JUSTGen: Effective Test Generation for Unspecified JNI Behaviors on JVMs

Sungjae Hwang
School of Computing
KAIST
Daejeon, South Korea
sjhwang87@kaist.ac.kr

Sungho Lee
Department of Computer Science and Engineering
Chungnam National University
Daejeon, South Korea
eshaj@cnu.ac.kr

Jihoon Kim
School of Computing
KAIST
Daejeon, South Korea
kjh618@kaist.ac.kr

Sukyoung Ryu
School of Computing
KAIST
Daejeon, South Korea
sryu.cs@kaist.ac.kr

Abstract—**Java Native Interface (JNI) provides a way for Java applications to access native libraries, but it is difficult to develop correct JNI programs. By leveraging native code, the JNI enables Java developers to implement efficient applications and to reuse code written in other programming languages such as C and C++. Besides, the core Java libraries already use the JNI to provide system features like a graphical user interface. As a result, many mainstream Java Virtual Machines (JVMs) support the JNI. However, due to the complex interoperation semantics between different programming languages, implementing correct JNI programs is not trivial. Moreover, because of the performance overhead, JVMs do not validate erroneous JNI interoperations by default, but they validate them only when the debug feature, the *-Xcheck:jni* option, is enabled. Therefore, the correctness of JNI programs highly relies on the checks by the -Xcheck:jni option of JVMs. Questions remain, however, on the quality of the checks provided by the feature. Are there any properties that the -Xcheck:jni option fails to validate? If so, what potential issues can arise due to the lack of such validation? To the best of our knowledge, no research has explored these questions in-depth.**

In this paper, we empirically study the validation quality and impacts of the -Xcheck:jni option on mainstream JVMs using unspecified corner cases in the JNI specification. Such unspecified cases may lead to unexpected run-time behaviors because their semantics is not defined in the specification. For a systematic study, we propose JUSTGEN, a semi-automated approach to identify unspecified cases from a specification and generate test programs. JUSTGEN receives the JNI specification written in our domain specific language (DSL), and automatically discovers unspecified cases using an SMT solver. It then generates test programs that trigger the behaviors of unspecified cases. Using the generated tests, we empirically study the validation ability of the -Xcheck:jni option. Our experimental result shows that the JNI debug feature does not validate thousands of unspecified cases on JVMs, and they can cause critical run-time errors such as violation of the Java type system and memory corruption. We reported 792 unspecified cases that are not validated by JVMs to their corresponding JVM vendors. Among them, 563 cases have been fixed and the remaining cases will be fixed in near future. Based on our empirical study, we believe that the JNI specification should specify the semantics of the missing cases clearly and the debug feature should be supported completely.

Index Terms—**Java Native Interface, Java Virtual Machine, Testing, Empirical Study, Debugging**

I. Introduction

Java developers use the Java Native Interface (JNI) in various application domains including games and multimedia, and mainstream Java Virtual Machines (JVMs) [1]–[4] support the JNI. The JNI is an interface that defines interoperation between Java code and native code written in C or C++. Using the JNI, developers can improve the performance of programs by implementing performance-critical modules in native code and composing them with Java modules into a single program via the JNI. In addition, the JNI reduces software development cost by allowing Java modules to reuse existing native libraries.

However, building correct JNI programs is a difficult task due to the complex interoperation semantics between different languages. In addition, because the JNI specification does not describe the interoperation semantics completely, the semantics of numerous cases are unspecified, which may lead to unexpected behaviors. Note that the JNI does not check for programming errors for the following reasons [5]:

- Forcing JNI functions to check for all possible error conditions degrades the performance of normal (correct) native methods.
- In many cases, there is not enough run-time type information to perform such checking.

While compilers can detect compile-time errors and provide useful debug features for programs written in a single programming language, they cannot find bugs in interoperation between Java and native code. Furthermore, no publicly available tools can detect such interoperation errors.

The JNI supports the *-Xcheck:jni* option to help developers to diagnose problems in JNI programs, but the specification does not define its semantics clearly. The -Xcheck:jni option is a command-line option that causes the VM to do additional validation on the arguments passed to JNI functions [6], [7]. However, the additional validation is not well defined:

> Note: The option is not guaranteed to find all invalid arguments or diagnose logic bugs in the application code, but it can help diagnose a large number of such problems.

Because the specification does not specify which problems the option diagnoses, JNI programs may behave differently on different JVMs depending on their implementation of the option, which makes reasoning of JNI programs challenging.

In this paper, we study the semantics of the -Xcheck:jni option and its impacts on mainstream JVMs. Our approach is to generate test programs for "unspecified cases," which are interoperation semantics that are not defined in the JNI specification, to execute the test programs on JVMs with the -Xcheck:jni option enabled, and to inspect the execution results. Our study has two technical challenges: 1) how to identify unspecified cases in the JNI specification, and 2) how to generate test programs that trigger the behaviors of unspecified cases. For a systematic study, we propose JUSTGEN, a semi-automated approach to identify unspecified cases and generate test programs. We first define a domain specific language (DSL) that can express the JNI interoperation semantics such as the return types and parameters of JNI functions. Then, we manually transform the interoperation semantics written in a natural language in Chapter 4 of the JNI specification [8] to a mechanized specification expressed in our DSL. Because the English phrases used to specify the interoperation semantics are well structured and use specific patterns, manually converting them to the DSL is considerably straightforward. Then, JUSTGEN receives the mechanized specification, automatically extracts unspecified cases from the specification, and generates test programs that provoke them. JUSTGEN leverages an SMT solver to find unspecified cases by verifying whether the specification describes all the conditions of JNI function calls. If JUSTGEN identifies a condition of a JNI function call that the specification does not describe, it considers the condition as an unspecified case. For test code generation, JUSTGEN takes an unspecified case and synthesizes C code consisting of JNI function calls with arguments that satisfy the conditions of the unspecified case. It then compiles and links the synthesized C code with prepared Java modules to generate an executable JNI program.

Using 34,990 test programs generated by JUSTGEN, we empirically evaluated the -Xcheck:jni option on five mainstream JVMs. Our study shows that thousands of unspecified cases are not validated by JVMs, and they can cause critical run-time errors such as run-time type errors and memory corruption. In addition, we found a bug of the -Xcheck:jni option, which leads to deadlock between multiple threads. We also observed that the -Xcheck:jni option of HotSpot, Zulu, Corretto, and GraalVM validate similar properties, but OpenJ9 validates properties significantly different from them. We reported the problems of the -Xcheck:jni option to their JVM vendors, and among 792 reported unspecified cases, 563 cases have been fixed. The tool used for the empirical study and the identified unspecified cases are publicly available[1].

The contributions of this paper include the following:

- **We present an approach to identify unspecified cases from a specification and implement JUSTGEN that automatically identifies them from a mechanized specification and generates test code provoking them.** We believe that JUSTGEN is applicable to other specifications with only changes in test code generation.
- **We identify unspecified cases from the JNI specification.** Describing the semantics of the identified unspecified cases in the JNI specification would make consistent behaviors of JNI programs on different JVMs.
- **It is the first work that analyzes the quality of the -Xcheck:jni option on five mainstream JVMs.** Our empirical study reports limitations of the -Xcheck:jni option, and JVM vendors fixed 563 among 792 reported cases. We believe that our work would be helpful in enhancing the quality of the JNI debug feature on JVMs, which in turn improves the quality of JNI programs.

