JET: Exception Checking in the Java Native Interface

Siliang Li
Lehigh University
sil206@cse.lehigh.edu

Gang Tan
Lehigh University
gtan@cse.lehigh.edu

Abstract

Java's type system enforces exception-checking rules that stipulate a checked exception thrown by a method must be declared in the `throws` clause of the method. Software written in Java often invokes native methods through the use of the Java Native Interface (JNI). Java's type system, however, cannot enforce the same exception-checking rules on Java exceptions raised in native methods. This gap makes Java software potentially buggy and often difficult to debug when an exception is raised in native code. In this paper, we propose a complete static-analysis framework called JET to extend exception-checking rules even on native code. The framework has a two-stage design where the first stage throws away a large portion of irrelevant code so that the second stage, a fine-grained analysis, can concentrate on a small set of code for accurate bug finding. This design achieves both high efficiency and accuracy. We have applied JET on a set of benchmark programs with a total over 227K lines of source code and identified 12 inconsistent native-method exception declarations.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.12 [*Software Engineering*]: Interoperability

General Terms Languages, Reliability

Keywords Exception Checking, Java Native Interface, Static Analysis

1. Introduction

The Java programming language has a strong type system that makes it a type-safe language. Among many other things, its type system performs compile-time checking of exceptions. Specifically, a Java compiler enforces that a checked exception must be declared in a method's (or constructor's) `throws` clause if it is thrown and not caught by the method (or constructor). When used properly, checked exceptions improve program reliability by enabling the compiler to identify unhandled exceptional situations during compile time.

The Java Native Interface (JNI) allows Java programs to interface with low-level C/C++/assembly code (i.e., native code). The JNI provides convenience and efficiency in the Java software development. It allows Java code to invoke and to be invoked by native-code modules. A native method is declared in a Java class by adding the `native` modifier. For example, the `ZipFile` class in Fig. 1 declares a native method named `open`. Once declared, native methods are invoked in Java in the same way as how Java methods are invoked. In the example, the `ZipFile` constructor invokes the native `open` method.

Native-method implementation can interact with Java through a set of JNI interface functions (called JNI functions hereafter) as well as using all the features provided by the native language. Through JNI functions, native methods can inspect, modify, and create Java objects, invoke Java methods, and so on.

What is relevant for this paper is that native methods can also throw, handle, and clear Java exceptions through JNI functions. However, a Java compiler does not perform static exception checking on native methods, in contrast to how exception checking is performed on Java methods. This lack of static exception checking on native methods might lead to unexpected crashes in Java programs. Let us use the program in Fig. 1 as an example. The Java-side declaration of `open` does not have any `throws` clause, leading programmers and the compiler to think that no checked exceptions can be thrown. However, the C-code implementation does throw an `IOException`, violating the declaration. This cannot be detected by a Java compiler and will escape from detection of Java's type system. Consequently, when the `ZipFile` constructor invokes the native `open` method, the constructor is not required to handle the exception or declare it. But in reality a thrown `IOException` in `open` crashes the program, out of the expectation of programmers and the compiler. This example was actually a real bug in `java.util.zip.Zipfile` of Sun's Java Development Kit (JDK); it was fixed only recently in JDK6U23 (JDK 6 update 23) by adding a "*throws IOException*" clause to `open`.

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-0940-0/11/10. . . $10.00

Java code

```
class ZipFile {
  // Declare a native method;
  // no exception declared
  private static native long open
    (String name, int mode, long lastModified);

  public ZipFile (...) {
    // Calling the method may crash the program
    ...; open(...); ...
  }

  static {System.loadLibrary("ZipFile");}
}
```

C code

```
void Java_ZipFile_open (JNIEnv *env, ...){
  ...
  // An exception is thrown
  ThrowIOException();
  ...
}
```

Figure 1: An example demonstrating how a native method can violate its exception declaration.

DEFINITION 1. *A native method has an inconsistent exception declaration if one or more checked exceptions that can be thrown from its implementation is not a subclass of the exceptions declared in the method's Java signature.*

This type of bugs can be difficult to debug when they occur. The JVM will report a Java stack trace through which programmers can know the exception originates in a native method. However, since the exception may be thrown due to nested function calls in native code, it is still difficult to locate the source of the exception without knowing the native call stack. JNI debuggers (e.g., Blink [12] and others [5]) can help, but not all bugs manifest themselves during particular program runs.

The major contribution of this paper is the design and implementation of JET, a complete tool that extends Java's exception checking mechanism to cover native code. JET statically analyzes and examines JNI code for finding exception inconsistencies between native method declarations and implementations. It makes sound engineering trade-offs to achieve both high efficiency and accuracy.

JET was evaluated on a set of benchmark programs from the Java Development Kit (JDK) and other sources with a total over 227K lines of C code. In all, 12 errors were identified.

The rest of the paper is structured as follows: we provide background information next. Details of JET's design are discussed in Section 3. Prototype implementation and evaluation are presented in section 4. We discuss closely related work in Section 5, and conclude in section 6.

2. Background

Checked vs. unchecked exceptions. Java uses exceptions to handle errors and exceptional situations. An exception that may be thrown by a Java method is an *effect* of the method. Therefore, we use the term *exception effects* to refer to the set of exceptions that may escape a Java method.

There are two kinds of exceptions: *checked exceptions* and *unchecked exceptions*[1]. Checked exceptions are used to represent those error conditions that a program can recover from and are therefore statically checked by Java's type system. That is, a checked exception that can escape a Java method (or constructor) has to be a subclass of the exception classes declared in the method's (or constructor's) `throws` clause. By contrast, unchecked exceptions represent irrecoverable situations and are not statically checked. Following Java's practice, JET performs exception checking only on checked exceptions. Nevertheless, JET can be easily adapted to infer the set of unchecked exceptions that are possibly pending in native methods; this information would be useful to check if exceptions (either checked or unchecked) are handled properly in native methods [15]. For convenience, we use exceptions and checked exceptions interchangeably in the rest of the paper unless specially noted.

