Dynamic Symbolic Execution for the Analysis of Web Server Applications in Java

***Abstract*—Symbolic execution is a well-known program analysis technique that explores multiple program paths simultaneously. Among other things, it is used to uncover subtle bugs and corner cases in programs, as well as to produce high-coverage test suites. Even though symbolic execution has seen successful use in practice, there remain challenges in applying it to real programs like web servers that use features such as multithreading and callbacks. This paper describes our dynamic symbolic execution framework for Java that was designed with these types of features in mind. Our framework uses bytecode instrumentation combined with a run-time agent to perform the symbolic execution. We give a detailed description of the challenges we faced along with our design choices. We also present benchmark results on various examples including programs that use web server frameworks.**

***Keywords*-symbolic execution; program analysis; testing**

I. Introduction

Symbolic execution is a well-known program analysis technique that explores multiple program paths exhaustively. The basic technique, which executes a program using symbolic inputs, was proposed several decades ago [12]. Since that time, improvements in constraint solving and variants to the original symbolic execution technique have led to widespread use.

Dynamic symbolic execution (DSE), also known as concolic execution [9], is one of the most popular variants of the original symbolic execution algorithm. Dynamic symbolic execution executes a program using real, concrete inputs while simultaneously performing symbolic execution. This allows an execution to proceed even when a program contains elements such as native system calls that typically cannot be executed symbolically without placeholder stubs.

Even though symbolic execution has seen successful use in both research and industry [6], [3], using it to analyze programs that use complex language features remains a challenge, especially in the Java programming language. For example, applications built on top of web server frameworks (such as NanoHTTPD and Spark) are difficult to execute symbolically. One challenge is the use of features like multithreading and callbacks. Web servers often use these two features in tandem: a user-defined callback is invoked by the web framework on a new thread created by the framework in response to a client request. This is difficult for symbolic execution frameworks because it requires the symbolic execution of multiple threads.

Another challenge with symbolic execution is scalability. Apart from path explosion issues that affect symbolic execution techniques in general [4], it is often the case that the user application code is relatively small compared to the size of the supporting Java libraries, such as web frameworks and the Java class library. However, if all of the libraries must also be symbolically executed along with the user application code, the scalability suffers.

Another challenge with performing symbolic execution of real-world Java programs is the complexity introduced by the Java Virtual Machine (JVM). For instance, static class initializers are not explicitly invoked by bytecode instructions in a program, but rather by the JVM. Behaviors like these make performing symbolic execution using an instrumentation-based approach difficult.

This paper describes our Java-based dynamic symbolic execution framework that was built with the above challenges in mind. Our framework makes the following specific contributions.

- Explicit handling of *multithreading* and *callbacks*; two features necessary to symbolically analyze web server applications.
- Selective *instrumentation* to include only selected classes in symbolic execution.
- A run-time *agent* to assist in maintaining the symbolic execution stack. This is necessary to robustly handle the complexities introduced by the implementation of the JVM.

The rest of this paper is organized as follows. Section II gives background information on symbolic execution. Section III describes the basics of our DSE framework, and Section IV describes how we support multithreading, callbacks, and selective instrumentation. Section V gives a small but complete NanoHTTPD-based web server and shows how our framework can be applied to it. We give benchmark analysis times on programs of varying complexity in Section VI. We survey related work in Section VII, and we conclude and describe future directions in Section VIII.

II. Background

This section provides a brief summary of symbolic execution; extensive treatments can be found in survey papers [5]. Pure symbolic execution, or *static* symbolic execution, executes a program using inputs that are purely symbolic. During program execution, constraints involving those symbolic variables are formulated. When a branching condition involving a symbolic variable is encountered, symbolic execution attempts to explore both branches by *forking* execution: along one branch, the symbolic constraint corresponding to the "true" condition is appended to the *path condition*, and the symbolic constraint for the "false" condition is appended to path condition of the other branch. The path constraint for a

given path can then be given to a constraint solver; a satisfying solution provides concrete values for the input variables that drive execution along that path. An unsatisfiable constraint indicates an invalid program path that can be pruned from further exploration.

Dynamic symbolic execution performs both a *concrete* execution of a program while simultaneously also using symbolic inputs and formulating path constraints over those symbolic inputs along the concrete path. Constraints whose solution can be used to drive execution through a different program path can be formulated by using a *prefix* of the path condition along with the *negation* of the final constraint in the prefix. Even though DSE is not exhaustive, in practice, it is popular due to both its speed and ability to find bugs in real-world programs [6], [3].

