Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Science of Computer Programming 00 (2010) 1–20
Science of Computer Programming

Modeling the Java Bytecode Verifier[☆]

Mark C. Reynolds

Department of Computer Science
Boston University

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 these bytecodes into relation initializers in Alloy. Any counterexamples found by the Alloy analyzer correspond directly to potentially insecure code. Analysis of the approach is provided in the context of known security exploits, including type confusion attacks, invalid memory accesses and control flow misdirection. This type of analysis represents a significant departure from standard malware analysis methods based on signatures or anomaly detection.

Keywords:
Java security, JVM bytecode, Alloy, lightweight modeling

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], [3]. This paper will describe the security verification approach taken by the Java Virtual Machine (JVM), and 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 Alloy model. The goal of this effort is to find instances where actual Java bytecodes violate one or more security rules in the JVM specification. If those bytecodes are also accepted by one or more actual implementations of the JVM, this represents a potential security vulnerability. It should be the case that an implementation of the JVM checks its security rules when a classfile is loaded, but real exploits have been developed, and continue to be developed, where that is not the case. Results will be presented in detail using a real world example of malicious code: the BlackBox applet. The work described in this paper is a significant extension of [4], which first described the analysis method that was conducted

[☆]Special thanks to Assaf Kfoury of the BU CS Dept for suggesting this line of inquiry and for his continued support

Email address: `markreyn@cs.bu.edu` (Mark C. Reynolds)

on 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, as it is capable of detecting several forms of malicious bytecodes. The approach of applying lightweight modeling as a means for checking JVM security constraints appears to be novel, and differs significantly from standard malware detection approaches based on signatures or anomaly finding [18].

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 [5] 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 [6] lists numerous Java bugs that can lead to privilege escalation, exfiltration of sensitive data, denial of service and other malicious outcomes. In 2002 the BlackBox malicious applet [2] was released, infecting thousands of machines. This applet exploited a number of Java security weaknesses, including a type confusion attack that will be described more fully in this paper. The BlackBox applet not only broke out of the supposedly inescapable applet sandbox, it also managed to escalate its privilege to the highest possible level. The BlackBox applet was easily customized, and several dozen variants were eventually running in the wild. Even worse, the applet was not only an exploit in itself, it was also the delivery vehicle for an arbitrary malicious payload. More recently, an announcement [20] (November 2009) contained more than ninety previously unpublished Java vulnerabilities, several of which were already represented by actual exploits running on the Internet.

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.

Many Java exploits to date [7], [8] have used weaknesses in the Bytecode Verifier implementation. The JVM specification [9] describes in great detail the security rules that the Bytecode Verifier is supposed to verify when it performs its checks. The Bytecode Verifier attempts to check each of these rules when 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. It is also worth noting that the Java runtime typically does not recheck the constraints checked by the Bytecode Verifier, so if its rules can be bypassed then malicious access becomes possible. The key issue that arises revolves around the question of what it is that the Bytecode Verifier is actually checking. Even if one assumes that the JVM specification is both internally consistent and also complete, there can still be issues if the Bytecode Verifier does not comprehensively check each rule given in the specification. The exploits that have been released, including the BlackBox exploit, have consistently used gaps between the specification of the security rules and the actual set of rules that are checked by the Bytecode Verifier implementation.

Our approach uses Alloy to perform constraint analysis on Java bytecodes. It attempts to check the rules as given in the JVM specification. Thus, if the Bytecode Verifier were perfect, our approach would emulate the constraint checking that is (ostensibly) being performed by the Bytecode Verifier. Our method analyzes actual bytecodes in classfiles against the specification. If it can be shown that our method finds bytecodes that violates the specification, as encoded in Alloy, but that the corresponding classfile passes the checks employed by some implementation of the Bytecode Verifier, then a potential security vulnerability has been exposed. 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 [10], [11], [12],

[13], [14], [15], [16], [17] 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. In addition, no previous work has attempted to use a constraint analyzer for the purpose performing a dynamic security analysis of Java bytecodes.

It would be straightforward to create an Alloy model for a specific block of bytecodes. This model could then be used to look for counterexamples to the security specification within that particular set of bytecodes. This is hardly a fully general approach, however. One goal of the current work, therefore, is to provide a framework in which any set of Java methods can be analyzed using an extensible model. As will be described in greater detail below, this implies that the model will have two parts, a template specifying the constraints that represent security rules, and a set of specific Alloy statements associated with the set of bytecodes being analyzed.