How JNI exceptions can be thrown and handled. Both Java code and native code may throw exceptions. There are many ways that an exception may become pending in the native side:

- Native code can throw exceptions directly through JNI functions such as `Throw` and `ThrowNew`.
- Many JNI functions throw exceptions to indicate failures. For instance, `NewCharArray` throws an exception when the allocation fails.
- A native method can call back a Java method through JNI functions such as `CallVoidMethod`. The Java method may throw an exception. After the Java method returns, the exception becomes pending in the native side.

All these situations result in a pending exception on the native side. In the rest of this paper, we will use the term *JNI exceptions* for those exceptions that are pending on the native side, while using the term *Java exceptions* for those pending on the Java side.

When a JNI exception is pending, native code is supposed to perform either one of the following things:

[1] Java's unchecked exceptions include `Error`, `RuntimeException` and their subclasses; all other exception classes are checked exceptions

- Perform some clean-up work (e.g., freeing buffers) and return the control to the Java side. Then the exception handling mechanism of the JVM takes over.
- Handle and clear the exception on the native side using certain JNI functions. For example, `ExceptionClear` clears the pending exception; `ExceptionDescribe` prints information associated with the pending exception; `ExceptionOccurred` checks if an exception is pending.

Interface vs. library code. Native code associated with a JNI package can roughly be divided into two categories: *interface code* and *library code*. The library code is the code that belongs to a common native library. The interface code implements Java native methods and glues Java with C libraries through the JNI. For example, the implementation of the Java classes under `java.util.zip` has a thin layer of interface code that links Java with the popular zlib C library. Typically, the size of interface code is much smaller than the size of library code. It is also worth mentioning that some JNI packages include only interface code. For example, native code in `java.io` directly invoke OS system calls for I/O operations and does not go through a native library.

3. Design of JET

JET is a static-analysis framework that examines exception effects of a native method's implementation and reports warnings if any inconsistency is found between the actual exception effects and the declared exception effects. The use of static analysis allows us to examine every control flow path in a program and to catch all potential bugs. This would have been more difficult if not impossible if using dynamic analysis.

3.1 System architecture

Fig. 2 presents a high-level diagram that depicts JET's architecture. At a high level, JET takes as input a JNI code package, which is comprised of both Java class files and native code. The package is fed into a two-stage exception analysis. For each native-method implementation, the exception analysis outputs its actual exception effects. Afterward, JET compares the actual effects with the declared exception effects extracted from Java class files. If there is a mismatch, the warning generator issues a warning. Note that our analysis does not examine Java code and examines only native code and type signatures in Java class files. The Java compiler already enforces Java code is well typed.

A two-stage design. The core of JET is a two-stage exception analysis, motivated by the following two observations.

(1) Library code in a JNI package does not affect Java's state. As discussed in the background section, native code in a JNI package consists of interface and library code. Intuitively, a general-purpose library such as the zlib C library works independently of Java. Only interface code, which bridges between Java and the library, inspects and changes Java's state via JNI functions. In particular, only interface code will throw, clear, or handle JNI exceptions.

(2) Accurately computing exception effects of native methods requires fine-grained static analysis. The need for this will be explained in more detail. But at a high level, the analysis needs path sensitivity since JNI code frequently correlates its exception state with other parts of the execution state; it needs context sensitivity because JNI code often invokes utility functions that groups several JNI function calls.

JET uses a two-stage design to take advantage of these observations. The first stage of JET's exception analysis is a coarse-grained analysis that aims to separate interface code from library code automatically. It is conservative and quickly throws away code that is irrelevant for calculating exception effects. Afterward, the second stage can focus on the remaining smaller set of interface code. The second stage is much slower as it needs the full set of sensitivity for accurate analysis. It also performs exception-effect lookups in Java class files since native code can call back Java methods, which may throw exceptions.

3.2 Separating interface and library code

The first stage of JET's exception analysis is a crude, flow-insensitive analysis with the goal of separating interface and library code. This analysis is helped by JNI's design philosophy that interaction with the JVM in native code is through a set of well-defined JNI interface functions.[2] Accordingly, a native function is defined as part of the interface code if

(1) it invokes a JNI function, or

(2) it invokes another native function that is part of the interface code.

If a native function does not belong to the interface code (by the above definition), then its execution will not have any effect on the JVM's exception state. A straightforward worklist algorithm that iterates through all functions is implemented to calculate the set of interface code.

We note that the above definition of interface code covers the case of calling back a Java method since a Java call back is achieved through invoking a JNI function such as `CallVoidMethod`. This implies that any native function that performs a Java call back is part of the interface code. Calling-back-into-Java is rare in some JNI packages, but can be common in others. For instance, the `sun.awt` JNI package of JDK6 has almost one hundred call-back functions.

[2] There are exceptions; for instance, native code can have a direct native pointer to a Java primitive-type array and read/rewrite the array elements through the pointer. Nevertheless, JVM's exception state (the focus of this paper) can be changed only by JNI functions.