[1]https://github.com/sjmini/justgen.

```
1  public class HelloJNI{
2    static{ System.loadLibrary("Hello"); }
3    private native String foo();
4    public static void main(String[] args){
5      String n = new HelloJNI().foo();
6      ... }
7    private String name(){
8      return this.getClass().getName(); }
9  }
```

(a) Java code

```
1  jmethodID get_name_id(JNIEnv *env, jobject obj){
2    jclass cls = (*env)->GetObjectClass(env, obj);
3    return (*env)->GetMethodID(env, cls, "name",
         "()LJava/lang/String;");
4  }
5  jstring Java_HelloJNI_foo(JNIEnv *env, jobject
       thisObj){
6    jmethodID mid = get_name_id(env, thisObj);
7    return (*env)->CallObjectMethod(env, thisObj,
         mid);
8  }
```

(b) Normal JNI interoperation in C code

```
1  jstring Java_HelloJNI_foo(JNIEnv *env, jobject
       thisObj){
2    jmethodID mid = get_name_id(env, thisObj);
3    jcharArray arr = (*env)->NewCharArray(env, 2);
4    return (*env)->CallObjectMethod(env, arr,
         mid);
5  }
```

(c) Unspecified JNI interoperation case in C code

Fig. 1: JNI code example for normal and unspecified behaviors

II. Background and Motivating Example

A. JNI Interoperation

The JNI is a foreign function interface that enables bidirectional interoperation between Java and native applications. Figure 1(a) shows Java code that has an entrypoint of a JNI program, and Figure 1(b) shows C code compiled to a native library `Hello.so`. In Java, a class `HelloJNI` has two Java methods, `main` and `name`, and a native method `foo` declared with the `native` keyword. The native method is linked with a C function `Java_HelloJNI_foo`, when executing the `System.loadLibrary` method at line 2. When the program runs, the `main` method calls the native method `foo` at line 5. Then, the JVM transfers the program control to the entry of the linked C function `Java_HelloJNI_foo`. In C code, the function `Java_HelloJNI_foo` calls the Java method `name` via a sequence of three JNI function calls. At line 6 in Figure 1(b), the C code obtains a Java method ID by calling `get_name_id` defined at line 1. The function calls the `GetObjectClass` JNI function that takes a `JNIEnv` pointer and a Java object and returns class information of the Java object. Then, it calls `GetMethodID` to obtain a Java method ID using the class information, a method name, and a method signature. Using the method ID and the Java object, it calls the Java method `name` via the `CallObjectMethod` JNI function.

B. Unspecified Cases and the JNI Debug Feature

Because JVMs do not validate argument values at run-time due to its performance overhead, the unspecified cases may lead to unexpected behaviors or even security vulnerabilities like memory corruption. Figure 1(c) shows one unspecified case for the JNI function `CallObjectMethod`. The function `Java_HelloJNI_foo` gets an ID of the Java method `name` via the `get_name_id` function as the same as (b). However, at line 4, it tries to call the Java method `name` using a Java character array object created at line 3 instead of the Java object propagated from Java. The JNI specification describes the behavior of `CallObjectMethod` as follow [8]:

```
NativeType Call<type>Method(JNIEnv *env,
        jobject obj, jmethodID methodID, ...);
```

> Methods from these three families of operations are used to call a Java instance method from a native method. $\cdots$ the `methodID` must be derived from the real class of `obj`, not from one of its superclasses.

However, in the example, because the method ID is derived from `HelloJNI` while a class of `this` is a character array, its behavior is not specified, which can cause problems on mainstream JVMs. For example, calling `CallObjectMethod` at line 4 triggers a segmentation fault on the HotSpot JVM, but executes normally on the OpenJ9 JVM producing a wrong result. After normal execution, the variable `n` has a string value `[C` at line 5 in (a), instead of `HelloJNI`, which is wrong in the Java semantics, because a `this` object in a Java method must be an instance of a class having the method.

To prevent segmentation faults or abnormal executions from JVMs due to the unspecified JNI interoperation semantics, the JNI provides a debug feature, the *-Xcheck:jni* option, but it does not guarantee to find all problems and its semantics depends on the implementation on JVMs. The -Xcheck:jni option enables JVMs to validate arguments passed to JNI functions [7]. If argument values are not valid, JVMs stop execution and report errors or warnings. For example, when executing the example in Figure 1(c) on the OpenJ9 JVM with the -Xcheck:jni option enabled, the JVM detects the invalid argument and throws an exception with the error message: `JNI error in CallObjectMethod/CallObjectMethodV: Ineligible receiver`. However, because the debuggability highly relies on JVMs, invalid interoperation in JNI programs may still remain undiscovered after the debugging process, which degrades the quality of JNI programs.

In this paper, we leverage unspecified cases to evaluate the quality of the -Xcheck:jni option on mainstream JVMs. We believe that unspecified cases are useful resources to evaluate the -Xcheck:jni option since developers might have missed them because they are not described in the specification.

III. Methodology

A. Overview

To evaluate the quality of the -Xcheck:jni option, we test JVMs with unspecified cases that are semi-automatically extracted from the JNI specification. Figure 2 presents an overview of our testing approach consisting of two phases.

In the *JNI Unspec. Code Generation Phase*, JUSTGEN extracts unspecified cases from the JNI specification and generates test programs that provoke the behaviors of the unspecified cases. Since the specification is written in a natural language that is not suitable for automated processing, our first step is to define a domain specific language (DSL) and to manually transform the semantics in the specification to a mechanized specification in the DSL. Then, Unspec. Extractor discovers unspecified cases from the mechanized specification by utilizing an SMT solver. It encodes the mechanized specification as logical formula in a way that the SMT solver can recognize, and leverages the SMT solver to determine whether a formula is unsatisfiable. If the formula is satisfiable, the SMT solver generates a counterexample of such a case. The counterexample denotes an unspecified case with a possible argument combination. Similar to CounterExample-Guided Abstraction Refinement (CEGAR) [9], Unspec. Extractor updates the specification by adding the identified unspecified case, and repeats the above process until the formula becomes unsatisfiable. Among various SMT solvers, we used Z3 [10]. For each unspecified case, Test Code Generator automatically generates a sequence of valid JNI function calls that triggers the behavior of the unspecified case. Then, it composes the JNI function calls with a Java and C template code that we defined.

In the *JVM Testing Phase*, we execute the generated test programs on mainstream JVMs with the -Xcheck:jni option enabled. Because different JVMs may produce different results, we manually inspect the test results and analyze the capability of the debug feature of each JVM, and identify potential hazards due to the lack of validation from JVMs.

B. JNI Specification in a Domain Specific Language

For automatic processing of the JNI specification, we define a simple DSL to describe the behaviors of JNI functions. Because the expected behaviors of JNI functions are defined in Chapter 4 of the JNI specification [8], we manually transformed them to a mechanized specification expressed in the DSL. Since the JNI function behaviors are written in a well-structured phrases using specific patterns, manually converting them to the DSL is considerably straightforward.