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 (but not necessarily Java 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 in this work the Alloy model is realized as a template containing a fixed set of relations, functions, facts, predicates and assertions. This template is then supplemented by relation initializers that are derived from particular JVM code being analyzed. 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 and methods, 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 assign 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 from 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

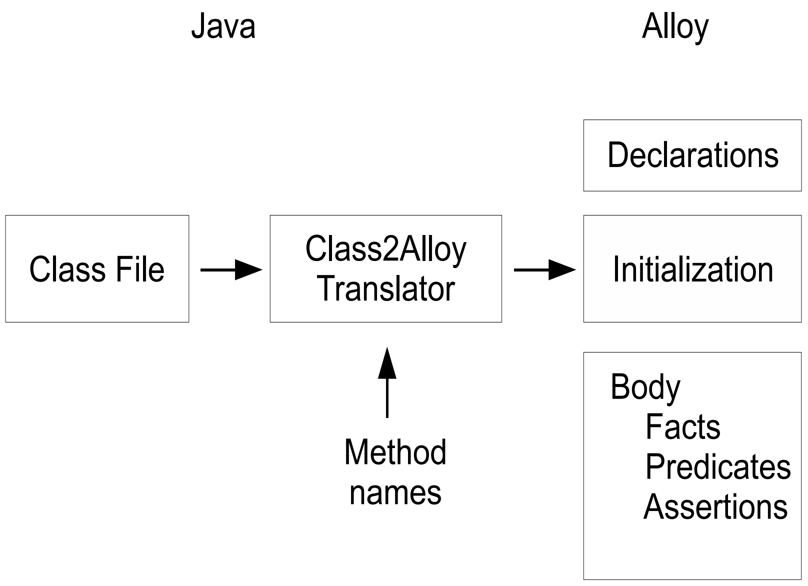

Figure 1: Analyzer components

synthetically executes method code in order to perform its own 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 to express the logic associated with state transitions. 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 for a method. The execution engine was then implemented to recognize this condition and act on it so as to not create further unique states within the context of the method. 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 template 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 four top level signatures, the *Global* 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 *Method* and *Instruction* signatures are used to represent the properties of the Java methods and instructions that are being modeled. Each of these signatures is declared abstract in order that each of the individual 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 dynamic, and the values of its relations are updated by the execution engine, using Alloy's ordering utility (which predefines certain relations such as *first*, *next* and *last*) as it executes during analysis. The Alloy definition of the three primary signatures is shown below. Since the *Global* signature definition is not central to this discussion, it is deferred to Appendix C.

```
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 virtual method target
    inm:          Int,       // ordinal of 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

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 for that method. 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. Many methods do not have such targets, so the values of these relations is often 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. If *current* and *next* denote the current instruction and the next instruction, respectively, 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*), and a unique ordinal (*ordinal*). 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. Consider the following very simple method:

```
public static void main(String[] args) {
    System.out.println("Hello");
    double d = 4.0;
    try {
      JavaClass clazz = Repository.lookupClass("hello");
      System.out.println(clazz);
    } catch (Exception e) {
        e.printStackTrace();
        d += e.getMessage().hashCode();
    } finally {
        if ( d > 5 )
            System.out.println("D = " + d);
    }
}
```

The JVM bytecodes for this method are shown in Appendix A. The complete set of Alloy initializers generated when the translator acts on the bytecodes for this method are shown in Appendix B. The first and second sets of initializers

```
one sig startup, getstatic_1, ldc_2, invokevirtual_3, ldc2_w_4,
...
extends Instruction {}

