# Dynamic Symbolic Execution of Java Programs Using JNI

Sergey Vartanov
Institute for System Programming of the Russian Academy of Sciences,
Lomonosov Moscow State University
Moscow, Russia
e-mail: svartanov@ispras.ru

## ABSTRACT

For the sake of better performance, platform-specific facilities support, or the use of legacy code, Java applications may use JNI (Java Native Interface) to call native functions. We present an approach to perform dynamic symbolic execution of a Java program that tracks tainted data flow through Java bytecode and native code of shared libraries. We propose a tool based on modified virtual machine and static binary code instrumentation. This allows us to collect path constraints for both bytecode and binary code execution but avoid redundant processing of virtual machine own code. Modified path constraints are checked for their satisfiability in order to generate new inputs and execute new paths (to cover new basic blocks of target program). We describe initial experiments with our implementation based on Avian virtual machine and Dyninst.

## Keywords

dynamic symbolic analysis, Java native interface, Java bytecode, path alternation

## 1. INTRODUCTION

As software becomes more complicated, it needs program analysis tools to control efficiency and detect defects. One of these approaches is *dynamic symbolic execution* (sometimes referred to as *concolic execution*). It allows us to construct new input data in order to cover new execution paths of the target program.

Java is one of the most popular programming languages. Programs written in the Java language are usually translated into Java bytecode, which is interpreted by a Java virtual machine. This approach allows us to add a set of convenient features into the language, e.g., automatic memory management, which reduces or fully eliminates such kind of defects as memory leaks and buffer overflows. However, it may significantly decrease the application performance in comparison to well-designed analogous program written in compiled languages. Sometimes, it is convenient to utilize features of both compilation-based and interpretation-based approaches within one software system. Java native interface (JNI) was created as a part of Java virtual machine specification to cover such issues. In this paper we consider approaches to perform comprehensive dynamic symbolic execution (DSE) of Java programs, which use JNI. We consider DSE for both Java bytecode and native code, as well as JNI mechanisms of their connection.

The remainder of the paper is organized as follows. Section 2 describes an approach to dynamic symbolic execution and highlights Java and native code connection issues. Section 3 is about the technical details of our implementation. We describe initial experiments in Section 4, review future work in Section 5 and conclude with Section 6.

## 2. DYNAMIC SYMBOLIC EXECUTION

Dynamic symbolic execution is based on the idea of path alternation. For every given execution path one can choose a branch node of that path and try to direct execution through an alternative branch. The only way to change program execution and preserve its functionality is to modify input data. To do so one should collect path constraints leading to target branch, invert branch condition constraint, and, if modified constraint is satisfiable, construct a new input based on this constraint. To create constraints, that fully designate an execution path and allow us to obtain dependency between the arbitrary branch condition and the input data, we should handle the all instructions, which could affect data and control flow. Constraints are usually collected in the form of satisfiability modulo theories (SMT) formulae. We use SMT solvers to check constraint satisfiability.

### 2.1 Bytecode and Native Code Connection

Many of dynamic symbolic execution tools that analyze Java programs track only bytecode instructions to generate constraints. Thus, any control and data flow dependencies generated by native code are not checked by these tools. This problem can be illustrated on a simple example (see Listing 1 for the Java code and Listing 2 for the C code). Branch condition (`sum == 5`) depends on data flow through native function. If analysis, based on dynamic symbolic execution, does not take into account the native code, it has a very small chance to pick up the needed file content (sum of two bytes should be equal to 5) and will miss the whole program part from line 12 of Java code.

Listing 1. Java part of example

```
1   package test;
2   import test.Util;
3
4   public class Test {
5       public static void
6       main(String[] args) {
7           byte[] buffer = Util.read(2);
8           int i = (int) buffer[0];
9           int j = (int) buffer[1];
10          int sum = nativeSum(i, j);
11          if (sum == 5) {
12              // Do something
```