Dynamic symbolic execution can be done either *online* or *offline*. Online techniques perform the symbolic constraint generation and solving while the target program is executing, whereas offline techniques typically "log" the executed instructions and symbolically execute them in a separate process . Our technique uses the online approach.

III. Architecture and operation

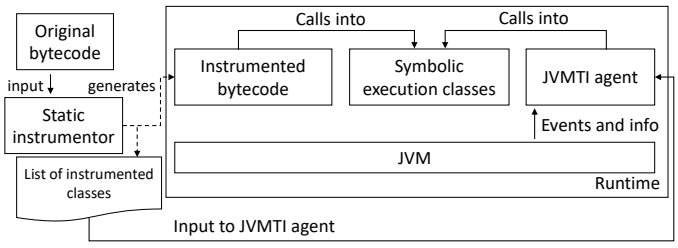

Fig. 1. High-level architecture.

The high-level architecture of our framework is shown in Figure 1. The static instrumentation component takes as input the original application jar (containing all classes that should be instrumented) and produces as output a new jar file containing instrumented versions of those classes, as well as a list of the instrumented classes. At runtime, the injected bytecode instructions invoke methods in our symbolic execution framework classes, which perform the symbolic execution. The attached JVMTI agent helps maintain the symbolic "shadow" stack (i.e., the stack used by the symbolic execution engine), and also invokes methods in the symbolic execution classes to notify the symbolic execution engine about events such as method entry. This section describes the basic workings of our framework, including the symbolic execution classes, the instrumentation, and symbolic variables.

A. Execution framework

Our runtime framework consists of four main classes, briefly sketched in Figure 2. The `Wrapper` class contains static methods, roughly one per Java bytecode instruction, that are invoked by the instrumented bytecode. The class also contains a map from thread identifiers to instances of the `Executor` class (described below); this map stores a separate symbolic execution engine (`Executor`) for each thread running instrumented code. The static methods (1) lookup the `Executor` for the calling thread, and then (2) invoke the corresponding method on that `Executor` instance.

Our symbolic execution engine is implemented in the `Executor` class. This class contains methods, roughly one per Java bytecode instruction, that implement the semantics of that instruction. The Executor class uses two main supporting classes in its implementation: a `StackFrame` class and a `Value` class.

The `StackFrame` class represents a JVM stack frame to the symbolic execution engine. Its behavior is designed to mimic the behavior of the stack frames as they are used by the JVM: a new stack frame is created when a method is called and removed when the method exits. The two most important fields of this class are the *local variable table* and the *operand stack*. The local variable table stores the local variables and is indexed by the bytecode instructions, i.e., `iload_1` retrieves the integer variable at index 1 in the local variable table.

The operand stack mimics the behavior of the JVM operand stack: in the same way that the JVM bytecode instructions manipulate the JVM stack (i.e., the `iconst_0` instruction pushes the constant 0 onto the JVM stack), the methods in the `Executor` class manipulate the symbolic stack. The symbolic stack is implemented as a `Stack` data structure that contains instances of the `Value` class.

The `Value` class is used to "box" values on the symbolic stack and ensure that all values of all types are treated in a uniform way. For instance, a `Value` instance of type `INT32` could contain a boxed integer, or it could contain a symbolic variable representing an integer. The `Value` class contains two fields: an `Object` that holds either a reference type or a boxed primitive type, and a *type* field describing the type held by this instance.

B. Static instrumentation

Our static instrumentation component takes as input a jar file containing the classes to instrument and produces as output a new jar file containing instrumented versions of the original classes. The instrumentation transforms every class file in the input jar individually. It works by sequentially examining every bytecode instruction of every method. For every original bytecode instruction in the method, additional bytecode instructions are injected before and/or after the original instruction. These injected bytecode instructions drive the execution of a symbolic *shadow* JVM that mimics the real one. This design is similar to the .NET tool Pex [15]; one difference is that Pex instruments the .NET IR at runtime when a class is loaded as opposed to statically.