one sig jthis, init__, clinit__,
...
extends Method {}
```

create an Alloy signature for each instruction and each method in the classfile being analyzed. Note that for each instruction, its actual name has a unique, incrementing ordinal appended. This allows the model to distinguish the instance of the *invokevirtual* instruction that appears at one location in the byte stream from that same instruction appearing at another location in the byte stream. For methods this approach is not necessary, since we are only interesting in recording the names of all methods that are referenced in the classfile, not each unique occurrence of such references.

The *map* and *len* relations record the byte offset of each instruction and the length of each instruction in bytes, respectively. Thus, we can see immediately that there is a *getstatic* instruction at offset 0 in the classfile, with length 3. This is followed by a *ldc* instruction at offset 3 (as one would certainly hope, this the previously instruction was three bytes long and started at offset zero), and length 3. Each instruction in the method under analysis has its offset and length recorded in these initializers. (Note that it is possible to use the approach described in this paper to analyze multiple methods at one time, but for the purposes of this simple example only a single method is being analyzed.)

The *r* and *w* relations record which local variables are read and written by the various instructions. Note again that local variables are conveniently described by integers. The *ubt* and *cbt* relations give the unconditional branch targets and conditional branch targets for those instructions that have them, and do not provide any relation initializer for those instructions that do not. (This is also the case for all those relations that are described as having a set type.) The *term* relation shows that the method contains a single *return* instruction, which is terminal for the method being analyzed.

The *smod* relation is an important relation that describes the number of values that a given instruction pushes onto or pops off the stack. For a push this relation will have a positive value, while for a pop it will have a negative value. Thus we see that the opening *getstatic* and *ldc* instructions both push a single value onto the stack, while the subsequent *invokevirtual* instruction pops off two values. The *smod* relation is important because stack manipulation is often used by malicious code. Indeed, in the BlackBox applet, stack manipulation is used together with type confusion to replace an instance of a trusted class with an instance of a malicious class that appears to satisfy the same security guarantees, but it fact does not. In its current realization the model only insures that the stack depth satisfies a series of constraints that will be further described below. Ideally, one would like to add additional constraints about the types of the objects being added or removed from the stack, so that one could check, for example, that the arguments for the *invokevirtual* instruction matched the actual signature for that instruction. This is a goal for future extensions of the model. Even in its present form, however, the model will detect obviously undesirable behavior, such as forcing the stack depth to a negative value. This is further described below.

The $framelen$ relation is equally important. Each method in the JVM is equipped with a particular set of local variable slots, known as its frame. The $framelen$ relation encodes the size of that frame. Access to a local variable index that is outside the current frame is a prohibited operation that could have serious implications for security, just as in the case of access to uninitialized memory. The *cpindex* relation is used to encode the value of the constant pool index for each string of interest. In the current version of the model this is used for method resolution for the *invokevirtual* instruction, the targets of which are captured in the *iv* relation. Thus, for example, the *invokevirtual* instruction that occurs as the third instruction of the method has a target of 24, which is the constant pool index for the method with signature $voidprintln(String)$ in the $java.io.PrintStream$ class, as can be seen by consulting the *cpindex* relation initializer

```
java_io_PrintStream_println__L_java_lang_String__V->24
```

The final relation initializer section, for the *ordinal* initializer, assigns a unique ordinal to each method that is referenced by the set of methods being analyzed. Since it can be observed that the *inm* relation is empty, these ordinals will not be used in the model analysis for this method. Said another way, in this particular case we are not examining the calling context for the method in question, only the method itself. Because of the separation between the bytecode translator, which generates these initializers, and the model template itself, which uses these initializers, this information is always created, even though in this case it is not actually used.

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 within a method. 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 there 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 register 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 that will be 0. Finally, the *meth* relation of *s0* will be set to the ordinal of 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. The Alloy code for creating the initial state is shown below.

```
fact initialState {
    let s0 = ord/first | (s0.prog = startup) && no s0.readers &&
        no s0.writers && (s0.depth = 0) && s0.meth = jthis
}
```

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 (and possibly method) state. This predicate calls the *nextInstruction* predicate, which updates the value of the current instruction for *s'*. It then updates the method by calling the predicate *nextMethod*. It also 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] &&
        nextMethod[s.meth, s'.meth] &&
        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 its *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* relation 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 in the current method, or perhaps pointed to an offset that corresponded to a memory location outside the current method entirely. The model, however, insures that internal consistency is maintained, and that no invalid transfers of control such as these can occur. This will be described more fully in the next subsection.

If there was no unconditional branch, but there was a conditional branch, then the Alloy analyzer can choose to take that branch, or it can instead simply go the next instruction by adding the current value of the *map* relation to the length of the current instruction. More precisely, the Alloy analyzer will not choose a single possible path, it will choose all possible paths, and analyze each as a separate instance of the model. 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 exception handler; the conditional branch target is then the beginning of the handler code block.