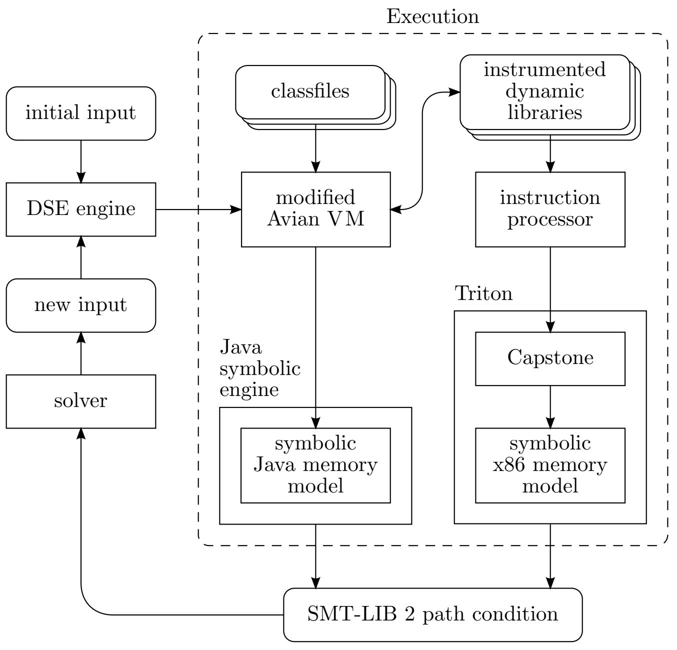

Figure 1. Tool diagram

```
        } else {
            // Do something else
        }
    }
    public static native int
    nativeSum(int i, int j);
}
```

Listing 2. C part of example

```
#include <jni.h>

JNIEXPORT jint JNICALL
Java_test_Test_nativeSum(
        JNIEnv* env, jclass class_,
        jint i, jint j) {
    return i + j;
}
```

The described problem could be solved using a binary analysis. Since Java virtual machine is a program itself, the binary dynamic symbolic execution could be applied to the entire system including virtual machine and libraries. But in such a case we will deal with a large amount of instructions of Java virtual machine code (class loading, memory management, bytecode interpretation, etc.) as well as target program and dynamic libraries instructions.

## 2.2 Approach

We propose to perform a combined dynamic symbolic execution of Java bytecode and native code in the following way. To handle native code instructions we will use static binary instrumentation of dynamically loaded libraries. For Java bytecode instruction handling and taint data flow propagating through JNI we will use a virtual machine modification. The result of the instrumented program execution should include a comprehensive path constraint in the form of a single SMT formula. The described approach allows us to provide analysis in the absence of source code of the target program.

## 2.3 Related Work

There are many program analysis tools for Java (e.g., Java PathFinder [1], GlassTT [2], Javana [3]). Such Java PathFinder plugins as Symbolic PathFinder [4] and JDart [5] provide dynamic symbolic execution of Java bytecode. To the best of our knowledge, these tools have native code support only for custom Java models of binary code. In other words, they do not provide dynamic symbolic execution of arbitrary binary code of `native` methods that called from Java code through JNI.

## 3. IMPLEMENTATION

In order to provide the proof of concept, we have implemented a tool for dynamic symbolic execution of JNI programs. It consists of the following components (see Figure 1): (1) dynamic symbolic execution engine, (2) modified virtual machine, (3) static binary instrumentation system, (4) symbolic and taint engines for Java and native code, (5) solving system.

## 3.1 Java Part

To handle Java bytecode we have modified a lightweight Java virtual machine Avian [6] and have implemented a Java symbolic engine library, that provides bytecode instruction processing and path constraint generation. Java virtual machine modifications include:

- bytecode instruction processing and path constraint collecting,
- argument marshalling mechanism modification to support taint data flow propagation through native function call from Java code,
- modification of JNI function implementation to support taint data flow propagation from native code to Java,
- processing of instructions and methods, that could be sources of tainted data.

**virtual machine**:
two bytes of the input file declaration

```
(declare-fun x_1 () (_ BitVec 8))
(declare-fun x_2 () (_ BitVec 8))
```

**Java bytecode execution**:
reading two bytes from the input file using `readBytes` method of `java.io.FileInputStream` and convert them to integer type

```
(declare-fun x_3 () (_ BitVec 32))
(assert (= x_3 ((_ sign_extend 24) x_1)))
(declare-fun x_4 () (_ BitVec 32))
(assert (= x_4 ((_ sign_extend 24) x_2)))
```

**virtual machine**:
marshalling

```
(declare-fun ref!1 () (_ BitVec 32))
(assert (= ref!1 x_3))
(declare-fun ref!2 () (_ BitVec 32))
(assert (= ref!2 x_4))
```