Figure 3 presents the DSL syntax. A specification s is a sequence of type declarations $\overline{\texttt{typedef}\ t}$, a sequence of refinement predicate declarations $\overline{\texttt{refinedef}\ t@p}$, and a sequence of JNI function specification $\overline{d}$. Types t denote types in the

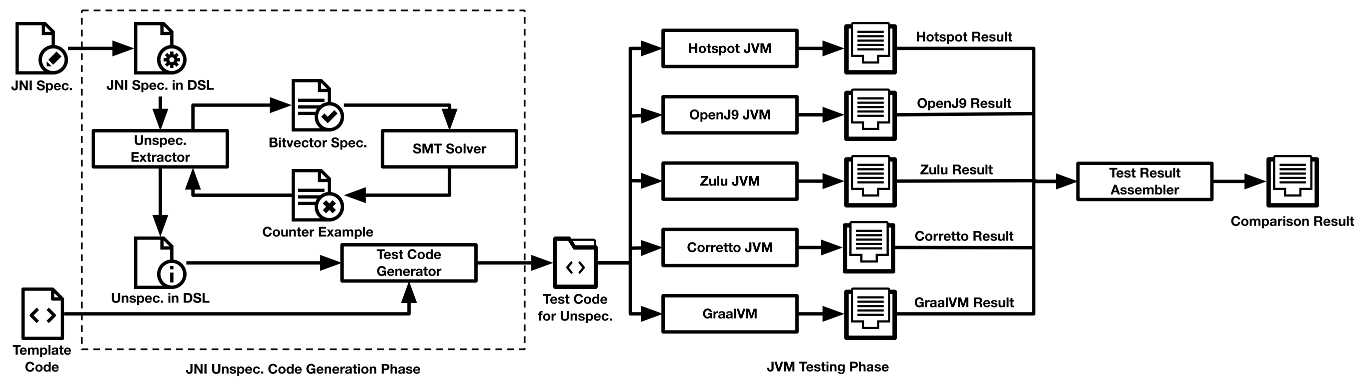

Fig. 2: Overall structure of JVM testing with unspecified cases

```
rule{
   type   void SetIntArrayRegion(JNIEnv*, jintArray, jsize, jsize, jint*)
   spec   void SetIntArrayRegion(JNIEnv*, jintArray@NotNULL, jsize@ValidIndex, jsize@ValidIndex,
                                 jint*@NotNULL)
   unspec void SetIntArrayRegion(JNIEnv*, jintArray@isNULL,  jsize@ValidIndex, jsize@ValidIndex,
                                 jint*@NotNULL)
   ...
}
```

Fig. 4: Example JNI function specification in DSL

$$\begin{array}{lcl}
s & := & \overline{\texttt{typedef}\ t}\ \ \overline{\texttt{refinedef}\ t@p}\ \ \bar{d} \\
d & := & \texttt{rule}\ \{\texttt{type}\ t\ F(\bar{t})\ \ \overline{\kappa\ t[@r]^?\ F(\overline{t[@r]^?})}\} \\
\kappa & := & \texttt{spec}\ \mid\ \texttt{unspec} \\
r & := & p\ \mid\ r \wedge r
\end{array}$$

Fig. 3: Domain specific language to define the JNI semantics

C programming language including primitive types and predefined types for the JNI in `jni.h`. A refinement predicate p denotes a predicate name, which returns whether an input satisfies its condition. A refinement type [11] $t@p$ denotes a set of values that have the type t and satisfy the refinement predicate p. For example, a set of all negative integer values is denoted as `int@isNegative` where `isNegative` is a predicate returning whether an input is a negative integer. A JNI function specification d consists of a JNI function descriptor and a sequence of refined function descriptors. A JNI function descriptor `type` $t\ F(\bar{t})$ denotes a return type t, a JNI function name F, and a sequence of parameter types $\bar{t}$. A refined function descriptor $\kappa\ t[@r]^?\ F(\overline{t[@r]^?})$ denotes an optionally refined return type $t[@r]^?$, a JNI function name F, and a sequence of optionally refined parameter types $\overline{t[@r]^?}$ where κ is either `spec` or `unspec`. A refinement r is a refinement predicate p or its conjunctions. For our research, we defined 38 types and 105 refinement types [12], [13]. To distinguish function descriptors JVMs do not verify return values of JNI functions, but because we use return types in test code generation as described in Section III-D, we include return types in refinement types. specified in the specification and those constructed while finding unspecified cases using the SMT solver, we use `spec` for the former and `unspec` for the latter.

Figure 4 shows an example JNI function specification in DSL. The JNI function `SetIntArrayRegion` takes a JNI environment pointer `JNIEnv*`, a Java array of integers `jintArray`, an integer value representing the start index of an array `jsize`, an integer value indicating the number of elements to be copied `jsize`, and the source buffer (`jint*`). The function copies a specified number of elements from the source buffer to the Java array. According to the JNI specification, the Java array must not be `NULL`, the start index and the number of elements must be greater than or equal to zero, and the source buffer must not be `NULL` as well. We transform the above specification in the DSL as the `spec` statement with refinement predicates representing the conditions for valid argument values. In this example, `NotNULL` denotes that input values should not be `NULL`, and `ValideIndex` denotes that input values must be greater than or equal to zero. In addition to the `spec` statement, various `unspec` statements are automatically generated in the process of finding unspecified cases of the JNI function as we discuss in the next subsection.

C. Finding Unspecified Behaviors with an SMT Solver

We leverage the Z3 SMT solver version 4.8.1 to find unspecified cases automatically from the mechanized JNI specification. To use the SMT solver, we convert the unspecified case finding problem to an SAT problem. For example, for a JNI function specification `spec` $t_1@r_1\ F(t_2@r_2)$, we extract a boolean formula as follows: is there a refined type x that is matched with the type t_2 but is not covered by the specification $t_2@r_2$? When the formula is satisfiable, the SMT solver produces an example satisfiable cases; the produced example is an unspecified case. Then, we update the boolean formula including the unspecified case to find another unspecified case. We repeat the process to find unspecified cases until the boolean formula is not satisfiable, which means that no more unspecified cases exist.

1) Bit vector representation for JNI function specification: We use bit vectors to represent each JNI function specification in a boolean formula so that the SMT solver can manipulate it. A JNI function specification consists of a possibly refined return type, a function name, and a list of possibly refined parameter types. Since only the parameters determine the behavior of the JNI function, we can encode a JNI function specification into a list of bit vectors where each bit vector represents each possibly refined parameter type.

Definition 1 (Bit vector representation):

- *(Type)* Each type t is mapped to a unique bit vector representation B_t.
- *(Refinement predicate)* For each refined type $t@r$, each refinement predicate p in r is mapped to a unique bit vector B_p whose *on-bits* do not overlap with other bit vectors representing refinement predicates of t.
- *(Refinement)* For a refinement $r = p_1 \wedge ... \wedge p_n$, its bit vector representation is $B_r = B_{p_1} \mid B_{p_n}$ where $|$ is the bitwise OR operator.
- *(JNI function specification)* For a JNI function specification $t_1@r_1\ F(t_2@r_2, ..., t_n@r_n)$, its bit vector representation is ((B_{t_2}, B_{r_2}), ..., (B_{t_n}, B_{r_n})) where B_{t_i} and B_{r_i} are bit vector representations of t_i and r_i, respectively.