Figure 3 gives a simple example that shows the correspondence between the Java source code, the original bytecode, and the instrumented bytecode. The source code declares two integer variables, x and y, and assigns their sum to the variable z. The original Java bytecode implements this on a stack machine: the constant 0 is pushed onto the stack and then the top of the stack is stored into local variable 0 (x).

```
public class Wrapper {
  // a map from thread ids to execution engines
  private static Map<Integer, Executor> executorMap;
  public static void loadConstant(int threadID) {
    // lookup Executor for threadID in executorMap
    // delegate call to threadID's Executor
  }
  // static methods for each bytecode instruction
}

public class Executor {
  private StackFrame currentStackFrame;
  private Stack<StackFrame> executionStack;

  // object map for representing class instances
  private Map<Integer, Map<String, Value>> objectMap;

  public void loadConstant(int index) {
      // retrieve local variable from current stack frame
  }
  // methods for other bytecode instructions
}

public class StackFrame {
  private int threadId;
  private String methodName; // name of executing method
  private Value[] localVariableMap;
  private Stack<Value> operandStack;
  ...
}

public class Value {
  public static final int BLN = 0x0001; // bool or Boolean
  public static final int CHR = 0x0002; // char or Char
  public static final int INT8 = 0x0004; // byte or Byte
  ...
  public static final int DBL = 0x0080; // double or Double
  public static final int STR = 0x0100; // String
  public static final int REF = 0x0200; // Object reference
  public static final int ARY = 0x2000; // single−dimensional array reference
  public static final int MARY = 0x4000; // multi−dimensional array reference
  public static final int SYM = 0x8000; // symbolic entry

  private Object value = null; // the actual value
  private int type = 0; // the type of the value
  ...
}
```

Fig. 2. The core framework classes.

The instrumented bytecode in Figure 3 shows how the *shadow* symbolic execution engine is driven by the injected bytecode. The first original instruction, which loads the constant 0 onto the JVM stack (iconst_0) results in the addition of two bytecode instructions. The first injected instruction loads the constant 0 onto the JVM stack, and the second injected instruction invokes the static `loadConstant` method in the `Wrapper` class. These static methods in the `Wrapper` class are described in Section III-A in more detail. Briefly, the `loadConstant` method in Figure 3 takes one argument (an integer), thus consuming the constant that was pushed onto the JVM stack by the first injected bytecode instruction. The `loadConstant` method then pushes this argument onto the symbolic shadow stack.

C. Defining symbolic values

Our DSE framework introduces symbolic values into the system on-the-fly. A configuration file is used to specify which method parameters, local variables, or fields (including static fields) are to be treated as symbolic. Specifically, this configuration file captures two pieces of information for each variable that should be treated symbolically: (1) the name name of the method in which the variable variable is accessed, and (2) the local variable number of the variable inside the method.

At runtime, methods in the `Executor` corresponding to load instructions (e.g., `iload_1`), are examined to check if symbolic values need to be injected. When a load instruction is executed, the framework checks whether the name of the currently executing method as well as the variable number (for load instructions) or field name (for field fetching instructions) are matched by an entry in the configuration file. In cases with a match, the concrete value of that variable is replaced with a symbolic value of the appropriate type and pushed onto the symbolic shadow stack. This substitution is performed at most one time for each symbolic variable, which reduces overhead and ensures that symbolic variables are introduced in a specified method at most once.

D. Representing classes

Our static instrumentation component (Section III-B) does not inspect the fields of the instrumented classes, nor does it introduce any additional fields into those classes. Instead, objects created using the `new` operator, as well as symbolic fields for such objects, are represented in the following way in the symbolic execution engine.

An object created with the `new` operator is represented[1] by a `Value` object whose *type* is `Value.REF` (Figure 2) *value* is an `Integer`. This Integer value identifies the corresponding Java object by indexing into a map that itself holds maps from Strings to Values. This second map is a *field* map, where the key (of type `String`) is the name of a field of the corresponding object, and the value (of type `Value`) holds the current value of that field.

An example is shown in Figure 4 in which an instance of the `Person` class is created and its `age` is assigned a value. The instrumented bytecode is shown on the right-hand side of Figure 4; lines beginning with a * were injected by the instrumentation. The instantiation of the `Person` class results in a call to the `newObject()` in the `Executor` class. This method (1) pushes a new `Value` object onto the symbolic stack whose *type* is `Value.REF` and whose actual value is a unique integer identifier, and (2) adds an entry to the object map (a static field in the `Wrapper` class) mapping from that unique identifier to a `HashMap` (the *fieldMap* in Figure 5).

Assigning a value to the `age` field triggers a call to the `putField()` method in the `Wrapper` class (line 58 in Figure 4). The content of the symbolic stack before the call is shown on the left-hand side of Figure 5. The `putField()` method uses the unique id (*id_1*) of the reference type to retrieve that object's field map; it then adds a key-value pair to that map that associates the key *age* with the value *99*; the value 99 is retrieved from the symbolic stack (the left-hand side of Figure 5), and the key is passed to the `putField()` method via the instrumented bytecode (line *56 in Figure 4).