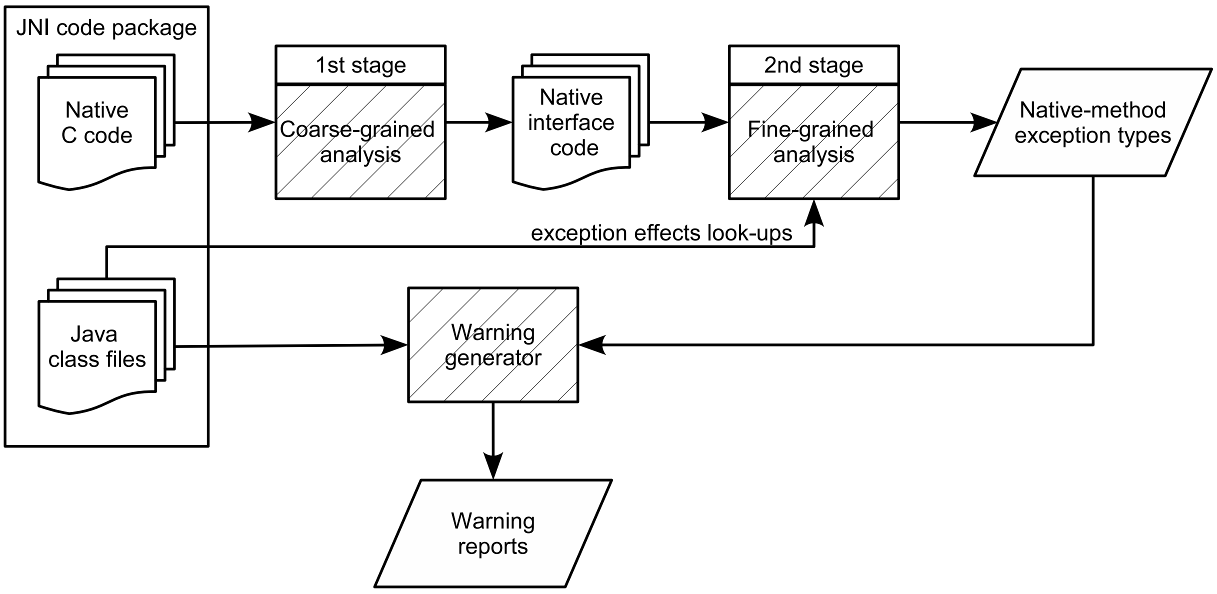


Figure 2: System architecture of JET.

Having the first-stage exception analysis helps JET achieve high efficiency, as our experiments demonstrate (see Sec. 4). Briefly, without the first-stage analysis, JET's analysis time would be increased by about 60 times on the set of JNI programs we studied. Another option that could avoid the first-stage analysis is to manually separate interface and library code. Manual separation is straightforward to perform in practice as a JNI package is usually organized in a way that its library code is in a separate directory. Nevertheless, it has two downsides. First, the manual process may make mistakes and code that does affect Java's state would be wrongly removed. Second, since manual classification most likely uses a file as the unit, it would include irrelevant functions into interface code when they are in files that mix relevant and irrelevant functions. Our first-stage analysis uses functions as the classification unit and provides automatic classification.

3.3 Fine-grained tracking of exception states

At the second stage, JET applies a fine-grained analysis that is path-sensitive and context-sensitive. By applying the fine-grained analysis on the remaining interface code, JET can precisely identify the exception effects at each control-flow edge.

We next discuss requirements of performing accurate tracking before presenting our solutions.

3.3.1 Requirements of JET's exception analysis

The first requirement is that the analysis needs path sensitivity because JNI programs often exploit correlation between exception states and other execution states. For instance, many JNI functions return an error value and at the same time throw an exception to signal failures (similar situation occurs in the Python/C API). As a result of this correlation between the exception state and the return value, JNI programs can either invoke JNI functions such as `ExceptionOccured` *or* check the return value to decide on the exception state. Checking the return value is the preferred way as it is more efficient. Fig. 3(a) presents such an example involving `GetIntArrayElements`, which returns the null value and throws an exception when it fails.[3]

Invoking `GetIntArrayElements` results in two possible cases: an exception is thrown and `p` equals `NULL`; no exception is thrown and `p` equals non-null. That is, the value of `p` is

(a)

```
int *p = GetIntArrayElements(arr, NULL);
if (p == NULL) /*exception thrown*/
  return;
for (i=0; i<10; i++) sum += p[i];
```

(b)

```
void ThrowByName(const char* cn){
  jclass cls = FindClass(cn);
  if(cls != NULL){
    ThrowNew(cls);
  }
}
```

Figure 3: Examples illustrate the need for path and context sensitivity.

[3] JNI programs use a JNI environment pointer to access JNI functions; to avoid cluttering, the use of the environment pointer is removed in all examples.

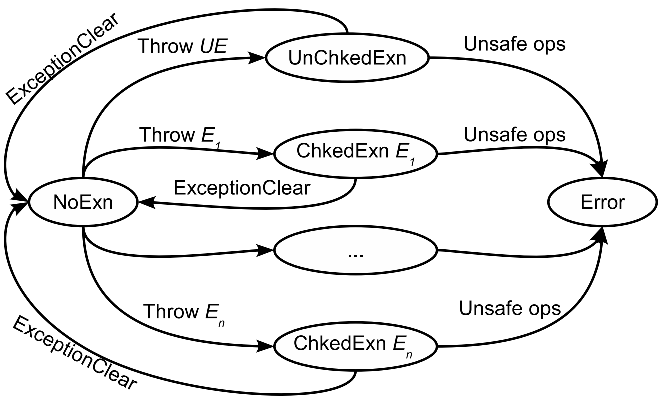


Figure 4: An FSM specification of exception states. Only the transitions of `Throw`, `ExceptionClear`, and unsafe operations are included.

correlated with the exception state. To infer correctly that the state before the loop is always a no-exception state, a static analysis has to be path sensitive, taking the branch condition into account.

The second requirement is that the analysis also needs context sensitivity because JNI programs often use utility functions to group several JNI function calls. JNI programming is tedious; a single operation on the Java side usually involves several steps in native code. For instance, the function in Fig. 3(b) uses a two-step process for throwing an exception: first to get a reference to the class of the exception and then to throw the exception using the reference.

For convenience, JNI programmers often use these kinds of utility functions to simplify programming. What exception is pending after an invocation of the function in Fig. 3(b) depends on what the context passes in as the argument.