One challenge for the bit vector representation is to define subtype relations among types. While C does not have any subtype relations between types, JNI reference types have a type hierarchy that corresponds to the Java type hierarchy [14]. For example, because `jstring` is a subtype of `jobject`, a JNI function that takes a `jobject` argument can take a `jstring` value as well.

We define a *subtype relation* ($<:_t$) between two bit vectors using the bitwise AND operator &. For the previous example, we encode `jobject` and `jstring` into two different bit vectors, B_{t_o} and B_{t_s}, which satisfy $B_{t_o}\ \&\ B_{t_s} = B_{t_o}$.

Definition 2 (Subtype relation): Assume that t_1 is a subtype of t_2 and B_{t_1} and B_{t_2} are bit vector representations of t_1 and t_2, respectively. Then, a subtype relation $B_{t_1} <:_t B_{t_2}$ is valid. The subtype relation is equivalent to a boolean expression, $B_{t_1}\ \&\ B_{t_2} = B_{t_2}$.

Similarly, we also define a *subrefinement relation* ($<:_r$) between two bit vectors representing refinements. A refinement r_1 is a subrefinement of another refinement r_2, when all the refinement predicates in r_1 are included in r_2.

Definition 3 (Subrefinement relation): Assume that r_1 is a subrefinement of r_2 and B_{r_1} and B_{r_2} are bit vector representations of r_1 and r_2, respectively. Then, a subrefinement relation $B_{r_1} <:_r B_{r_2}$ is valid. The subrefinement relation is equivalent to a boolean expression, $B_{r_1}\ \&\ B_{r_2} = B_{r_2}$.

2) Satisfiability for unspecified case finding: Using encoded JNI function specifications, we convert each JNI function specification to a boolean formula to find unspecified cases. One complexity is that because most JNI functions take multiple parameters, we should consider many parameter combinations. For example, if a JNI function takes five parameters and each parameter has ten refined types, the number of possible parameter combinations is 10^5, which is a huge search space.

For practicality, we handle each parameter independently to reduce the search space. For a JNI function F with n parameters, we make n functions $F_1, \cdots, F_n$ where F_i takes only the i-th parameter of F. Then, each function F_i gets converted to a boolean formula, and the SMT solver finds unspecified cases for each boolean formula independently. This approach may miss some unspecified cases that are caused only by a combination of multiple parameters. However, our experiments showed that because most unspecified cases are due to a single parameter, few such cases are missing.

After defining four auxiliary definitions, we define a boolean formula to check the satisfiability of a JNI function parameter using the auxiliary definitions. Note that we now consider only such JNI functions that take one parameter.

Definition 4 (Complete refinement): For each type t, its *complete refinement* bit vector is B_c^t that is a conjunction of all the refinement predicates of t.

Definition 5 (Validity of refined type): A bit vector representation of a refined type (B_{t_x}, B_{r_x}) is *valid* for a type t, if $B_{t_x} <:_t B_t$ and $B_{r_x} <:_r B_c^t$. A logical predicate $valid((B_{t_x}, B_{r_x}), t)$ is *true* only when (B_{t_x}, B_{r_x}) is *valid* for the type t.

Definition 6 (Equivalent type satisifiability): A bit vector representation of a refined type (B_{t_x}, B_{r_x}) is *equivalent type satisfiable* to another bit vector representation (B_{t_y}, B_{r_y}), iff $B_{t_x} = B_{t_y}$ and $B_{r_x} <:_r B_{r_y}$. A logical predicate $sat_{eq}((B_{t_x}, B_{r_x}), (B_{t_y}, B_{r_y}))$ is *true* only when (B_{t_x}, B_{r_x}) is *equivalent type satisfiable* to (B_{t_y}, B_{r_y}).

Definition 7 (Subtype satisifiability): A bit vector representation of a refined type (B_{t_x}, B_{r_x}) is *subtype satisfiable* to another bit vector representation (B_{t_y}, B_{r_y}), iff $B_{t_x} <:_t B_{t_y}$ and $B_{t_x} \neq B_{t_y}$. A logical predicate $sat_{sub}((B_{t_x}, B_{r_x}), (B_{t_y}, B_{r_y}))$ is *true* only when (B_{t_x}, B_{r_x}) is *subtype satisfiable* to (B_{t_y}, B_{r_y}).

Using the auxiliary definitions, we define a boolean formula for the parameter satisifiability solved by the SMT solver. Assume that a JNI function declaration is $t_1\ F(t_2)$ and the bit vector representation of its specification is (B_{t_y}, B_{r_y}). Then, we can define its satisfiability problem as follows:

$$\begin{aligned} \exists(B_{t_x}, B_{r_x}).\ & valid((B_{t_x}, B_{r_x}), t_2) \rightarrow \\ & \neg(\ \ sat_{eq}((B_{t_x}, B_{r_x}), (B_{t_y}, B_{r_y})) \\ & \quad \vee sat_{sub}((B_{t_x}, B_{r_x}), (B_{t_y}, B_{r_y}))) \end{aligned}$$

The formula represents that there is a valid parameter (B_{t_x}, B_{r_x}) that is matched with the JNI function parameter type t_2 but not covered by the specification (B_{t_y}, B_{r_y}). When the SMT solver concludes that this formula is satisfiable, it produces an example such as (B_{t_z}, B_{r_z}) as a result representing an unspecified case that is not covered by the current specification. Then, we record the unspecified case and update the formula by adding it as follows:

$$\begin{aligned}\exists(B_{t_x}, B_{r_x}).\ valid((B_{t_x}, B_{r_x}), t_2) \rightarrow \\ \neg(\ (\ \ sat_{eq}((B_{t_x}, B_{r_x}), (B_{t_y}, B_{r_y})) \\ \vee sat_{sub}((B_{t_x}, B_{r_x}), (B_{t_y}, B_{r_y}))) \\ \vee (\ \ sat_{eq}((B_{t_x}, B_{r_x}), (B_{t_z}, B_{r_z})) \\ \vee sat_{sub}((B_{t_x}, B_{r_x}), (B_{t_z}, B_{r_z}))))\end{aligned}$$

Thus, in the next iteration, the SMT solver tries to find another unspecified case except for ones discovered previously. When the SMT solver fails to find more unspecified cases, it concludes that the formula is not satisfiable, implying that the updated specification covers all the valid parameters.

D. Test Case Generation and Testing on JVMs

One difficulty in the test code generation phase is to generate *valid* test code. As described in Section II, in JNI programs, C code leverages a sequence of JNI function calls to interact with Java modules. To investigate behaviors caused by unspecified cases, generated test code should contain JNI function call chains that do not introduce run-time errors caused by other factors such as invalid arguments, unintended unspecified cases, and so on. Random test generation is widely used, but it is not suitable for our purpose; it produces too many wrong JNI function call chains that cause run-time errors unrelated to unspecified cases.