**binary code execution**:
addition of two integers

*Some asserts that imply `ref!48` = `ref!1` and `ref!50` = `ref!2` are skipped.*

```
(declare-fun ref!52 () (_ BitVec 64))
(assert (= ref!52 ((_ zero_extend 32)
    (bvadd
        ((_ extract 31 0) ref!50)
        ((_ extract 31 0) ref!48)))))
```

**virtual machine**:
demarshalling

```
(declare-fun x_5 () (_ BitVec 32))
(declare-fun x_6 () (_ BitVec 32))
(assert (= x_6 ((_ extract 31 0) ref!52)))
(assert (= x_6 x_5))
```

**Java bytecode execution**:
branch condition

```
T (assert (not (= x_5 #x00000005)))
```

Figure 2. Combined path constraint in the format of SMT-LIB 2

In Avian virtual machine we have modified the function argument marshalling mechanism by informing Triton [7] (see below) that the registers and memory cells contain tainted function arguments and correspondence between them and symbolic variables. This mechanism is used to connect the tainted data flow of Java bytecode with the tainted data flow of native code. Java symbolic engine consists of a bytecode instruction processor and a symbolic Java memory model, that contains a symbolic analogue of stack, local variables, arrays, and fields.

## 3.2 Binary Instrumentation

For instrumentation of binary code we have created a small tool, that uses a binary rewriting mechanism of the powerful Dyninst [8] framework. Dyninst allows us to define instrumentation points as follows: points before every instruction (except some instructions, such as NOP or unconditional jumps) of native library, that could be loaded by virtual machine during target application execution. At the described points the tool processes the following an instruction opcode and inserts instrumentation code. That code extracts the concrete values of registers and memory cells, that could be influenced by the instruction, and inserts a call of instruction handler (see below). Such kind of heavy-weight instrumentation introduces a huge overhead, but allows us to achieve the necessary level of data flow tracking accuracy.

## 3.3 Binary Part

To handle native code instructions we use Triton [7]. It is a tool for symbolic execution of binary code. It disassembles the binary code using Capstone [9], supports taint data tracking and symbolic expression construction based on a comprehensive memory model for x86 and x86-64. Instruction handler, mentioned above, passes information about instruction and concrete values of memory cells and registers to Triton. As a result, Triton generates path condition in the format of SMT-LIB 2. This format is in widespread use and is supported by almost all modern SMT solvers, such as Z3 [10]. One slight modification of Triton allows us to get a path condition with specific markers for conditional branch assertions.

## 3.4 Constraints

As a result of target program execution, we get comprehensive path constraint in the form of one conjunction over a set of Boolean variables. Each conjunct represents a condition, that corresponds to: (1) Java bytecode instruction, (2) native instruction (x86), (3) marshalling and demarshalling, or (4) input data reading function call.

For the example mentioned above (see Listings 1 and 2), we will get a combined path constraint presented in Figure 2.

## 3.5 Dynamic Symbolic Execution Engine and Solving

Dynamic symbolic execution engine is the main part of the tool. It manages other components execution and stores fundamental analysis artefacts: path conditions and inputs. Firstly, the engine executes instrumentator based on Dyninst to instrument the list of specified native libraries. After the static instrumentation phase, it starts iterative analysis loop depicted in Figure 1. The main analysis loop consists of two major phases: (1) path constraint generation and (2) input generation.

In the first phase the engine selects one of the inputs from the storage and executes the modified Avian virtual machine on the given Java bytecode with the use of instrumented native libraries. Generated path constraint is collected in the storage along with path qualifiers, such as code coverage, the number of executed instructions, and the number of tainted conditional branches.