[1]Our framework supports symbolic fields for classes, but purely symbolic references are not currently supported.

```
public static void sample() {
  int x = 2;
  int y = 3;
  int z = x + y;
  …
}

iconst_2  // push 2 onto stack
istore_0  // store stack into local variable 0 (x)
iconst_3 // push 3 onto stack
istore_1 // store stack into local variable 1 (y)
iload_0 // load local var 0 (x) onto stack
iload_1  // load local var 1 (y) onto stack
iadd // add elements on stack, store result on stack
istore_2 // store top of stack into local var 2 (z)
return // return from method
```

```
*iconst_2
*invokestatic #39 // Method Wrapper.loadConstant:(I)V
 iconst_2
*bipush 0
*invokestatic #42 // Method Wrapper.storeInteger:(I)V
 istore_0
// int y = 3 handled similarly
…
*bipush 0
*invokestatic #43 // Method Wrapper.loadInteger:(I)V
 iload_0
*bipush 1
*invokestatic #43 // Method Wrapper.loadInteger:(I)V
 iload_1
*invokestatic #44 // Method Wrapper.addInteger:()V
 iadd
*bipush 2
*invokestatic #42 // Method Wrapper.storeInteger:(I)V
 istore_2
return
```

Fig. 3. Original Java code and bytecode (left), and instrumented Java bytecode (right).

IV. Multithreading and complex features

This section describes how a native JVMTI agent is used to support the symbolic execution of selectively instrumented code, multiple threads, and features such as callbacks.

A. Selective instrumentation with a native agent

Static instrumentation alone is enough to perform symbolic execution for relatively simple programs. However, programs that use features such as static class initializers, native methods, virtual methods, callbacks, and multithreading can make symbolic execution through instrumentation alone very difficult. This is especially true if not every class is instrumented, which is necessarily the case with methods that are implemented natively: there is no bytecode available to instrument.

The biggest reason for this difficulty is related to method invocation and the management of the symbolic execution stack that must happen when methods are invoked. For example, static initializers are invoked by the JVM rather than explicitly by bytecode. If a static initializer itself invokes a method, maintaining the symbolic execution stack becomes complicated when deciding whether an instrumented or uninstrumented method was invoked.

Our framework uses a native JVMTI agent to solve such issues related to method invocation. This agent works by registering to receive two callbacks from the JVM: one whenever a method is entered and one whenever a method is exited. The method entry callback is responsible for removing parameters from the symbolic stack, and, if necessary, creating a stack frame for the called method. The method exit callback is responsible for pushing the return value of the method onto the symbolic stack, if necessary. In each of the two callbacks, there are four cases to consider:

- Instrumented method calls an instrumented method.
- Instrumented method calls an uninstrumented method.
- Uninstrumented method calls an instrumented method.
- Uninstrumented method calls an uninstrumented method.

The first case case, instrumented to instrumented, means that the parameters to the method being invoked have already been pushed onto the symbolic stack by the injected bytecode. In this case, the agent calls back into the execution engine, which (1) creates a new stack frame for the method being invoked, and (2) removes method arguments from the symbolic stack and puts them in the local variable table of the stack frame for the method being invoked.

The second case, instrumented to uninstrumented, means that the instrumented bytecode may have pushed arguments onto the symbolic stack anticipating that the callee would remove them. In this case, the agent calls a method in the execution engine that simply removes those arguments from the symbolic stack.

The third case, uninstrumented to instrumented, means that the symbolic stack contains no parameters for the method being invoked. This happens, for instance, when the program under analysis uses callbacks: if a callback is instrumented, but the caller of the callback is not instrumented, then the arguments are not available via the symbolic stack. In this case, the native agent retrieves the concrete method arguments via the JNI interface, and passes them to the symbolic execution engine, one at a time. For primitive types, the arguments are simply wrapped in a Value object and placed into the appropriate slot in the local variable table. For reference type arguments, the symbolic execution engine first attempts to lookup the corresponding symbolic representation through a reverse lookup map. If found, the corresponding symbolic representation is retrieved and placed into the local variable table. Otherwise, a new Value wrapper for the object is created (see Section III-D) and placed into the local variable table.