The Test Code Generator utilizes both "a JNI function mapping table" and "template code" to generate valid test code. Firstly, it constructs a mapping table by matching each refined parameter type of JNI functions with a refined return type of other JNI functions. For example, if a first refined parameter type of a JNI function `foo` is matched with a refined return type of another JNI function `bar`, the mapping `bar` → `foo@1` is added to the table; it denotes that the return value of `bar` can be used for the first argument of `foo`. JNI functions often take values of primitive types such as integers and character arrays as arguments, and some primitive types may not have matching JNI functions that have them as return types. Therefore, for primitive types, we manually write template code that consists of functions that return values of primitive types according to refined parameter types. With the mapping table and the template code, Test Code Generator generates valid C test code for unspecified cases by building valid JNI function call chains and propagating valid arguments for refined parameter types. We compile the generated C test code and link it with Java modules we developed to build executable JNI test programs. Note that we focus on only C test code while JNI supports both C and C++ because all the mainstream JVMs handle the JNI interoperation for C and C++ in the same way.

For testing on multiple JVMs, we install each JVM on an individual Docker container [15] and execute the generated test code on each container. Because different JVMs print execution results in different formats, we develop a result parser for each JVM to translate its execution result to a unified format. Then, Test Result Assembler classifies the execution results into four categories: *Misbehave*, *SegFault*, *Exception* and *Validation*. The Misbehave category represents that test code terminates normally without any warnings or errors. Since behaviors of unspecified cases are not defined, each JVM may produce different execution result from one another for an unspecified case. The SegFault and Exception categories represent that test code causes a segmentation fault and an exception thrown, respectively, and the Validation category represents that test code terminates with an error or a warning produced by the debug feature as its validation result. In addition, Test Result Assembler compares the execution results of the JVMs to identify different debugging capabilities of the JVMs.

TABLE I: Evaluation results of the -Xcheck:jni option on five JVMs for 34,990 unspecified cases

Category	**HotSpot**	**OpenJ9**	**Zulu**	**Corretto**	**GraalVM**
Misbehave	4,922	445	4,918	4,918	4,918
SegFault	1,050	567	1,050	1,050	1,010
Exception	24,338	23,377	24,339	24,339	24,339
Validation	4,680	10,601	4,683	4,683	4,723

IV. Evaluation of the JNI Debug Feature on JVMs

A. Evaluation Results for Unspecified Cases

To evaluate the debug capability of mainstream JVMs, we chose five JVMs based-on their popularity [16]: Oracle's Hotspot, IBM's OpenJ9, Azul's Zulu, Amazon's Corretto, and Oracle's GraalVM. Among various Java versions, we chose Java 11, since its popularity is ranked the second following Java 8 and all five JVMs have implementations for Java 11.

Table I shows the categorized evaluation results on the five JVMs for the test programs. JUSTGEN generated 34,990 test programs for unspecified cases from the JNI specification within 403 seconds. As Table I indicates, the evaluation results are very similar between JVMs except for OpenJ9, because they are variants of OpenJDK. Even though they have similar evaluation results, we believe that analyzing them is significant because different vendors support them. OpenJ9 validated much more unspecified cases than the others, but it validated only 30.3% of the unspecified cases. The other JVMs validated only 13.4%. In most cases, when JVMs do not validate unspecified cases, they let test programs terminate with exceptions. Throwing exceptions may be helpful for developers diagnose the problems, but segmentation faults may not be useful to point out where the problems occur. Moreover, misbehaviors may make the problems undiscovered, which can lead to unexpected behaviors.

In the following subsections, we report our empirical study results for unspecified cases in the categories of misbehaviors and segmentation faults. In addition, we discuss the differences in the debug capability of the JVMs, the unspecified cases that were not validated by all of the JVMs, and threats to validity.

B. Category: Misbehave

We manually investigated the unspecified cases in the Misbehave category. Since there are many unspecified cases as shown in Table I, it is impractical to investigate them all. Instead, we chose one unspecified case for each parameter of JNI functions. In addition, for JNI function families that provide the same functionality for different types such as `CallVoidMethod`, `CallIntMethod`, and so on, we randomly selected one of them. Finally, we manually investigated 132 unspecified cases for Hotspot, Zulu, Correctto, and GraalVM, but only 17 cases for OpenJ9 because it has a small number of Misbehave cases.

1) In All the JVMs: We found two cases where all five JVMs failed to identify unspecified cases.

Finding 1: Error handling using return values of JNI functions is not reliable.

Some JNI functions return values that indicate the success or failure of their execution, but they may not return correct results. One example is `RegisterNative` that registers native functions. According to the specification, the function should take a positive number of native functions, and return 0 on success and a negative value on failure. However, all five JVMs failed to validate the case when the function takes no native functions. Moreover, in such cases, while OpenJ9 returns `-1`, the other JVMs return `0` indicating the execution success. Thus, relying on return values of JNI functions to handle execution failures is not reliable.

Finding 2: Deleted references are not completely validated.

Using deleted objects may lead to unexpected behaviors, but JVMs do not validate it completely even with the debug feature enabled. For instance, `GetObjectRefType` receives a Java object and returns the reference type of the object. One of our test code passed a deleted Java object as an argument to `GetObjectRefType`, which is an unspecified case. Even though the JNI specification states that `GetObjectRefType` should not use deleted objects, we found that the function returns the object type as a local reference on OpenJ9, while the other JVMs successfully detect the case. We reported the problem to IBM and it is now fixed. On the contrary, when a deleted object is stored in an array using `SetObjectArrayElement`, OpenJ9 detects this case but the other JVMs cannot detect it. Thus, all the VMs failed to validate them completely.

2) In four JVMs: We report six cases where all JVMs except OpenJ9 failed to identify unspecified cases.

Finding 3: Methods may be treated as constructors.

According to the JNI specification, the name of constructors should be `<init>`. However, our test code revealed that when we call `GetMethodID` with a method whose name is `NULL`, the function returns the method ID of a constructor on all JVMs except for OpenJ9. We found out that the JVMs return the method ID of a constructor only if there is a constructor that satisfies the method signature passed to `GetMethodID` as a fourth argument.

Finding 4: Java objects may be incorrectly initialized.

In JNI programs, native code can create Java objects using `NewObjectV`. While the specification states that the method ID of a constructor should be passed to `NewObjectV`, all JVMs except OpenJ9 cannot detect such cases when the method ID of a non-constructor is used and simply create an object using the non-constructor.

Finding 5: Native code may call Java methods with ill-typed Java objects.

We observed several unspecified cases where native code call a Java method with a Java object that has a different type from what the method expects. One such an example is calling a Java method with a receiver object of an incorrect type using `CallNonvirtualIntMethod`. All JVMs except OpenJ9 cannot detect such cases. Even though Java provides strong type checking, native code can break its type system.

Finding 6: Array elements may be updated with values of incompatible types.

We observed another case that JNI programs break the Java type system. The `ReleaseBooleanArrayElements` JNI function takes a Java array object as a second argument, and a pointer to array elements as a third argument. If this function takes 0 as its fourth argument, it copies the values of the pointer to the Java array object and frees the memory pointed by the pointer. According to the specification, the pointer used by `ReleaseBooleanArrayElements` must be derived by the `GetBooleanArrayElements` function. However, one of our test code passed a pointer that is derived by the `GetIntArrayElements` function to `ReleaseBooleanArrayElements`, and all JVMs except OpenJ9 could not detect it and copied the values of the pointer to the Java array object with an incompatible type.