```
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 } )
}
```

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 type confusion, 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; and 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 case would correspond to accessing out-of-bounds memory. The following Alloy assertions are used to verify these local variable 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 variable 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.)

The JVM specification makes 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, which also uses a complex application of type confusion. 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)
    }
```

Another constraint of interest is the constraint that expresses the internal consistency of the *map* and *len* relations. As stated earlier, these values encode redundant information about the instruction stream that makes up the methods in the classfile. One way of expressing this internal consistency is by means of a method termination predicate:

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

This predicate also relates to the handling of looping constructs. It 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. (In fact, as stated above, it actually exercises all possibilities; one possibility is chosen for each instance of the model.) 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 a 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 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. It need not be the case that a method terminates within the search space of a given model, but it should never be the case that the failure to terminate derives from an inconsistent instruction stream. Thus instruction transfer must be checked more carefully, since the failure of the *Terminates* predicate can have multiple causes.

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 executed. 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 invocation. 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 an 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

Java 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.ordinal = i.inm }
    }
```

The first constraint asserts that for all *Instruction*s, 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 or methods. That encoding is handled by the relation initializers, which must initialize all the instruction and method relations based on the bytecodes of a specified methods. The initialization must also handle the creation of concrete signatures that extend the abstract *Instruction* and *Method* and 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 [19]. 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 or method under consideration.

The operation of the Class2Alloy 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 all 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, including 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 and method 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.

3.5. 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; a simplified version will be described instead. It is important to note, however, that both the reduced version and the full version of BlackBox are detected by the techniques described herein.

The Java Virtual machine loads classes using a series of helper classes know as classloaders. The classloader class responsible for loading classes across the network is the *URLClassLoader* class. The *URLClassLoader* class not only verifies classfile syntax, it also 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 the *URLClassLoader* class 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 with the *SecurityManager*, one could then load and instantiate an arbitrary network class. This type of attack 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. 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 without modifying the JVM internal state, then the subsequent stack operation will pop the *myUCL* object off the stack and treat it as if it were an object of type *URLClassLoader*. One part of the BlackBox approach to performing this type confusion attack was to use malicious exception handling code. When an exception is thrown in Java, an exception object is placed on the stack so that the exception handler may access it. After the handler completes, the state of the stack is supposed to be restored to the same state it had before the exception was thrown. If it could be arranged that the malicious exception handler code actually modifies the stack such that two objects are popped off when the handler exits, the stack depth invariance constraint will be violated. The net effect in the actual BlackBox applet is than 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 mechanism used by BlackBox is significantly more complicated than this reduced version.) When presented with either form of BlackBox, the reduced form or the full, original form, the Alloy analyzer detects the violation of the stack depth invariance constraint. For more than a year (late 2001 to late 2002) it was the case that one particular popular implementation of the Java Bytecode Verifier would successfully load the BlackBox applet, without raising any constraint violations, hence BlackBox was a major source of infection during that time period. The flaw has since been corrected, and all current implementations of the Bytecode Verifier do detect this flaw. This example does illustrate the basic premise of this paper, however: any gap between the security specification and the implementation of that specification will not be caught by the offending implementation of the Bytecode Verifier. The approach proposed herein will catch it, however, since any specification violation is automatically a constraint violation that will be found by Alloy.

In addition to BlackBox, the approach in question has been tested extensively and produces no false positives. In addition, when run against known malicious applets, counterexamples are generated. Several malicious applets were synthesized to illustrate invalid memory accesses, instruction diversion and function diversion. All were detected. The approximate time to detect the indicated malicious applets ranged from thirty seconds to several minutes. In addition, numerous benign applets were tested with the analyzer and no counterexamples were found.

The approach used in this work generates very efficient Alloy models. Since the model template is fixed, it can (and has) been highly optimized to provide for efficient execution of the analyzer. Typical model runs focused on a single, specific method known in advance, e.g. when analyzing the full version of the BlackBox applet, take from a few seconds to a few minutes on a modern dual core CPU. Models spanning all the methods in a reasonably well designed Java class (e.g. tens of methods, each with tens of lines of Java source in length) can be analyzed in tens of minutes. One approach to performance acceleration would be to have the Class2Alloy translator output Java source code that directly called Alloy's Java API, rather than generating Alloy source text. This approach could be taken even further to target the Java API to Alloy's underlying constraint solver, KodKod. While these improvements will no doubt have to be made at some point in order to be able to analyze larger models, at the present time the convenience of being able to manipulate the Alloy source text outweighs the inconvenience of the corresponding performance penalty.

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 ultimate goal, of course, would be to check all constraints described in the JVM specification, not just the small number of high value constraints that are currently being checked.

The current model does not completely handle exceptions. In particular, only a single level of exception handling 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. In addition, interpreted languages such as JavaScript have also been the focus of recent study. JavaScript is particularly difficult since it does not have a formal security model, only a set of best current practices, but it is an important target, since cross-site scripting (XSS) and cross-site request forgery (CSRF) attacks using JavaScript are extremely widespread.

5. Conclusion

This paper has demonstrated that Alloy is an extremely powerful tool for performing security constraint analysis on Java bytecodes. Even at the current 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.

Appendix A. JVM bytecodes for a simple method