The last case, uninstrumented to uninstrumented, can be ignored because the caller does not place any information on the symbolic stack, nor does the called method return anything used by instrumented code. For uninstrumented methods, we are only interested in the cases that interface with instrumented code.

```
class Person {                *10: ldc           #58 // String Person
  public int age, id;         *12: invokestatic  #62 // Method Wrapper.newObject:(Ljava/lang/String;)V
}                              15: new           #2  // class Person
                              *18: invokestatic  #65 // Method Wrapper.executeDup:()V
Person p = new Person();       21: dup
p.age = 99;                    22: invokespecial #66 // Method "<init>":()V
                              *32: dup
                              *33: bipush        1
                              *35: invokestatic  #69 // Method Wrapper.storeLocalReference:(Ljava/lang/Object;I)V
                               38: astore_1
                               39: aload_1
                              *40: dup
                              *41: bipush        1
                              *43: invokestatic  #21 // Method Wrapper.loadLocalReference:(Ljava/lang/Object;I)V
                              *46: bipush        99
                              *48: invokestatic  #72 // Method Wrapper.pushByteAsInteger:(I)V
                               51: bipush        99
                               53: putfield      #74 // Field age:I
                              *56: ldc           #75 // String age
                              *58: invokestatic  #78 // Method Wrapper.putField:(Ljava/lang/String;)V
```

Fig. 4. Left-hand side: a simple class and code using this class. Right-hand side: the corresponding instrumented bytecode. Lines beginning with a * were injected by the instrumentation.

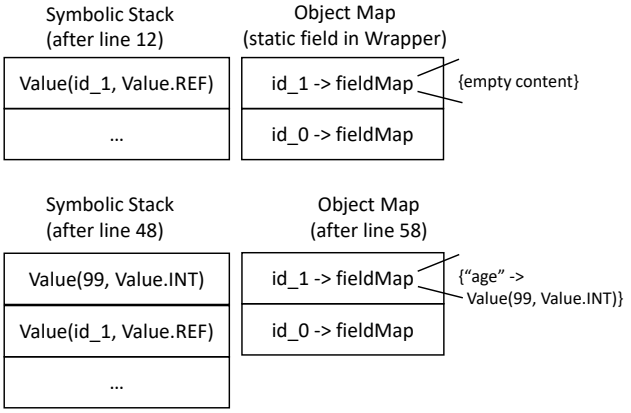

Fig. 5. The symbolic stack and object map during the execution of the bytecode in Figure 4.

B. Supporting threads

Symbolic execution of multithreaded Java programs is complicated by the fact that most JVM implementations are inherently multithreaded. For example, the JVM implicitly creates threads to handle background tasks that are not explicitly invoked by the application, such as garbage collection. Additionally, many libraries, such as web server frameworks, create background threads that are only "indirectly" related to the main application, e.g., they listen for client requests. A symbolic execution framework that blindly runs all threads symbolically will suffer performance penalties.

Our framework automatically creates a separate symbolic execution engine for each thread that is executing instrumented code. Figure 6 shows the process of how a new thread created to run the `serve()` method callback in Figure 7 results in the creation of a new symbolic execution engine. At run-time, upon the new thread's first call to an instrumented method (in this case, the `serve()` method, indicated by (1) in Figure 6), the native agent receives a callback (step (2) in Figure 6)

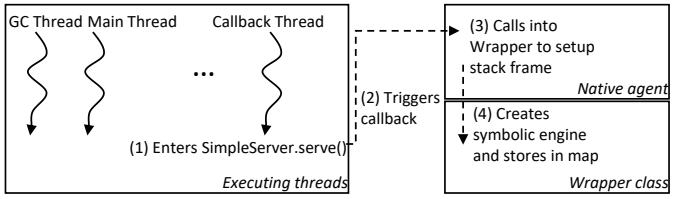

Fig. 6. Thread creation example.

and detects that the current thread is entering an instrumented method; the agent knows whether a method is instrumented or not by checking the fully-qualified class name against a of list of instrumented classes given to the agent (through a file) at startup.

The agent then detects, in this case, that the instrumented method is being invoked from an uninstrumented method (Section IV-A), and invokes a static method in the `Wrapper` class to initialize the stack frame for the method being called (step (3) in Figure 6). This method in the `Wrapper` class takes the thread identifier of the current thread as a parameter and checks whether a symbolic execution engine exists for this thread id. If so, it is retrieved, and if not, a new engine is created. In both cases, the stack frame for the method is established.

The same mechanism is used not only for true multithreaded programs, but also to create the symbolic execution engine for the "main" thread in a single-threaded program.