Finding 7: JNI functions may change return values of Java methods.

While native code should use `CallIntMethod` to call Java methods that return integers, our test code invoked `CallIntMethod` with a Java method that returns values of type `Float`, which was detected only by OpenJ9. In all the other JVMs, the return value of the Java method was unexpectedly changed. For example, when a Java method returns 3.5f of type `Flot`, the native code invoking `CallIntMethod` returned 1080033280, which is very different from 3.5f.

Finding 8: An object can be popped from a local reference frame even when no local reference frame exists on the stack.

The `PopLocalFrame` JNI function takes a local reference object, pops off the current local reference frame, and returns a local reference in the frame for the given object.

However, we found that if a global reference is passed to `PopLocalFrame`, the JVMs except for OpenJ9 behave abnormally: i) the global reference object was returned when the function is supposed to return local reference objects only, and ii) the object was returned even when no local reference frame exists on the stack.

3) Only in OpenJ9: We found four cases where only OpenJ9 failed to identify unspecified cases. We reported them all to IBM, and they are all fixed [2].

Finding 9: Invalid Java objects can be constructed with incorrect class names, and they lead to segmentation faults in subsequent JNI function calls.

The `FindClass` JNI function takes a fully-qualified class name and returns a class object, but OpenJ9 constructs Java class objects even with incorrect class names. Our test code called `FindClass` with an incorrect class name, and OpenJ9 constructed a class object and returned it to native code without any error or warning. However, once the JNI program uses the class object, it results in a segmentation fault.

Finding 10: Local references can be created with a negative capacity.

According to the specification, `EnsureLocalCapacity` should take the capacity of a local reference, which must not be negative. However, our auto-generated test code called `EnsureLocalCapacity` with -5 for the capacity, which was OpenJ9 did not detect. OpenJ9 silently generated a new local reference object without any error or warning.

Finding 11: Invalid JNI function calls may be silently ignored.

We observed that OpenJ9 sometimes silently ignore unspecified JNI function calls. One such an example is to change the value of a static field of an object using `SetObjectField`, which should change the value of only non-static fields. OpenJ9 did not catch this unspecified case but silently ignored the illegal operation.

Finding 12: JNI Programs may not terminate.

We found one bug in the debug feature of OpenJ9 that caused JNI programs to not terminate. When our test code releases `NULL` as array elements using the `Release<type>ArrayElements` JNI function, the code does not terminate because of a deadlock between multiple threads. One thread holds a VM access and tries to enter `MemMonitor`, while the other thread already in `MemMonitor` tries to acquire the VM access to exit JVM. As we discussed, IBM fixed the bug after our report.

C. Category: SegFault

In addition to abnormal behaviors described so far, we observed that unspecified cases can cause segmentation faults due to the lack of validation from the -Xcheck:jni option. The root cause of the segmentation faults is accessing illegal memory locations, and attackers can exploit such defects to hijack the control flows of the programs [17]. Therefore, we consider them as potential security vulnerabilities.

[2]https://github.com/sjmini/justgen/blob/main/README.md.

Table I shows the number of unspecified cases that caused segmentation faults for each JVM. As the table shows, 567 unspecified cases provoked segmentation faults in OpenJ9, 1,011 cases in GraalVM, and 1,051 cases in the other JVMs. We manually investigated them to understand which properties were not validated, and Table II summarizes the results.

We categorized the segmentation faults into four reasons: *Type Check*, *NULL Check*, *Liveness Check*, and *Releasability Check*. For OpenJ9, the lack of `NULL` Check is the main reason with 74.5% of segmentation faults. For the other JVMs, the primary cause of the segmentation faults is the lack of Type Check; 76.1% of the segmentation faults on HotSpot, Zulu, and Corretto, and 75.2% on GraalVM are due to missing type checks. The second major reason for segmentation faults on OpenJ9 is missing Liveness Check. Especially, when JNI functions use deleted references, only OpenJ9 cannot detect them, which amounts to 11.8% of the segmentation faults on OpenJ9. Moreover, we also observed unspecified cases where unreleasable arrays are released by JNI functions such as `ReleaseCharArrayElements`. Due to the space limitation, we provide detailed information on each unspecified case in a companion report [18]. We found that these unspecified cases can be detected only by OpenJ9, and the other JVMs throw segmentation faults. We reported the segmentation fault problems to JVM vendors and 415 unspecified cases have been fixed so far and more fixes are on their way.

Another interesting point we found is that HotSpot, Zulu, and Corretto have the same number of segmentation faults. Our analysis on these cases concluded that the -Xcheck:jni option of these JVMs validate the same properties. On the other hand, OpenJ9 behaves quite different from the other JVMs. Overall, the -Xcheck:jni option on OpenJ9 validates more properties than the other JVMs. In the next subsection, we further analyze the differences between JVMs.

D. Differences between JVMs

We now compare the debug capability of five JVMs. As discussed in the previous subsection, the debug capability of OpenJ9 is different from that of the other JVMs. On the contrary, the other four JVMs show similar debug capability.

Our evaluation results show that there are 6,499 unspecified cases that OpenJ9 validates correctly, but HotSpot misses. Conversely, there are 578 unspecified cases that HotSpot validates, but OpenJ9 misses. For those 6,499 and 578 unspecified case, we chose representative case as we did in Section IV-B and manually investigated them to understand the debug capability of Hotspot and OpenJ9. The analysis results are summarized in Table III, where O indicates that the -Xcheck:jni option of the JVM properly validates the unspecified case, and X indicates that the JVM does not validate the unspecified case. The table clearly shows which unspecified cases can or cannot be validated by HotSpot and

TABLE II: Root causes of segmentation faults

Root Causes	Unspecified Cases	HotSpot	OpenJ9	Zulu	Corretto	GraalVM
Type Check	Call ill-typed objects' methods	720	0	720	720	680
	Define classes with ill-typed classloaders	0	40	0	0	0
	Use reflection with ill-typed objects	80	38	80	80	80
`NULL` **Check**	Call methods with `NULL` arguments	30	30	30	30	30
	Call methods using a `NULL` method ID	0	372	0	0	0
	Access fields using a `NULL` field ID	18	18	18	18	18
	Get fields of a `NULL` object	0	1	0	0	0
	Get field/method IDs using a `NULL` signature	166	0	166	166	166
	Get field/method IDs and classes from a `NULL` object	29	0	29	29	29
	Get strings into a `NULL` destination buffer	1	1	1	1	1
Liveness Check	Call objects' methods using deleted references	0	60	0	0	0
	Compare types between a live and a deleted reference	0	3	0	0	0
	Set fields with deleted references	0	1	0	0	0
	Define classes with deleted a reference name	0	1	0	0	0
	Get field/method IDs using a deleted reference	0	2	0	0	0
Releasability Check	Release unreleasable arrays	6	0	6	6	6
Total		1,050	567	1,050	1,050	1,010

