( to.map = from.ubt ) else
        ( ( to.map = add[from.map, from.len] ) ||
          some bt: from.cbt { to.map = bt } ) }
```

The Alloy model template captures some of the JVM security constraints checked by the Bytecode Verifier. The security constraints being checked are the local variable constraint, the stack depth invariance constraint, the stack guard constraint, the branch consistency constraint and the instruction length constraint. The local variable constraint states that no local variable can be read until it has first been written. The purpose of this constraint is to avoid accessing uninitialized local variables. The Java compiler enforces this constraint at the source code level for any variable (not just those that end up being stored in JVM local variables), and the Bytecode Verifier checks it at the classfile level. The stack depth invariance constraint states that the depth of the stack will always be the same at any program point, no matter how that program point was reached. The stack depth invariance constraint is one of the constraints that is violated by the BlackBox applet, and will be discussed further below. The stack guard constraint is actually an amalgam of several closely related constraints. It states that the depth of the stack never becomes negative, and also that it should be zero on method entry and on any branch that leads to method exit, which is at any terminal instruction for the method. This latter constraint is a critical constraint for the JVM architecture. Unlike the architectures of many real machines, the JVM does not use the stack to pass parameters or return values; local variables are always used for both. Thus, the state of the stack (empty) should be the same on exit as on entry for every method. Each of the constraints corresponds to a single Alloy assertion. The branch consistency constraint and the instruction length constraint are different forms of the same consistency check, namely that neither normal flow of execution nor execution of any branch can put the JVM into a state in which it is not at an instruction boundary. Finally, there is also a special predicate that performs consistency checks on the model. The assertions and the predicate are:

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

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

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

pred Terminates {
    some finalState: State | finalState.prog.term = 1 }
```

Note that each of the constraints is expressible in a single Alloy statement. The local variable constraint, *LocalVar*, asserts that for all states, the set of integers in the *readers* relation must be a subset of the set of integers in the *writers* relation. Since a state has a one-to-one correspondence with a program point (except for the special state that has *startup* as its instruction) this exactly expresses the local variable constraint. The stack depth invariance constraint, *StackDepth*, asserts that for any pair of states *s* and *s'* that have the same

program point *(s.prog.map = s'.prog.map)* the depth of the stack must be the same *(s.depth = s'.depth)*. The stack guard constraint, *StackGTE*, asserts that for all states the corresponding stack depth must be greater than or equal to zero. These constraints are positive constraints: if Alloy finds a counterexample this demonstrates that the constraint has been violated. A violation of the constraint then indicates that the corresponding JVM code does not conform to the classfile standard, and contains buggy or potentially malicious bytecode. It is worthwhile to observe, however, that the existence of nonconforming bytecode does not necessarily imply that the resulting code is exploitable.

The *Terminates* predicate bears closer examination, since it relates to the handling of looping constructs and also internal consistency checking. This predicate asserts that there is some state with an instruction that is terminal. In effect, this predicate asserts that execution terminates for some set of branch choices. When faced with a conditional branch choice, Alloy will choose a possibility. Thus, if there is any path to a terminal instruction, it will be reached by some set of choices by Alloy (provided the search space is large enough). What conditions could cause this predicate to fail? One case would be the case of an unconditional branch whose target is an earlier program point corresponding to an unambiguous infinite loop. Another situation that would cause this predicate to fail is if the *map* and *len* relations are not internally consistent. Examination of the *nextInstruction* predicate shows that if there are no branches the instruction in the next state is calculated from the instruction in the current state by adding the length of the current instruction (the *len* relation) to the byte offset of the current instruction (the *map* relation). Alloy must then find a matching instruction whose offset (*map* relation) is equal to this sum. If no such instruction exists, then the *nextInstruction* predicate will return false and the *Terminates* predicate will never be satisfied. The *Terminates* predicate therefore also provides a test of the internal consistency of the *map* and *len* relations, and thus also indirectly checks the constraint that asserts that the JVM can never reach a program point that is not at an instruction boundary.

3.2 Class2Alloy Classfile Parser

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

A classfile parser, known as Class2Alloy, was written to generate these Alloy relation initializers given a Java classfile and also a method name. Class2Alloy was implemented in Java using the Byte Code Engineering Library, BCEL [11]. 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 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, the name of a method, 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 instruction signature information. BCEL is then used to load the classfile, enumerate its methods, and search for the named method in the array of methods; on success a BCEL *Method* object is obtained. Class2Alloy then parses this *Method* object to obtain a list of instructions contained within the method. 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. 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 in the method being analyzed.

Once this output file is combined with the model template, a complete model specialized for the method under analysis is obtained. The Alloy analyzer is then run on that model, and each of the constraint assertions, as well as the *Terminates* consistency predicate, is invoked to determine the presence of counterexamples or a failure to converge to a terminal state.