```
getstatic       #16  <Field PrintStream System.out>
ldc1            #22  <String "Hello">
invokevirtual   #24  <Method void PrintStream.println(String)>
ldc2w           #30  <Double 4D>
dstore_1
ldc1            #32  <String "hello">
```

```
invokestatic    #33  <Method org.apache.bcel.classfile.JavaClass
                      Repository.lookupClass(String)>
astore_3
getstatic       #16  <Field PrintStream System.out>
aload_3
invokevirtual   #39  <Method void PrintStream.println(Object)>
goto            112
astore_3
aload_3
invokevirtual   #42  <Method void Exception.printStackTrace()>
dload_1
aload_3
invokevirtual   #47  <Method String Exception.getMessage()>
invokevirtual   #51  <Method int String.hashCode()>
i2d
dadd
dstore_1
dload_1
ldc2w           #57  <Double 5D>
dcmpl
ifle            142
getstatic       #16  <Field PrintStream System.out>
new             #59  <Class StringBuilder>
dup
ldc1            #61  <String "D = ">
invokespecial   #63  <Method void StringBuilder(String)>
dload_1
invokevirtual   #65  <Method StringBuilder StringBuilder.append(double)>
invokevirtual   #69  <Method String StringBuilder.toString()>
invokevirtual   #24  <Method void PrintStream.println(String)>
goto            142
astore          4
dload_1
ldc2w           #57  <Double 5D>
dcmpl
ifle            109
getstatic       #16  <Field PrintStream System.out>
new             #59  <Class StringBuilder>
dup
ldc1            #61  <String "D = ">
invokespecial   #63  <Method void StringBuilder(String)>
dload_1
invokevirtual   #65  <Method StringBuilder StringBuilder.append(double)>
invokevirtual   #69  <Method String StringBuilder.toString()>
invokevirtual   #24  <Method void PrintStream.println(String)>
aload           4
athrow
dload_1
ldc2w           #57  <Double 5D>
dcmpl
ifle            142
getstatic       #16  <Field PrintStream System.out>
```

```
     new             #59  <Class StringBuilder>
     dup
     ldc1            #61  <String "D = ">
     invokespecial   #63  <Method void StringBuilder(String)>
     dload_1
     invokevirtual   #65  <Method StringBuilder StringBuilder.append(double)>
     invokevirtual   #69  <Method String StringBuilder.toString()>
     invokevirtual   #24  <Method void PrintStream.println(String)>
     return
```

Appendix B. Alloy initializers for a simple method

```
one sig startup, getstatic_1, ldc_2, invokevirtual_3, ldc2_w_4,
    dstore_1_5, ldc_6, invokestatic_7, astore_3_8, getstatic_9,
    aload_3_10, invokevirtual_11, goto_12, astore_3_13,
    aload_3_14, invokevirtual_15, dload_1_16, aload_3_17,
    invokevirtual_18, invokevirtual_19, i2d_20, dadd_21,
    dstore_1_22, dload_1_23, ldc2_w_24, dcmpl_25, ifle_26,
    getstatic_27, new_28, dup_29, ldc_30, invokespecial_31,
    dload_1_32, invokevirtual_33, invokevirtual_34,
    invokevirtual_35, goto_36, astore_37, dload_1_38, ldc2_w_39,
    dcmpl_40, ifle_41, getstatic_42, new_43, dup_44, ldc_45,
    invokespecial_46, dload_1_47, invokevirtual_48,
    invokevirtual_49, invokevirtual_50, aload_51, athrow_52,
    dload_1_53, ldc2_w_54, dcmpl_55, ifle_56, getstatic_57,
    new_58, dup_59, ldc_60, invokespecial_61, dload_1_62,
    invokevirtual_63, invokevirtual_64, invokevirtual_65,
    return_66 extends Instruction {}