TABLE III: Debug capability of Hotspot and OpenJ9

Causes	Unspecified Cases	Hot	J9
Type	Define classes with wrong typed classloaders	O	X
	Get classes using a wrong class descriptor	O	X
	Throw exceptions using non-throwable objects	O	X
	Call methods with unmatched return types	X	O
	Create objects with array classes	X	O
	Use reflection for fields using wrong typed objects	X	O
`NULL`	Iniailize objects using a `NULL` method ID	O	X
	Get fields of a `NULL` object	O	X
	Call methods of a `NULL` object	X	O
	Call methods using a `NULL` method ID	O	X
	Get field/method IDs of a `NULL` object	X	O
	Get field/method IDs using a `NULL` signature	X	O
	Get field/method IDs using a `NULL` name	X	O
	Release a `NULL` string	X	O
Liveness	Get types from deleted references	O	X
	Put deleted references into an array	X	O
	Call objects' methods using deleted references	O	X
	Compare types between a live and a deleted reference	O	X
	Pop a local frame from an empty frame stack	X	O
Modifier	Access non-static fields using a static field ID	O	X
	Access static fields with a non-static field ID	O	X
	Call private methods of super classes	X	O
Negative Integer	Set a negative capacity of local frames	O	X
	Access a negative index of arrays	X	O
Constructor	Initialize objects using non-constructor methods	X	O
Releasability	Release unreleasable arrays	X	O

OpenJ9. It shows three new causes: *Modifier*, *Negative Integer*, and *Constructor*. The Modifier cause denotes when JVMs do not check the correct use of modifiers such as `static`, and `public`. For example, such a case when a JNI function uses a `static` field ID where a `non-static` field ID is expected belongs to the Modifier cause. OpenJ9 cannot validated them whereas Hotspot can. The Negative Integer problem is when a negative number is used where a non-negative number is expected. For instance, we found that OpenJ9 cannot identify creation of a local frame with a negative capacity, while Hotspot validates it. We also observed an inverse case. OpenJ9

TABLE IV: Unspecified cases not validated by JVMs

Causes	Unspecified Cases
`NULL` **Check**	Call methods with a `NULL` argument list
	Access static fields using a `NULL` field ID
	Copy a string value into a `NULL` buffer
Size Check	Push a zero-sized local frame into a frame stack
	Register zero number of native functions
Type Check	Use reflection for methods using wrong typed objects

throws an error message when accessing a negative index of array, but Hotspot cannot validate it. The Constructor problem is initialization of Java objects with non-constructor functions. OpenJ9 detects the problem when a non-constructor method ID is used to create a Java object, but Hotspot cannot.

We found no unspecified cases that HotSpot validates, but Zulu, Corretto, and GraalVM misses. However, we found three unspecified cases that Zulu and Corretto validate, but HotSpot misses. Similarly, we found 43 unspecified cases that GraalVM validates, but HotSpot misses. We manually investigated all those 3 and 43 unspecified cases, and our analysis revealed that they are due to the use of deleted objects and ill-typed objects, respectively.

Our evaluation results show that the debug features of different JVMs check different properties, and therefore validating JNI programs on multiple JVMs especially OpenJ9 and one of the others would be desirable. We provide detailed description of each unspecified case in a companion report [18].

E. No Validation from Any JVMs

We observed that 23,880 unspecified cases are not validated by any JVMs. Among them, all JVMs throw exceptions for 23,367 cases, but no JVMs throw exceptions for the remaining 513 cases. We believe that they are problematic cases because even the -Xcheck:jni option cannot validate them on any JVMs. We further analyzed them to understand what properties are not validated by any JVMs. The analysis results are summarized in Table IV.

As shown in the table, their root causes have three categories. The first category is missing NULL check. We found that a large number of cases are due to the missing NULL check of jvalue. It seems that no JVMs properly validate argument lists like jvalue. Because dereferencing null values may cause significant errors, we believe that all JVMs should validate them. The other two categories are missing size check and type check. For example, the FromReflectedMethod JNI function expects to receive an object of type java.lang.reflect.Method/Constructor. If it receives an object of different type, JVMs should throw an error. However, no JVMs validates this property. It is particularly strange that OpenJ9 does not validate such a case because it throws an error for similar cases. For instance, the FromReflectedField JNI function does not receive an object of type java.lang.reflect.Field, OpenJ9 throws an error with the following message: *Argument #2 is not a subclass of java/lang/reflect/Field.* OpenJ9 should throw an error with an appropriate message for the FromReflectedMethod function as well.

F. Threats to Validity

To evaluate the quality of the -Xcheck:jni option on mainstream JVMs, we leveraged unspecified cases. We adopted a systematic and automated approach to find unspecified cases from the JNI specification. The only manual part in finding unspecified cases is transforming the JNI specification into our DSL. Manual transformation may contain human errors leading to incorrect transformation. However, we believe that our transformation is correct because we considered only Chapter 4 of the JNI specification, which is written in a well-structured style using specific patterns. Furthermore, we manually investigated automatically produced results by JUSTGEN. Our manual investigation confirmed that the results are accurate, which indicates that the JNI specification is correctly transformed. The results of our manual evaluation are trustworthy because external reviewers such as JVM vendors validated them as well. Note that if the specification is written in a structural way like the JavaScript specification, it may be possible to make JUSTGen fully automatic [19].

Finally, because the transformed specification in the DSL is publicly available, researchers can reuse and validate it. Note that JUSTGEN does not prove the correctness of *-Xcheck:jni* implementations, but tests them, which may lead to false positives and false negatives.

V. RELATED WORK

JVM Verification. Testing JVMs to ensure that they work consistently with each other is crucial to support Java's slogan "Write Once Run Anywhere." The existing research proposed methodologies to automatically generate test cases for verification of JVMs. Sirer and Bershad [20] developed *lava*, a domain specific language for specifying production grammars to generate test cases for a JVM. They demonstrated the effectiveness of *lava* when it is used with other testing techniques by showing high code and value coverage achievements. Yoshikawa et al. [21] implemented a generator for Java classes that is finite and executable on the Java runtime environment. Similarly, Chen et al. [22] introduced classfuzz, which generates invalid Java classes to reveal defects in the startup processes of JVMs. classfuzz leverages predefined mutation operators to mutate seeding Java classes and Markov Chain Monte Carlo (MCMC) sampling to select appropriate mutation operators. On the other hand, classming [23] also proposed by Chen et al generates executable Java classes in order to verify execution engines of JVMs. classming also leverages mutation operators to change the control and data flows of bytecode to generate semantically different test cases. Freund and Mitchell [24] introduced a type system to generate Java classes that can test an implementation of a bytecode verifier.

Even though the above research can generate efficient test cases for verifying startup processes, execution engines, and a bytecode verifier of JVMs, they are not suitable for verifying implementations of JNI interoperations on JVMs. Indeed, no research exists that proposes an efficient technique to generate test cases for testing JNI interoperations.