3.3 Analysis of the BlackBox applet

The BlackBox applet is a malicious applet that breaks out of the applet execution sandbox, elevates its privilege level to the maximum possible value, and then downloads (and optionally executes) a completely arbitrary payload. The BlackBox applet uses a variety of exploitation techniques in order to achieve its goals. A complete description of the workings of this applet is beyond the scope of this paper; instead a reduced version will be described. It is very important to note that both the reduced version and the complete version of BlackBox are detected by the technique described herein.

The Java Virtual Machine loads classes using a series of helper classes known as classloaders. The classloader class responsible for loading classes across the network is the *URLClassLoader* class. *URLClassLoader* not only verifies classfile syntax, it works closely with the *SecurityManager* class to check for permitted or forbidden operations. As one might readily imagine, most of the methods and members of *URLClassLoader* are either protected or private. If it were possible to define a class (call it *myUCL*) that had the same methods and members as *URLClassLoader*, but which permitted all operations and did not consult the

SecurityManager, one could then load and instantiate an arbitrary network class. This type of exploit is known as a type confusion exploit: an object of one class (*URLClassLoader*) is replaced by an object of another class (*myUCL*) without triggering a type coercion exception.

One way to cause type confusion to occur is to violate the stack depth invariance constraint. Suppose that an object of type *myUCL* is placed on the stack, followed by an object of type *URLClassLoader*. Suppose further than in the normal flow of execution the *URLClassLoader* object will be popped off the stack and loaded into a local variable of that type. If the flow of control can be modified such that the stack depth invariance constraint is violated, and such that the *URLClassLoader* object is popped off the stack while leaving the JVM state unmodified, then a subsequent stack operation will pop the *myUCL* object off the stack and treat it as if it were an object of type *URLClassLoader*. One approach for doing this was through the use of malicious exception handling code. When an exception is thrown in Java, an exception object is placed on the stack so that the exception handler code may access it. After the handler completes, the state of the stack is supposed to be restored to the state it had just before the exception was thrown. If it can be arranged that the malicious exception code actually modifies the stack such that two objects are popped when the handler exits, the stack depth invariance constraint will be violated. The net effect in the actual BlackBox applet is that an object of type *myUCL* is substituted for an actual *URLClassLoader* object; this object is then used to load malicious code over the network and execute it. (As stated above, the actual process is significantly more complicated.)

More than a hundred benign applets were subjected to analysis using the Alloy analyzer; no stack depth invariance constraint violations were found. However, when the bytecodes from the full or reduced version of BlackBox were analyzed, a violation of the stack guard invariance constraint was detected. The approximate time to run the Alloy analyzer and find this counterexample was three minutes. Detailed security analysis with a Java disassembler (such as [12]) then revealed that a type confusion attack was being launched by this applet. Note that the applet itself could not have been compiled directly from Java; the malicious portions, in particular those portions handling the type confusion attack, had to have been constructed using a Java assembler, such as Jasmin [13].

3.4 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. In addition, extending the current consistency checking of the *map* and *len* properties is also a worthwhile step, since the current model does not distinguish the two possible cases in which the *Terminates* predicate fails to

converge, namely infinite loops versus an inconsistent set of *map* and *len* relation values. The latter should be checked explicitly.

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.

4 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.

Acknowledgments The author wishes to extend special thanks to Assaf Kfoury of the Computer Science Department of Boston University for suggesting this line of inquiry, for his valuable input, and for his continued support and encouragement.

References

1. Alloy website, `http://alloy.mit.edu`
2. Jackson, D.: Software Abstractions: Logic, Language and Analysis. MIT Press, Cambridge (2006)
3. McGraw, G., Felten, E.: Securing Java: Getting Down to Business with Mobile Code 2^{nd} Edition. Wiley, New York (1999)
4. Common Vulnerabilities and Exposures, `http://cve.mitre.org`
5. BlackBox Security Advisory, `http://www.ca.com/us/securityadvisor/virusinfo/virus.aspx?ID=36725`
6. 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`
7. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification Second Edition. Addison Wesley, Boston (2003)
8. Xu, H.: Java Security Model and Bytecode Verification, `http://www.cis.umassd.edu/~hxu/Papers/UIC/JavaSecurity.PDF`
9. Posegga, J., Vogt, H.: Java bytecode verification using model checking, `http://eprints.kfupm.edu.sa/47269`
10. Leroy, X.: Java Bytecode Verification: An Overview. Proceedings of the Thirteenth International Conference on Computer Aided Verification, p. 265–285, 2001.
11. Jakarta BCEL, `http://jakarta.apache.org/bcel`
12. DJ disassembler, `http://members.fortunecity.com/neshkov/dj.html`
13. Jasmin assembler, `http://jasmin.sourceforge.net`