Clearly, it is not possible for JET to infer in every case the exact exception effects. For instance, suppose a JNI program calls back a Java method and JET cannot determine which Java method it is. In such cases, JET has to be conservative and assume any kind of exceptions can be thrown. In the analysis, this is encoded by specifying the exception state afterwards becomes `java.lang.Exception`, the root class of all checked exceptions.

3.3.2 An FSM specification of exception-state transitions

Before we introduce JET's (second-stage) exception analysis, we first present an FSM (Finite State Machine) that specifies transitions of exception states; this is used in JET's static analysis to track how exception states change.

JNI functions can change JVM's exception state. For instance, an exception becomes pending after `Throw`; after `ExceptionClear`, the current pending exception is cleared. Transitions between exception states can be described by an FSM. Fig. 4 presents a partial FSM specification. The following table summarizes the meaning of the states in the FSM:

`NoExn`	No JNI exception is pending
`ChkedExn`(E)	A checked exception E is pending
`UnChkedExn`	There is an unchecked exception pending
`Error`	A pending exception is not properly handled

Since JET is primarily concerned with checked exceptions, the FSM has a state for each specific type of checked exceptions but has only one state that collectively represents all unchecked exceptions.

The error state is used for capturing mistakes of mishandling JNI exceptions. The JNI requires very strict rules on handling JNI exceptions. For instance, after a JNI exception is pending, calling `Throw` again results in undefined behavior; the original exception must be properly handled first. Our previous work [15] proposed a framework for identifying bugs of mishandled JNI exceptions. JET's focus is to compute exception effects of native methods, which the previous system cannot do due to its simpler static analysis. We next discuss these differences between the two work in providing the design rational of JET.

3.3.3 Path-sensitive analysis

Static-analysis algorithms that capture full path sensitivity (e.g., [20]) are rather slow since they track all execution states. Fortunately, we are interested only in exception states and transitions between exception states that are described by the FSM. JET adopts ESP, proposed by Das *et al.* [4], a framework that is well-suited for capturing partial path sensitivity.

Given a safety property specified by an FSM, ESP symbolically evaluates the program being analyzed, tracks and updates *symbolic states*. A symbolic state consists of a *property state* and an *execution state*. A property state is a state in the FSM. An execution state models the rest of the program's state and can be configured with different precision. The framework provides a conservative, scalable, and precise analysis to verify program safety properties. Readers may refer to the ESP paper for a more detailed description. In the end, ESP infers information of the following format *at each control-flow edge*:

$$\{ \langle ps_1, es_1 \rangle, \ldots, \langle ps_n, es_n \rangle \}$$

It contains n symbolic states and each symbolic state has its own property state (ps) and execution state (es). Intuitively, it means that there are n possible cases when the program's control is at the control-flow edge. In the i-th case, the property state is ps_i and the execution state is es_i.

In the context of exception analysis, JET uses the FSM specified in Fig. 4. That is, a property state is an exception state. For the execution state, JET tracks two kinds of information: (1) constant values of variables of simple types; and

$$\begin{array}{lll} t & ::= & \texttt{jobj}^n(s_c) \mid \texttt{jcls}^n(s_c) \mid \texttt{jthw}^n(s_c) \\ & \mid & \texttt{jfid}^n(s_c, s_f, s_t) \mid \texttt{jmid}^n(s_c, s_m, s_t) \\ \\ s & ::= & \text{“}Str\text{”} \mid \top \\ n & ::= & 0 \mid 1 \mid * \end{array}$$

Figure 5: Syntax of Java types that JET tracks.

(2) Java-side information of variables that hold Java references. We next explain how these two kinds of information are tracked.

Tracking constant values. For variables of simple types, JET tracks their constant values using an interprocedural and conditional constant propagation [26]. In particular, it tracks integer constants and string constants. String constants are tracked since JNI programs often use them for finding a class, finding a method ID, and other operations.

Take the program in Fig. 3(a) as an example. After `GetIntArrayElements`, there are two symbolic states encoded as follows.

$$\{\, \langle \texttt{NoExn}, \{\, p = \top \,\}\rangle, \langle \texttt{UnChkedExn}, \{\, p = 0 \,\}\rangle \,\}$$

There are two cases. In the first case, there are no exception pending and p is not a constant. In the second case, there is an unchecked exception pending, and $p = 0$. This information correlates the exception state with the value of p and enables the analysis to take advantage of the following if-branch to infer that before the loop it must be the case that no exception is pending.

Following the standard constant propagation, we use $\perp$ for uninitialized variables and $\top$ for non-constants.

Tracking Java types of C variables. JNI programs hold references to Java-side objects and use references to manipulate the Java state. These references are of special types in the native side. For example, a reference of type `jobject` holds a reference to a Java object; a reference of type `jmethodID` holds a reference to a Java method ID.

To track accurately exception states, it is necessary to infer more Java-side information about these references. For instance, since the JNI function `Throw` takes a `jobject` reference, JET has to know the class of the Java object to infer the exact exception class it throws. JET uses a simple type system to capture this kind of information. The type system is presented in Fig. 5. The following table summarizes the meaning of each kind of types.

$\texttt{jobj}^n(s_c)$	A reference to a Java object whose class is s_c
$\texttt{jcls}^n(s_c)$	A reference to a Java class object with the class being s_c
$\texttt{jthw}^n(s_c)$	A reference to a Java `Throwable` object whose class is s_c
$\texttt{jmid}^n(s_c, s_m, s_t)$	A reference to a Java method ID in class s_c with name s_m and type s_t
$\texttt{jfid}^n(s_c, s_f, s_t)$	A reference to a Java field ID in class s_c with name s_f and type s_t

Each s represents either a constant string, or a unknown value (represented by $\top$). When n is 0, it means the reference is a null value; when n is 1, it is non-null; when n is $*$, it can be either null or non-null.