JNI Verification Researchers have tried to improve the safety and reliability of JNI programs. Several papers address the discrepancies in exception handling of Java and native code. The exceptions from native code of JNI programs do not get handled by JVMs immediately, but they are handled by JVMs only after the native code finishes its execution. Such different semantics can lead to mishandling of exceptions in JNI programs. Li and Tan [25] proposed a static analysis framework that examines exceptions and identifies bugs in JNI programs. Tan [26] proposed an operational semantics for a core of the JNI. The work includes the formal semantics of handling shared heap during cross-language function calls, exception handling, and garbage collection. Because programs written in C may be unsafe due to the lack of type system while programs written in Java are type-safe, native code written in C may make JNI programs unsafe. To ensure type safety of JNI programs, Tan et al. [27] proposed SafeJNI. Tan and Croft [28] also performed an empirical study of native codes in JDK. Leveraging a large number of native code in JDK with various static analysis techniques and manual inspection, the authors identified new bugs and security issues in the native code of JDK. Gu et al. [29] demonstrated another security issues in JNI programs. They introduced the JGRE attack and proposed a defense mechanism. Wei et al. [30] proposed a static analysis framework that can analyze JNI programs. They introduced $\mathcal{JN}$*-SAF* that performs an inter-language dataflow analysis to find security flaws in JNI programs. Lee et al. [31] devised a static analysis technique utilizing both modular and whole program analyses to analyze multilingual programs. Using the technique, they proposed a static analyzer that detects possible programmer errors in JNI interoperation.

Even though the existing research attempted to improve the safety and reliability of JNI programs, none of them systematically studied the unspecified behaviors in the JNI specification and their actual implementations on mainstream JVMs.

VI. Conclusion

JVMs provide a powerful run-time debug feature, the -Xcheck:jni option, for developers to detect interoperation bugs in JNI programs. However, some bugs may remain unidentified even after intensive testing, due to the insufficient validation capability of the debug feature. In this paper, we present a semi-automated approach to evaluate the validation capability and impacts of the -Xcheck:jni option on various JVMs. From the JNI specification, we manually translated the core functionality of JNI functions to our DSL, JUSTGen automatically discovers unspecified cases and generates test code executing the cases. Then, we execute the test code on five mainstream JVMs and categorize the execution results to assess their validation capability. In our empirical evaluation, JUSTGen discovered 34,990 unspecified cases that the JNI specification does not cover. We identified that 5,972 unspecified cases could not be validated by Hotspot's debug feature, 5,968 by Zulu and Corretto, 5,928 by GraalVM, and 1,012 by OpenJ9. These unspecified cases can ciritical run-time errors due to violation of the Java type system and memory corruption. We reported the validation issues of the debug feature to corresponding JVM vendors with 792 unspecified cases, and 563 cases among them have been fixed and the remaining ones are planned to be fixed. We believe that our study improves the validation capability of the -Xcheck:jni option and the quality of JNI programs.

Acknowledgment

This work was supported by National Research Foundation of Korea (NRF) (Grants NRF-2017R1A2B3012020 and 2017M3C4A7068177).

References

[1] Oracle, "Java SE HotSpot at a Glance," https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html, 2019.

[2] Eclipse, "OpenJ9," https://www.eclipse.org/openj9/, 2020.

[3] Azul System, "Azul Zing," https://www.azul.com/products/zing/, 2020.

[4] Oracle, "GraalVM," https://www.graalvm.org/, 2020.

[5] ——, "Java SE Documentation - JNI Design Overview," https://docs.oracle.com/en/java/javase/11/docs/specs/jni/design.html, 2018.

[6] IBM, "Troubleshooting: JNI Checklist," https://www.ibm.com/support/pages/troubleshooting-jni-checklist, 2020.

[7] Oracle, "The -Xcheck:jni Option," https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/clopts002.html, 2020.

[8] ——, "Java SE Documentation - JNI Functions," https://docs.oracle.com/en/java/javase/11/docs/specs/jni/functions.html, 2018.

[9] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided Abstraction Refinement," in *International Conference on Computer Aided Verification*. Springer, 2000, pp. 154–169.

[10] Microsoft Research, "The Z3 Theorem Prover," https://github.com/Z3Prover/z3, 2020.

[11] T. Freeman and F. Pfenning, "Refinement Types for ML," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, 1991, pp. 268–277.

[12] Sungjae Hwang, "JNI Specification Expressed in Our DSL," https://github.com/sjmini/justgen/blob/main/test/rule, 2021.

[13] ——, "Explanation of Refinements," https://github.com/sjmini/justgen/blob/main/test/refine_meaning, 2021.

[14] Oracle, "Java SE Documentation - JNI Types and Data Structures," https://docs.oracle.com/en/java/javase/11/docs/specs/jni/types.html, 2018.

[15] Docker Inc., "Docker," https://www.docker.com/, 2020.

[16] JReBel, "2020 Java Technology Report," https://www.jrebel.com/blog/2020-java-technology-report, 2020.

[17] OWASP, "Buffer Overflow," https://owasp.org/www-community/vulnerabilities/Buffer_Overflow, 2020.

[18] Sungjae Hwang Sungho Lee, Jihoon Kim, Sukyoung Ryu, "A Study of Unspecified Cases in the JNI Specification," https://github.com/sjmini/justgen/blob/main/supplementary/, 2021.

[19] J. Park, J. Park, S. An, and S. Ryu, "Jiset: Javascript ir-based semantics extraction toolchain," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 647–658.

[20] E. G. Sirer and B. N. Bershad, "Using Production Grammars in Software Testing," *ACM SIGPLAN Notices*, vol. 35, no. 1, pp. 1–13, 1999.

[21] T. Yoshikawa, K. Shimura, and T. Ozawa, "Random Program Generator for Java JIT Compiler Test System," in *Third International Conference on Quality Software, 2003. Proceedings.* IEEE, 2003, pp. 20–23.

[22] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed Differential Testing of JVM Implementations," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 85–99.

[23] Y. Chen, T. Su, and Z. Su, "Deep Differential Testing of JVM Implementations," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1257–1268.

[24] S. N. Freund and J. C. Mitchell, "A Type System for the Java Bytecode Language and Verifier," *Journal of Automated Reasoning*, vol. 30, no. 3-4, pp. 271–321, 2003.

[25] S. Li and G. Tan, "Finding Bugs in Exceptional Situations of JNI Programs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009, pp. 442–452.

[26] G. Tan, "JNI Light: An Operational Model for the Core JNI," in *Asian Symposium on Programming Languages and Systems*. Springer, 2010, pp. 114–130.

[27] G. Tan, A. W. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, and D. Wang, "Safe Java Native Interface," in *Proceedings of IEEE International Symposium on Secure Software Engineering*, vol. 97. Citeseer, 2006, p. 106.

[28] G. Tan and J. Croft, "An Empirical Security Study of the Native Code in the JDK." in *Usenix Security Symposium*, 2008, pp. 365–378.

[29] Y. Gu, K. Sun, P. Su, Q. Li, Y. Lu, L. Ying, and D. Feng, "JGRE: An Analysis of JNI Global Reference Exhaustion Vulnerabilities in Android," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 427–438.

[30] F. Wei, X. Lin, X. Ou, T. Chen, and X. Zhang, "JN-SAF: Precise and Efficient NDK/JNI-aware Inter-anguage Static Analysis Framework for Security Vetting of Android Applications with Native Code," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1137–1150.

[31] S. Lee, H. Lee, and S. Ryu, "Broadening horizons of multilingual static analysis: Semantic summary extraction from c code for jni program analysis," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 127–137.