one sig jthis, init__, clinit__,
    java_io_PrintStream_println__Ljava_lang_String__V,
    java_io_PrintStream_println__Ljava_lang_Object__V,
    java_lang_Throwable_printStackTrace___V,
    java_lang_Throwable_getMessage___Ljava_lang_String_,
    java_lang_String_hashCode___I,
    java_lang_StringBuilder_append__D_Ljava_lang_StringBuilder_,
    java_lang_StringBuilder_toString___Ljava_lang_String_ extends Method {}

fact maps {
    map = startup->(-1) + getstatic_1->0 + ldc_2->3 +
    invokevirtual_3->5 + ldc2_w_4->8 + dstore_1_5->11 +
    ldc_6->12 + invokestatic_7->14 + astore_3_8->17 +
    getstatic_9->18 + aload_3_10->21 + invokevirtual_11->22 +
    goto_12->25 + astore_3_13->28 + aload_3_14->29 +
    invokevirtual_15->30 + dload_1_16->33 + aload_3_17->34 +
    invokevirtual_18->35 + invokevirtual_19->38 + i2d_20->41 +
    dadd_21->42 + dstore_1_22->43 + dload_1_23->44 +
    ldc2_w_24->45 + dcmpl_25->48 + ifle_26->49 +
    getstatic_27->52 + new_28->55 + dup_29->58 + ldc_30->59 +
    invokespecial_31->61 + dload_1_32->64 +
    invokevirtual_33->65 + invokevirtual_34->68 +
    invokevirtual_35->71 + goto_36->74 + astore_37->77 +
    dload_1_38->79 + ldc2_w_39->80 + dcmpl_40->83 + ifle_41->84 +
    getstatic_42->87 + new_43->90 + dup_44->93 + ldc_45->94 +
    invokespecial_46->96 + dload_1_47->99 +
    invokevirtual_48->100 + invokevirtual_49->103 +
    invokevirtual_50->106 + aload_51->109 + athrow_52->111 +
    dload_1_53->112 + ldc2_w_54->113 + dcmpl_55->116 +
    ifle_56->117 + getstatic_57->120 + new_58->123 +
    dup_59->126 + ldc_60->127 + invokespecial_61->129 +
    dload_1_62->132 + invokevirtual_63->133 +
    invokevirtual_64->136 + invokevirtual_65->139 +
    return_66->142
}

fact lens {
    len = startup->1 + getstatic_1->3 + ldc_2->2 +
    invokevirtual_3->3 + ldc2_w_4->3 + dstore_1_5->1 + ldc_6->2 +
    invokestatic_7->3 + astore_3_8->1 + getstatic_9->3 +
    aload_3_10->1 + invokevirtual_11->3 + goto_12->3 +
    astore_3_13->1 + aload_3_14->1 + invokevirtual_15->3 +
    dload_1_16->1 + aload_3_17->1 + invokevirtual_18->3 +
    invokevirtual_19->3 + i2d_20->1 + dadd_21->1 +
    dstore_1_22->1 + dload_1_23->1 + ldc2_w_24->3 + dcmpl_25->1 +
    ifle_26->3 + getstatic_27->3 + new_28->3 + dup_29->1 +
    ldc_30->2 + invokespecial_31->3 + dload_1_32->1 +
    invokevirtual_33->3 + invokevirtual_34->3 +
    invokevirtual_35->3 + goto_36->3 + astore_37->2 +
    dload_1_38->1 + ldc2_w_39->3 + dcmpl_40->1 + ifle_41->3 +
    getstatic_42->3 + new_43->3 + dup_44->1 + ldc_45->2 +
    invokespecial_46->3 + dload_1_47->1 + invokevirtual_48->3 +
    invokevirtual_49->3 + invokevirtual_50->3 + aload_51->2 +
    athrow_52->1 + dload_1_53->1 + ldc2_w_54->3 + dcmpl_55->1 +
    ifle_56->3 + getstatic_57->3 + new_58->3 + dup_59->1 +
    ldc_60->2 + invokespecial_61->3 + dload_1_62->1 +
    invokevirtual_63->3 + invokevirtual_64->3 +
    invokevirtual_65->3 + return_66->1
}

fact rs {
    r = aload_3_10->3 + aload_3_14->3 + dload_1_16->1 +
    dload_1_16->2 + aload_3_17->3 + dload_1_23->1 +
    dload_1_23->2 + dload_1_32->1 + dload_1_32->2 +
    dload_1_38->1 + dload_1_38->2 + dload_1_47->1 +
    dload_1_47->2 + aload_51->4 + dload_1_53->1 + dload_1_53->2 +
    dload_1_62->1 + dload_1_62->2
}

fact ws {
    w = startup->0 + startup->1 + dstore_1_5->1 + dstore_1_5->2 +
    astore_3_8->3 + astore_3_13->3 + dstore_1_22->1 +
    dstore_1_22->2 + astore_37->4
}
```