As an example, the following syntax denotes a non-null Java method ID in class “Demo” with name “callback” and type “()V”. The type means that the function takes zero arguments and returns the `void` type.

$$\texttt{jmid}^1(\text{“Demo”}, \text{“callback”}, \text{“()V”})$$

Fig. 6 presents a more elaborate example demonstrating how path sensitivity works in JET. The JNI program invokes a call-back method in class `Demo`. We assume that the call-back method throws a checked exception `IOException`, abbreviated as `IOExn` in the figure. Before invoking a Java method, there is a series of preparation steps in the program: (1) find a reference to the class `Demo`; (2) allocate an object; (3) get the method ID of the call-back function. Since each step may throw an exception, proper checking of null return values after each call is inserted in the code.

For each control-flow edge, Fig. 6 contains annotation that specifies what JET's exception analysis infers. Notice that after `FindClass`, there are two symbolic states representing two cases. The following conditional test checks the return value and as a result the number of symbolic states is reduced to one on both branches.

After `CallVoidMethod`, the only exception state is `ChkedExn(IOException)`. JET can infer this accurately because it knows exactly which Java method the code is calling thanks to the information associated with `mid` before the call. Since JET also takes Java class files as input, it uses the method ID to perform a method look-up in class files and extracts its type signature. The type signature tells what exceptions are declared in the method's `throws` clause. That is, our system takes advantage of the fact that the Java compiler enforces the declared exception effects on Java methods.

3.3.4 Context-sensitive analysis

ESP-style path-sensitivity works well on a single function. However, the context of its inter-procedural version is based on the property state alone [4] and is not sufficient for JNI programs. Fig. 7 presents a typical JNI program for demonstrating the problem.

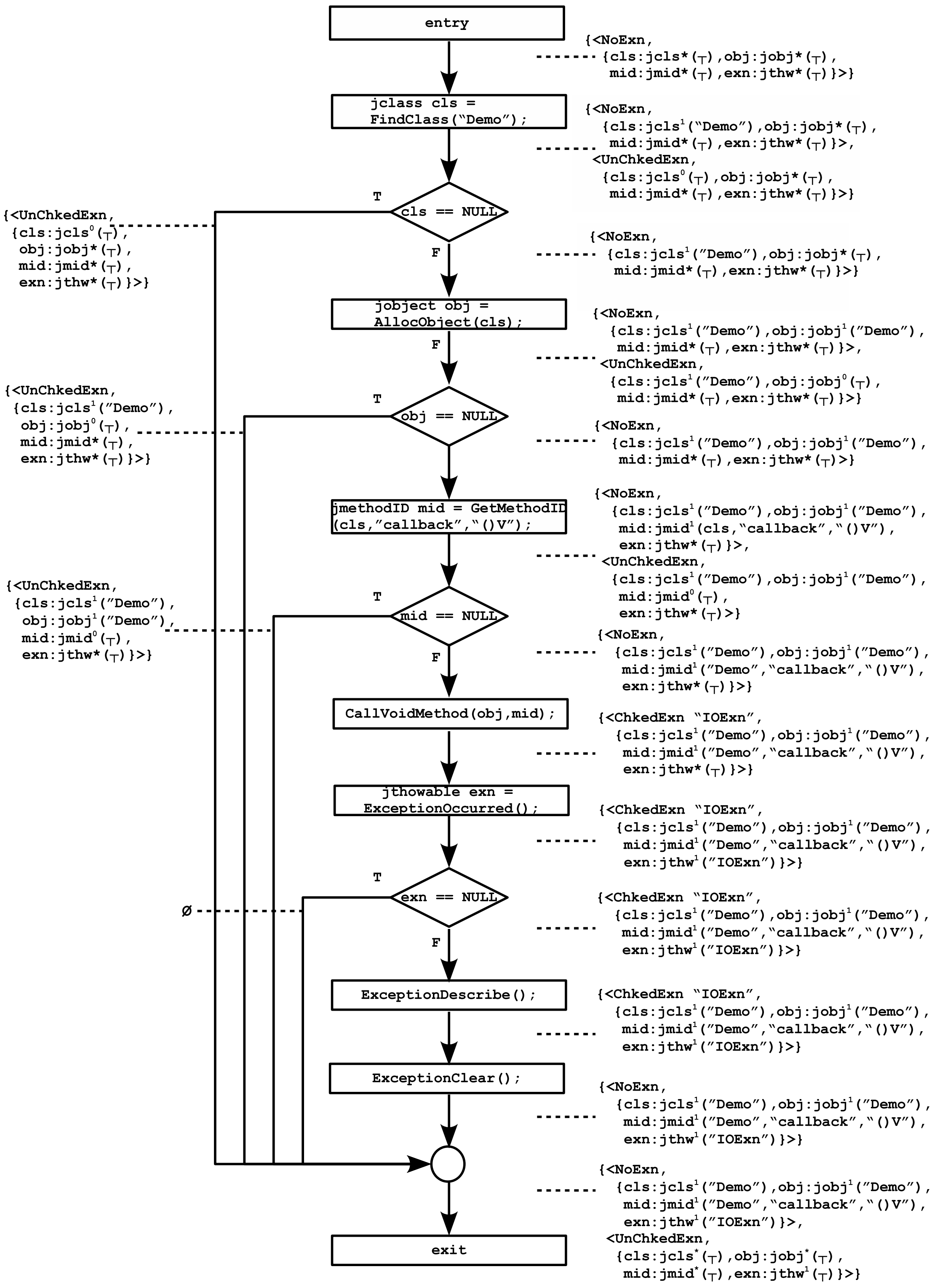
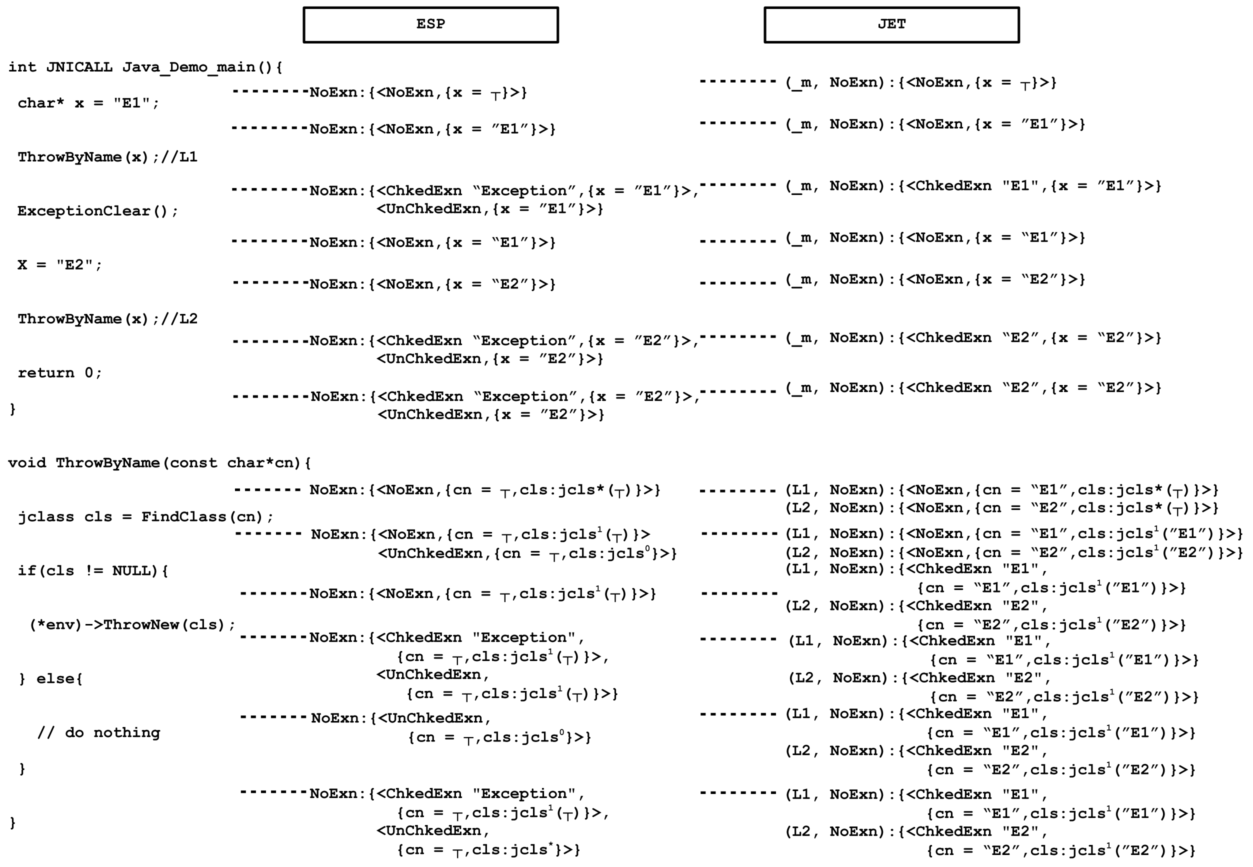