V. Simple example

This section provides a simplified but complete example of analyzing a NanoHTTPD-based web server program with our framework. The entire implementation for the example is shown in Figure 7. The `main` method creates an instance of the `SimpleServer`, whose constructor specifies `8080` as the listening port and starts the server. The `NanoHTTPD` framework classes then listen for http connections.

```
import java.io.IOException;
import java.util.Map;
import fi.iki.elonen.NanoHTTPD;

public class SimpleServer extends NanoHTTPD {

  public SimpleServer() throws IOException {
    super(8080); // listen on port 8080
    start(NanoHTTPD.SOCKET_READ_TIMEOUT, false);
    System.out.println("Server running on http://localhost:8080/");
  }

  public static void main(String[] args) {
    try {
      new SimpleServer();
    } catch (IOException e) {
      System.err.println("Error starting nano server:\n" + e);
    }
  }

  @Override
  public Response serve(IHTTPSession session) {
    Map<String, String> parms = session.getParms();
    String res;
    String a = parms.get("a");
    String b = parms.get("b");
    if (a != null && b != null) {
      int i_a = Integer.parseInt(a);
      int i_b = Integer.parseInt(b);
      res = getResponse(i_a, i_b);
    } else {
      res = "<html><body><h1>Missing arguments!</h1>\n";
    }

    return newFixedLengthResponse(res + "</body></html>\n");
  }

  public String getResponse(int a, int b) {
     int x = 1 , y = 0;
     if (a != 0) { // symbolic branch
       y = 3 + x;
       if (b == 0) // symbolic branch
         x = 2 * (a + b);
     }
     if (x - y == 0) // possibly symbolic branch
       return "Error!";
     else
       return "Result of x-y: " + (x - y);
  }
}
```

Fig. 7. A simple NanoHTTPD web server example.

When a client connects, the framework makes a callback to the user-supplied `serve` method on a new thread and supplies an `IHTTPSession` object. The `IHTTPSession` object contains, among other things, the user-specified input parameters. Our `serve` method attempts to extract integer parameters named `a` and `b`, and if successful, invokes the `getResponse` method with `a` and `b`. The `getResponse` method implements a simple computation from [5]; the goal of symbolic analysis is to determine if there are values for `a` and `b` such that the value of `x - y` is 0.

Analysis of this program using our framework works as follows. First, the code (in Figure 7) is compiled and packaged into a jar that is given to our instrumentor; this produces a new jar file containing instrumented code and a text file containing a list of classes that were instrumented (used at runtime by the agent to know which methods are instrumented). Next, symbolic variables are specified. In this example, the inputs to the `getResponse()` method are both user-supplied input; they are local variables 1 and 2 in the method (the `this` Object is parameter 0 in an instance method). The instrumented jar is then run; concretely, this starts a web server on port `8080` of the local machine.

When the user makes a request[2], the NanoHTTPD framework invokes the `serve` method on a new thread. The native agent receives a callback upon entry to the `serve()` method, which results in the creation of a new symbolic executor for the thread (as described in Section IV-B). The instrumented bytecode in the `serve()` and `getResponse()` methods invokes methods on this `Executor`.

Symbolic variables are introduced in the symbolic shadow stack when variables `a` and `b` are loaded in the `getResponse()` method. When concrete execution reaches a branch that depends on a symbolic variable (marked with comments in Figure 7), the `Executor` uses the collected constraints on the symbolic variables to formulate a symbolic constraint whose solution would drive execution through the opposite branch. For example, with concrete input values `a = 4, b = 1`, when the final `else` branch in the `getResponse()` method is reached, the `Executor` formulates a constraint that would instead explore the conditional branch that returns the "Error!" string. The `Executor` then invokes an SMT solver [8], which reports a solution such as `a = 2, b = 0`, which, when given as input values for the variables `a` and `b` cause the "Error!" string to be returned to the user.

The process of "closing the loop" (i.e., running the program with the newly generated inputs) is currently manual; in the example above, this is done by using the inputs in a new web request to the server.

VI. Benchmarks

This section gives benchmarks and running times for applications of various sizes. We are unaware of a standardized set of web server applications[3] specifically for exercising program analysis tools, and so we give descriptions of the applications we used, including the number of source lines of code and the number of bytecode instructions executed.

Table I describes the applications tested and shows the amount of increase in the code size of the jar file due to the bytecode added by instrumentation. This is a potential issue for some applications because of the 64 kB limit imposed per Java module; this is a limitation of the JVM due to its use of a signed 16-bit offset value for branch statements in Java bytecode. This only allows a branch range of +32767 to -32768 bytes from the current instruction, which would prevent the ability to jump outside of a 64 kB code range.

Note that although the size increases in the table range between 25% and 49%, the individual classes may have a larger range. For instance, the Tollbooth application increased by 32% overall. However, the individual classes increased anywhere between 50% to 140% . This is because some of the contents of the jar file contain data other than classes.