```
fact ubts {
    ubt = goto_12->112 + goto_36->142
}

fact cbts {
    cbt = ifle_26->142 + ifle_41->109 + ifle_56->142
}

fact terms {
    term = return_66->1
}

fact smods {
    smod = startup->0 + getstatic_1->1 + ldc_2->1 +
    invokevirtual_3->(-2) + ldc2_w_4->2 + dstore_1_5->(-2) +
    ldc_6->1 + invokestatic_7->0 + astore_3_8->(-1) +
    getstatic_9->1 + aload_3_10->1 + invokevirtual_11->(-2) +
    goto_12->0 + astore_3_13->(-1) + aload_3_14->1 +
    invokevirtual_15->(-1) + dload_1_16->2 + aload_3_17->1 +
    invokevirtual_18->0 + invokevirtual_19->0 + i2d_20->1 +
    dadd_21->(-2) + dstore_1_22->(-2) + dload_1_23->2 +
    ldc2_w_24->2 + dcmpl_25->(-3) + ifle_26->(-1) +
    getstatic_27->1 + new_28->1 + dup_29->1 + ldc_30->1 +
    invokespecial_31->(-2) + dload_1_32->2 +
    invokevirtual_33->(-2) + invokevirtual_34->0 +
    invokevirtual_35->(-2) + goto_36->0 + astore_37->(-1) +
    dload_1_38->2 + ldc2_w_39->2 + dcmpl_40->(-3) +
    ifle_41->(-1) + getstatic_42->1 + new_43->1 + dup_44->1 +
    ldc_45->1 + invokespecial_46->(-2) + dload_1_47->2 +
    invokevirtual_48->(-2) + invokevirtual_49->0 +
    invokevirtual_50->(-2) + aload_51->1 + athrow_52->0 +
    dload_1_53->2 + ldc2_w_54->2 + dcmpl_55->(-3) +
    ifle_56->(-1) + getstatic_57->1 + new_58->1 + dup_59->1 +
    ldc_60->1 + invokespecial_61->(-2) + dload_1_62->2 +
    invokevirtual_63->(-2) + invokevirtual_64->0 +
    invokevirtual_65->(-2) + return_66->0
}

fact jsrs {
    no jsr
}

fact framelens {
    framelen = jthis->4 + init__->12 + clinit__->0 +
    java_io_PrintStream_println__Ljava_lang_String__V->0 +
    java_io_PrintStream_println__Ljava_lang_Object__V->0 +
    java_lang_Throwable_printStackTrace___V->0 +
    java_lang_Throwable_getMessage___Ljava_lang_String_->0 +
    java_lang_String_hashCode___I->0 +
    java_lang_StringBuilder_append__D_Ljava_lang_StringBuilder_->0 +
    java_lang_StringBuilder_toString___Ljava_lang_String_->0
}

fact accs {
    acc = jthis->0 + jthis->3 + init__->0 + clinit__->0 +
    java_io_PrintStream_println__Ljava_lang_String__V->0 +
    java_io_PrintStream_println__Ljava_lang_Object__V->0 +
    java_lang_Throwable_printStackTrace___V->0 +
    java_lang_Throwable_getMessage___Ljava_lang_String_->0 +
    java_lang_String_hashCode___I->0 +
    java_lang_StringBuilder_append__D_Ljava_lang_StringBuilder_->0 +
    java_lang_StringBuilder_toString___Ljava_lang_String_->0
}

fact cpindexs {
    cpindex = jthis->0 + init__->0 + clinit__->0 +
    java_io_PrintStream_println__Ljava_lang_String__V->24 +
    java_io_PrintStream_println__Ljava_lang_Object__V->39 +
    java_lang_Throwable_printStackTrace___V->42 +
    java_lang_Throwable_getMessage___Ljava_lang_String_->47 +
    java_lang_String_hashCode___I->51 +
    java_lang_StringBuilder_append__D_Ljava_lang_StringBuilder_->65 +
    java_lang_StringBuilder_toString___Ljava_lang_String_->69
}