entry
{<NoExn, {cls:jcls*(⊤),obj:jobj*(⊤), mid:jmid*(⊤),exn:jthw*(⊤)}>}
jclass cls = FindClass("Demo");
{<NoExn, {cls:jcls¹("Demo"),obj:jobj*(⊤), mid:jmid*(⊤),exn:jthw*(⊤)}>, <UnChkedExn, {cls:jcls⁰(⊤),obj:jobj*(⊤), mid:jmid*(⊤),exn:jthw*(⊤)}>}
T
cls == NULL
F
{<UnChkedExn, {cls:jcls⁰(⊤), obj:jobj*(⊤), mid:jmid*(⊤), exn:jthw*(⊤)}>}
{<NoExn, {cls:jcls¹("Demo"),obj:jobj*(⊤), mid:jmid*(⊤),exn:jthw*(⊤)}>}
jobject obj = AllocObject(cls);
{<NoExn, {cls:jcls¹("Demo"),obj:jobj¹("Demo"), mid:jmid*(⊤),exn:jthw*(⊤)}>, <UnChkedExn, {cls:jcls¹("Demo"),obj:jobj⁰(⊤), mid:jmid*(⊤),exn:jthw*(⊤)}>}
T
obj == NULL
F
{<UnChkedExn, {cls:jcls¹("Demo"), obj:jobj⁰(⊤), mid:jmid*(⊤), exn:jthw*(⊤)}>}
{<NoExn, {cls:jcls¹("Demo"),obj:jobj¹("Demo"), mid:jmid*(⊤),exn:jthw*(⊤)>}
jmethodID mid = GetMethodID(cls,"callback","()V");
{<NoExn, {cls:jcls¹("Demo"),obj:jobj¹("Demo"), mid:jmid¹(cls,"callback","()V"), exn:jthw*(⊤)}>, <UnChkedExn, {cls:jcls¹("Demo"),obj:jobj¹("Demo"), mid:jmid⁰(⊤), exn:jthw*(⊤)>}
T
mid == NULL
F
{<UnChkedExn, {cls:jcls¹("Demo"), obj:jobj¹("Demo"), mid:jmid⁰(⊤), exn:jthw*(⊤)}>}
{<NoExn, {cls:jcls¹("Demo"),obj:jobj¹("Demo"), mid:jmid¹("Demo","callback","()V"), exn:jthw*(⊤)}>}
CallVoidMethod(obj,mid);
{<ChkedExn "IOExn", {cls:jcls¹("Demo"),obj:jobj¹("Demo"), mid:jmid¹("Demo","callback","()V"), exn:jthw*(⊤)}>}
jthowable exn = ExceptionOccurred();
{<ChkedExn "IOExn", {cls:jcls¹("Demo"),obj:jobj¹("Demo"), mid:jmid¹("Demo","callback","()V"), exn:jthw¹("IOExn")}>}
T
exn == NULL
F
∅
{<ChkedExn "IOExn", {cls:jcls¹("Demo"),obj:jobj¹("Demo"), mid:jmid¹("Demo","callback","()V"), exn:jthw¹("IOExn")}>}
ExceptionDescribe();
{<ChkedExn "IOExn", {cls:jcls¹("Demo"),obj:jobj¹("Demo"), mid:jmid¹("Demo","callback","()V"), exn:jthw¹("IOExn")}>}
ExceptionClear();
{<NoExn, {cls:jcls¹("Demo"),obj:jobj¹("Demo"), mid:jmid¹("Demo","callback","()V"), exn:jthw¹("IOExn")}>}
{<NoExn, {cls:jcls¹("Demo"),obj:jobj¹("Demo"), mid:jmid¹("Demo","callback","()V"), exn:jthw¹("IOExn")}>, <UnChkedExn, {cls:jcls*(⊤),obj:jobj*(⊤), mid:jmid*(⊤),exn:jthw¹(⊤)}>}
exit