In the second phase the engine selects one of the path

| Project | VectAlign | flexmark-java | `javax.xml` | jsoup |
|---|---|---|---|---|
| Executions | 17639 | 13343 | 437 | 6626 |
| Unique paths | 16561 | 12970 | 212 | 5572 |
| SAT rate | 96.49 % | 98.81 % | 58.09 % | 99.95 % |
| Maximum number of branch conditions | 516 | 178 | 100152 | 99 |
| Maximum number of assertions per query | 1005 | 325 | 200279 | 166 |
| Average number of assertions per query | 269.9 | 93.83 | 85869.26 | 51.90 |
| Solving time | 34.41 % | 4.50 % | 61.61 % | 8.71 % |
| Execution time | 54.18 % | 90.35 % | 14.63 % | 88.52 % |
| Native method calls per execution | 396.61 | 1714.48 | 946.43 | 5023.52 |

Table 1. Experimental results

constraints from the storage (path conditions and inputs are selected according to strategies and heuristics, that could be configured by the user: breadth-first, depth-first search, coverage heuristics), chooses conditional branches, and inverts them. In the example, there is a single branch condition assertion. It was true on the previous execution, therefore we should use an inversion: `(assert (not (not (= x_5 x00000005))))`.

Then the engine executes SMT solver. The solver is used to check satisfiability of the given constraint and to construct the models (set of concrete variable values) of the satisfiable constraint. The models are used to create new input data, that leads to a corresponding execution path. For these ends we use Z3 [10].

## 4. EXPERIMENTAL RESULTS

We have tested our experimental tool on a set of small artificial programs and several real-world projects to evaluate how many unique paths can be handled using such a heavy-weight instrumentation. For all experiments, we used Ubuntu machine with a 3.3 GHz Intel Core i5 and 8 GB RAM. We used one or two input files as tainted data sources and run analysis with a time cap of one hour.

**VectAlign** processes the string representation of two SVG paths and aligns them in order to allow morphing animations. **flexmark-java** is a Java implementation of Markdown parser, **jsoup** is a Java HTML Parser, and we also use XML parser from `javax.xml` package. Table 1 gives our experimental results. The number of generated unique paths is less than the number of executions due to the unaccounted implicit data flows. SAT rate is the ratio of `sat` solver answers, to all solver queries (other cases are `unsat`, `unknown` answers, and timeout). Solving time and execution time percentage show how much time we spend in path condition solving and target program execution, respectively.

## 5. FUTURE WORK

Dynamic symbolic execution is widely used along with different dynamic analysis techniques to solve a variety of problems, e.g., test case generation, defect detection, reverse engineering, and profiling. We are planning to use the described approach for detecting unhandled exceptions in Java code, as well as crashes in native code.

## 6. CONCLUSION

The main contribution of this work is an approach to combine dynamic symbolic execution of Java bytecode and native code based on virtual machine modification and static binary instrumentation. We show that whereas Java bytecode and native code have different behavior and require different instrumentation techniques, there is an approach to propagate tainted data and symbolic variables through foreign function calls.

## REFERENCES

[1] W. Visser, C. S. Păsăreanu, S. Khurshid: Test Input Generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, pp. 97–107, 2004.

[2] R. Mller, C. Lembeck, H. Kuche: A Symbolic Java Virtual Machine for Test Case Generation. 2004.

[3] J. Maebe, D. Buytaert, L. Eeckhout, K. De Bosschere: Javana: A System for Building Customized Java Program Analysis Tools. *SIGPLAN Not.*, pp. 153–168, 2006.

[4] C. S. Păsăreanu, N. Rungta: Symbolic PathFinder: Symbolic Execution of Java Bytecode. *Proc. of the IEEE/ACM Int. Conf. on Aut. Softw. Eng.*, pp. 179–180, 2010.

[5] K. Luckow, M. Dimjašević, D. Giannakopoulou, F. Howar, M. Isberner, T. Kahsai, Z. Rakamarić, V. Raman: JDart: A Dynamic Symbolic Analysis Framework. , pp. 442–459, 2016.

[6] Avian. A lightweight alternative to Java. https://readytalk.github.io/avian/

[7] F. Saudel, J. Salwan: Triton: A Dynamic Symbolic Execution Framework. *Symp. sur la séc. des tech. de l'inf. et des com., SSTIC, France, Rennes, June 3-5 2015*, pp. 31–54, 2015.

[8] B. Buck, J. K. Hollingsworth: An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.*, pp. 317–329, 2000.

[9] Capstone. The Ultimate Disassembler. http://www.capstone-engine.org/

[10] L. De Moura, N. Bjørner: Z3: An Efficient SMT Solver. *Proc. of the Theory and Prac. of Softw., 14th TACAS*, pp. 337–340, 2008.