[2]For example: `http://localhost:8080/?a=4&b=1`

[3]These examples will be included with our framework, which we plan on open-sourcing soon.

Application	Description	Methods	Original	Instr	%Increase
ThreadTest	Creates a shared object (integer value) and 2 threads. One thread writes to the shared object and the other reads it.	6	2898	4110	42
ThreadArray	Creates a shared object (array value) and 2 threads. One thread writes to the shared object and the other reads it.	6	2991	4274	46
SimpleNano	The server waits for user input from the network on a separate thread and sets and tests a parameter.	4	2395	3129	31
Tollbooth	A peer-to-peer server using NanoHTTPD used for running automated toll-booths. Vehicle transponders issue commands to debit the account and users can issue payments and view their account.	127	49012	64568	32
LinearAlgebra	An RPC server using NanoHTTPD for processing several common linear algebra and graph analysis operations. The user inputs NxN matrices and the operation to be performed in a JSON request and the server will return the response in a JSON format.	224	69425	98088	41
Stufftracker	A database server using Jetty that allows users to upload, retrieve and search inventory records. The inventory records are saved using an XML-based markup language.	783	213455	275963	29

TABLE I
CODE SIZE INCREASE DUE TO INSTRUMENTATION

Application	Agent calls	% Instr.	Instr. count	Run time increase
ThreadTest	19.5 k	80	58	2.2 x
ThreadArray	21.8 k	91	66	2.8 x
SimpleNano	56.2 k	61	49	3.7 x
Tollbooth	25.9 M	8	22247	8.2 x
LinearAlgebra	9.6 M	50	29057	10.5 x
Stufftracker	42.8 M	77	145081	29.0 x

TABLE II
RUN TIME OVERHEAD DUE TO INSTRUMENTATION

Table II shows the amount of overhead that the agent adds to the execution of the application code. Recall that the main purpose of the agent (Section IV-A) is to properly adjust the symbolic shadow stack when a method is entered or exited. Each method entry/exit callback events consumes additional time in the running application.

Agent calls indicates the number of times the agent's enter/exit callbacks were invoked for cases which are not relevant to maintaining the shadow stack. Although each of these visits to the agent callbacks is small (less than 20 μs), a large number can add an appreciable delay to the valid calls. These "invalid" calls fall into the categories of:

- Calls from uninstrumented methods to other uninstrumented methods
- Calls to methods while processing an Executor method
- Calls to methods while processing ClassLoader processes

The *% Instr.* column indicates the approximate percentage of the total running time that the application spent executing instrumented code. Uninstrumented code only requires instrumentation upon entry and exit from instrumented calls, and therefore does not increase running time as much as instrumented code, in which every opcode executed is impacted in performance.

The *Instruct count* column gives an approximation of the number of execution cycles performed by the application during its run. It is incremented for each opcode executed by the instrumented code only, so this excludes contributions by the uninstrumented code.

The *Run time increase* column indicates the factor increase in running time of the instrumented code versus the uninstrumented code.

Both the percentage of time the application spends in the instrumented portion of the code and the number of times the agent's callback methods are executed impact the speed of running the instrumented code. The JVMTI interface only provides the ability to filter which threads trigger callbacks based on unique thread identifiers, which are not known a priori; knowing these unique identifiers could decrease the agent's contribution to the overall slowdown. The delays due to running instrumented code may also be subject to some optimization that would also improve performance.

VII. RELATED WORK

Symbolic PathFinder (SPF) [2] is a static symbolic execution tool for Java programs. It is built as an extension on top of Java PathFinder (JPF) [16], a custom implementation of the JVM. A concolic extension to JPF, called jFuzz [10], also exists. One advantage of using a custom JVM is that the state of the JVM can be saved and restored efficiently; state saving and restoration has been shown to be a major source of overhead in practice [13].

Another recent DSE tool built on the JPF framework is jDART [13], which uses a solver interface to support several specialized solvers, including CORAL, SMTInterpol and Z3 [8]; our framework currently uses the Z3 solver for constraint solving. The primary design goal of jDART is similar to ours: to build a robust DSE framework capable of analyzing complex, real-world programs. However, to our knowledge, jDART has not been applied to multithreaded programs with callbacks, such as web servers. In general, approaches based on a custom JVM aim to symbolically execute every bytecode instruction; for programs that use large libraries, such as web servers, this can cause scalability issues. Our framework addresses this by allowing the user to select which classes should be symbolically executed.