Figure 6: Example of JET path sensitivity.

In the example, function `ThrowByName` is used by the `Java_demo_main` function to throw exceptions. There are two places, denoted by `L1` and `L2`, that invoke `ThrowByName`. At `L1`, exception `E1` is thrown and at `L2`, exception `E2`. We assume both `E1` and `E2` are checked exceptions.

ESP merges the execution states of the two call sites when analyzing `ThrowByName` since their property states are the



```
                                 ESP                                               JET
int JNICALL Java_Demo_main(){
                        --------NoExn:{<NoExn,{x = ⊤}>}                  -------- (_m, NoExn):{<NoExn,{x = ⊤}>}
 char* x = "E1";
                        --------NoExn:{<NoExn,{x = ”E1”}>}               -------- (_m, NoExn):{<NoExn,{x = ”E1”}>}
 ThrowByName(x);//L1
                        --------NoExn:{<ChkedExn “Exception”,{x = ”E1”}>,-------- (_m, NoExn):{<ChkedExn "E1",{x = ”E1”}>}
 ExceptionClear();                     <UnChkedExn,{x = ”E1”}>}
                        --------NoExn:{<NoExn,{x = “E1”}>}               -------- (_m, NoExn):{<NoExn,{x = “E1”}>}
 X = "E2";
                        --------NoExn:{<NoExn,{x = “E2”}>}               -------- (_m, NoExn):{<NoExn,{x = “E2”}>}

 ThrowByName(x);//L2
                        --------NoExn:{<ChkedExn “Exception”,{x = ”E2”}>,-------- (_m, NoExn):{<ChkedExn “E2”,{x = “E2”}>}
                                       <UnChkedExn,{x = ”E2”}>}
 return 0;
                        --------NoExn:{<ChkedExn “Exception”,{x = ”E2”}>,-------- (_m, NoExn):{<ChkedExn “E2”,{x = “E2”}>}
}                                      <UnChkedExn,{x = ”E2”}>}


void ThrowByName(const char*cn){
                        ------- NoExn:{<NoExn,{cn = ⊤,cls:jcls*(⊤)}>}    -------- (L1, NoExn):{<NoExn,{cn = “E1”,cls:jcls*(⊤)}>}
                                                                                  (L2, NoExn):{<NoExn,{cn = “E2”,cls:jcls*(⊤)}>}
 jclass cls = FindClass(cn);
                        ------- NoExn:{<NoExn,{cn = ⊤,cls:jcls^1(⊤)}>    -------- (L1, NoExn):{<NoExn,{cn = “E1”,cls:jcls^1(”E1”)}>}
                                       <UnChkedExn,{cn = ⊤,cls:jcls^0}>}          (L2, NoExn):{<NoExn,{cn = “E2”,cls:jcls^1(”E2”)}>}
 if(cls != NULL){                                                                 (L1, NoExn):{<ChkedExn "E1",
                        -------NoExn:{<NoExn,{cn = ⊤,cls:jcls^1(⊤)}>}    --------             {cn = “E1”,cls:jcls^1(”E1”)}>}
                                                                                  (L2, NoExn):{<ChkedExn "E2",
  (*env)->ThrowNew(cls);                                                                      {cn = “E2”,cls:jcls^1(”E2”)}>}
                        -------NoExn:{<ChkedExn "Exception",             -------- (L1, NoExn):{<ChkedExn "E1",
                                         {cn = ⊤,cls:jcls^1(⊤)}>,                             {cn = “E1”,cls:jcls^1(”E1”)}>}
 } else{                               <UnChkedExn,                               (L2, NoExn):{<ChkedExn "E2",
                                         {cn = ⊤,cls:jcls^1(⊤)}>}                             {cn = “E2”,cls:jcls^1(”E2”)}>}
                        ------- NoExn:{<UnChkedExn,                      -------- (L1, NoExn):{<ChkedExn "E1",
   // do nothing                          {cn = ⊤,cls:jcls^0}>}                               {cn = “E1”,cls:jcls^1(”E1”)}>}
                                                                                  (L2, NoExn):{<ChkedExn "E2",
 }                                                                                            {cn = “E2”,cls:jcls^1(”E2”)}>}
                        -------NoExn:{<ChkedExn "Exception",             -------- (L1, NoExn):{<ChkedExn "E1",
                                         {cn = ⊤,cls:jcls^1(⊤)}>,                             {cn = “E1”,cls:jcls^1(”E1”)}>}
}                                      <UnChkedExn,                               (L2, NoExn):{<ChkedExn "E2",
                                         {cn = ⊤,cls:jcls^*}>}                                {cn = “E2”,cls:jcls^1(”E2”)}>}
```

Figure 7: An example illustrating the difference between ESP and JET.

same; both are in the NoExn state. As a result, the value of parameter cn of ThrowByName equals $\top$ and the analysis determines that ThrowByName has the ChkedExn(Exception) effect. But in reality, E1 can only be thrown at L1, and E2 at L2.