fact ivs {
    iv = invokevirtual_3->24 + invokevirtual_11->39 +
    invokevirtual_15->42 + invokevirtual_18->47 +
    invokevirtual_19->51 + invokevirtual_33->65 +
    invokevirtual_34->69 + invokevirtual_35->24 +
    invokevirtual_48->65 + invokevirtual_49->69 +
    invokevirtual_50->24 + invokevirtual_63->65 +
    invokevirtual_64->69 + invokevirtual_65->24
}

fact ordinals {
    ordinal = jthis->0 + init__->1 + clinit__->2 +
    java_io_PrintStream_println__Ljava_lang_String__V->3 +
    java_io_PrintStream_println__Ljava_lang_Object__V->4 +
    java_lang_Throwable_printStackTrace___V->5 +
    java_lang_Throwable_getMessage___Ljava_lang_String_->6 +
    java_lang_String_hashCode___I->7 +
    java_lang_StringBuilder_append__D_Ljava_lang_StringBuilder_->8 +
    java_lang_StringBuilder_toString___Ljava_lang_String_->9
}
```

Appendix C. The Global signature

```
one sig Globals {
    ACC_PUBLIC:       Int,
    ACC_PRIVATE:      Int,
    ACC_PROTECTED:    Int,
    ACC_STATIC:       Int,
    ACC_FINAL:        Int,
    ACC_SYNCHRONIZED: Int,
    ACC_BRIDGE:       Int,
    ACC_VARARGS:      Int,
    ACC_NATIVE:       Int,
    ACC_INTERFACE:    Int,
    ACC_ABSTRACT:     Int,
    ACC_STRICT:       Int,
    ACC_SYNTHETIC:    Int,
    ACC_ANNOTATION:   Int
}

fact Globals_Init {
     ACC_PUBLIC =       Globals->0  &&
     ACC_PRIVATE =      Globals->1  &&
     ACC_PROTECTED =    Globals->2  &&
     ACC_STATIC =       Globals->3  &&
     ACC_FINAL =        Globals->4  &&
     ACC_SYNCHRONIZED = Globals->5  &&
     ACC_BRIDGE =       Globals->6  &&
     ACC_VARARGS =      Globals->7  &&
     ACC_NATIVE =       Globals->8  &&
     ACC_INTERFACE =    Globals->9  &&
     ACC_ABSTRACT =     Globals->10 &&
     ACC_STRICT =       Globals->11 &&
     ACC_SYNTHETIC =    Globals->12 &&
     ACC_ANNOTATION =   Globals->13
}
```

References

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

[2] 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

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

[4] Reynolds, M.: Lightweight Modeling of Java Virtual Machine Security Constraints. Abstract State Machines, Alloy, B and Z; Proceedings of the Second International Conference, ABZ 2010. M. Frappier, ed. Springer (2010).

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

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

[7] BlackBox Security Advisory (2002), http://www.ca.com/us/securityadvisor/virusinfo/virus.aspx?ID=36725

[8] 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

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

[10] Xu, H.: Java Security Model and Bytecode Verification (2000), http://www.cis.umassd.edu/~hxu/Papers/UIC/JavaSecurity.PDF

[11] Posegga, J., Vogt, H.: Java bytecode verification using model checking (1998), http://eprints.kfupm.edu.sa/47269

[12] Leroy, X.: Java Bytecode Verification: Algorithms and Formalizations. J. Autom. Reasoning 30(3-4): 235-269 (2003).

[13] Gal, A., Probst, C., Franz, M.: A Denial of Service Attack on the Java Bytecode Verifier. Technical Report 03-23, University of California, Irvine, School of Information and Computer Science, November 2003.

[14] Bodden, E.: Project Report: Efficient Java bytecode verification by the means of proof-carrying code, April 2006.

[15] Freund, S., Mitchell, J.: A Type System for the Java Bytecode Language and Verifier. J. Autom. Reasoning 30(3-4): 271-321 (2003).

[16] Sirer, E., Bershad, B.: Using production grammars in software testing. DSL 1999: 1-13.

[17] Higuchi, T., Ohori, A.: A static type system for JVM access control, ACM Transactions on Programming Languages and Systems, Vol 29, Issue 1 (January 2007).

[18] Preda, M., Christadorescu, M., Jha, S., Debray, S.: A semantics-based approach to malware detection. ACM Transactions on Programming Languages and Systems, Vol 30, Issue 5 (2008).

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

[20] Gentoo Security Advisory, `http://security.gentoo.org/glsa/glsa-200911-02.xml`