The Java Bytecode Symbolic Executor (JBSE) [7] is a custom JVM, written in Java, to provide symbolic analysis

for Java programs. The user specifies verification properties as sets of assumptions and assertions; JBSE then attempts to find inputs that satisfy all assumptions while making at least one assertion fail. One limitation of JBSE compared to our approach is that JBSE will not symbolically analyze more than one thread.

A closely related approach is Pex [15], which performs concolic execution of .NET code. Pex instruments the bytecode when it is loaded at runtime, and this instrumentation is then used to drive the execution of a shadow symbolic execution engine. Pex was originally designed for parameterized unit tests, defined as a method that takes parameters, performs a sequence of method calls that exercise the code-under test, and asserts properties of the codes expected behavior. Our framework aims to do dynamic symbolic execution of entire Java programs. One feature of Pex is that it includes bytecode implementations of many of the .NET core library classes that are implemented in lower-level languages such as C, which allows Pex to symbolically execute those core classes. Our framework, on the other hand, explicitly chooses to ignore many classes from symbolic execution for scalability reasons.

Concolic testing tools have been used to analyze multithreaded programs for thread-safety [14], [11]. Our work focuses on input generation for multithreaded programs rather than having an explicit focus on discovering thread-safety and scheduling issues.

Dynamic symbolic execution of binary code has been used at industrial scale to find bugs in real-world programs [6], [3]. Most real-world JVMs extensively use multithreading, even to run a single-threaded program, which makes the symbolic execution of multithreaded Java programs challenging. Our framework addresses this challenge by allowing the user to statically select which classes should be executed symbolically, then using a native runtime agent to create a symbolic executor whenever a thread executes instrumented code.

VIII. Conclusions

We presented a dynamic symbolic execution framework for Java that is capable of running on complex, multithreaded programs. Scalability is increased by allowing the user to statically select which classes are executed symbolically. Our framework uses a combination of static instrumentation and a run-time JVMTI agent to drive a "shadow" symbolic execution engine. This combination allows multithreaded programs, such as web servers, to be analyzed more efficiently than if all threads are executed symbolically. The run-time agent increases robustness in the face of complexities introduced by the implementation of the JVM. Our mechanism for introducing symbolic variables helps in analyzing unmodified programs without the need for source code modification. One practical application of our framework is the analysis of web server programs, which use multithreading extensively.

Our future work includes the addition of a database mechanism into which path constraint solutions are logged and from which program inputs used by the DSE are obtained. Such a database will assist in closing the DSE exploration loop in which generated inputs are fed back into the program. Additionally, a database mechanism will allow us to parallelize our implementation; similar mechanisms are used in other popular program testing tools [1]. Finally, we also plan on open-sourcing our framework, including our web server benchmark programs, in the near future.

Acknowledgment

This material is based on research sponsored by DARPA under agreement number FA8750-15-2-0087. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] American fuzzy lop. http://lcamtuf.coredump.cx/afl/, Accessed on Dec. 14th, 2018.

[2] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to java pathfinder. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007*, pages 134–138, 2007.

[3] T. Avgerinos, D. Brumley, J. Davis, R. Goulden, T. Nighswander, A. Rebert, and N. Williamson. The mayhem cyber reasoning system. *IEEE Security & Privacy*, 16(2):52–60, 2018.

[4] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1083–1094. ACM, 2014.

[5] R. Baldoni, E. Coppa, D. C. Delia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):50, 2018.

[6] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 122–131. IEEE Press, 2013.

[7] P. Braione, G. Denaro, and M. Pezzè. Jbse: A symbolic executor for java programs with complex heap inputs. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 1018–1022, New York, NY, USA, 2016. ACM.

[8] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.

[10] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. jfuzz: A concolic whitebox fuzzer for java. In *First NASA Formal Methods Symposium - NFM 2009, Moffett Field, California, USA, April 6-8, 2009.*, pages 121–125, 2009.

[11] K. Kähkönen, O. Saarikivi, and K. Heljanko. Lct: A parallel distributed testing tool for multithreaded java programs. *Electron. Notes Theor. Comput. Sci.*, 296:253–259, Aug. 2013.

[12] J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, 1976.

[13] K. Luckow, M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamarić, and V. Raman. Jdart: A dynamic symbolic analysis framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 442–459. Springer, 2016.

[14] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In T. Ball and R. B. Jones, editors, *Computer Aided Verification*, pages 419–423, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[15] N. Tillmann and J. De Halleux. Pex–white box test generation for. net. In *International conference on tests and proofs*, pages 134–153. Springer, 2008.

[16] W. Visser and P. C. Mehlitz. Model checking programs with java pathfinder. In *Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings*, page 27, 2005.