JET improves the context by using the call-string approach (of length one) [22]. The context now becomes (n_c, ps), where n_c is the caller node and ps the property state. As a result, the format of information at a control-flow edge becomes

$$(n_c, ps) : \{ \langle ps_1, es_1 \rangle, \ldots, \langle ps_n, es_n \rangle \}$$

It represents the set of symbolic states when the context is (n_c, ps).

With the added contextual information, JET is able to infer the exception effects accurately for the program in Fig. 7. In the figure, we use "_m" to represent a special (unknown) context that invokes the Java_Demo_main function.

The overall interprocedural algorithm is given in the appendix, using notations similar to those used in the ESP paper. The complexity of this algorithm is $Calls \times |D|^2 \times E \times V^2$, where $Calls$ is the number of call sites in the program, $|D|$ is the number of exception states, E is the number of control-flow edges in the control-flow graph, and V is the number of variables in the program. The time complexity is more than that of ESP, but still remains polynomial.

3.3.5 Merging symbolic states

At a merge point of a control flow graph, JET groups symbolic states in the following fashion. First, JET merges symbolic states with respect to identical execution states. This is similar to the grouping function in ESP. Second, JET merges symbolic states that have identical execution states and merge their respective exception states. As property states are just Java exceptions, JET merges exception types according to the Java exception class hierarchy. The second step of merging is possible because of the Java exception-class hierarchy. Grouping exception classes and only using a parent class of the exceptions is sound and conservative. This may introduce imprecision. The advantage is that by having less symbolic states, the analysis runs faster.

3.3.6 Transfer functions for JNI functons

Invoking a JNI function updates the symbolic state. The effects on the symbolic state are defined by a set of transfer functions. These transfer functions are defined according to their specification in the JNI manual [16].

4. Prototype Implementation and Evaluation

The core functionality of JET is implemented in OCaml, with about 3,000 lines of source code. It utilizes CIL [18] to transform native C code into CIL's intermediate representation for exception analysis. It uses the `merge` option provided by CIL so that all native C code in a JNI software package is merged into a single file and analyzed together. JET cannot analyze C++ code because CIL cannot parse it. JET also utilizes JavaLib [10] for looking up and accessing information in Java class files. It allows JET to find Java method signatures and to retrieve information in the Java class hierarchy including those related to exception classes, JET can be configured to search in any set of Java class files. In our experiments, we configured JET to work with the entire set of classes in the JDK as well as non-JDK benchmark programs.

4.1 Experimental evaluation

To evaluate JET, we conducted experiments to evaluate its accuracy and efficiency. All experiments were carried out on a Linux Ubuntu 9.10 box with Intel Core2 Duo CPU at 2.53GHz and with 512MB memory.

The experiments were designed to answer the following set of questions:

(1) How effective is JET at uncovering true bugs? Does it generate too many false positives along the process?

(2) How efficient is JET in terms of analysis time? Does it scale to large programs?

(3) Is the two-stage exception analysis necessary?

We compiled a set of JNI packages for experimentation. The packages and their numbers of lines of source code are listed in Table 1. The packages with names starting with `java` are extracted from Sun's JDK6U03 (release 6 update 3). These packages cover all native code in Sun's JDK under the `share` and `solaris` directories. Other packages include the following programs: (1) `libec` is a library for elliptic curve cryptography; OpenJDK provides JNI bindings for interfacing with the library; (2) `hpdf` is an open-source PDF creation library; as it does not ship with JNI bindings, we created our own; (3) `libreadline` is a package that provides JNI wrappers for accessing the GNU readline library; (4) `posix 1.0` is a package that provides JNI wrappers for accessing OS system calls; (5) `spread` is an open-source toolkit that provides a high performance messaging service across local and wide area networks; it comes with its default JNI bindings.

Evaluating JET's effectiveness and accuracy. For the set of benchmark programs, Table 1 presents the number of warnings and the number of true bugs. Of these programs, JET reported a total of 18 warnings, among which 12 are true bugs after manual inspection. The overall false positive rate is 33%.

```
...
data->name = "java.io.IOException";
...
if(data->name != NULL){
  ThrowByName(data->name);
}
...
```

Figure 8: A typical example of false positives.

There are conceptually two categories of bugs. The first category occurs when a native method does not declare any exceptions but its implementation can actually throw some. The second category occurs when a native method declares some exceptions but the implementation can actually throw a checked exception that is not a subclass of any of the declared exceptions. All the bugs we identified in the benchmark programs belong to the first category. For example, in the `libec` package, the implementation of native method `generateECKeyPair` throws `java.security.KeyException`. However, the method's Java signature does not declare any exception. A same type of bug is found in the `java.nio` package of Sun's JDK, where native method `force0` throws `java.io.IOException` but the method's Java-side signature does not declare any exception. We manually checked against JDK6U23, a new version of the JDK, for the bugs identified in JDK6U03 and found that one bug in `ZipFile` has been fixed in JDK6U23. But other bugs in the JDK remain.

There are six false positives. All false positives are caused by the imprecision when tracking the execution state. Fig. 8 presents a typical example. A string constant that represents the name of a Java exception class is stored in a C struct and used later for throwing an exception. Since JET tracks only constant values of C variables of simple types, when it encounters the `ThrowByName`, it cannot tell the exact exception class. Consequently, it will report the method that contains the code can possibly throw `java.lang.Exception`, the top checked exception class. However, the native method's Java-side signature declares a more precise exception class, `java.io.IOException`. We could further improve JET's static analysis to reduce these false positives. But since the total number of false positives is quite small, we did not feel it is necessary.

Although JET is designed to be conservative, it is possible for JET to have false negatives due to errors in its implementation. We conducted manual audit in half of the packages of our benchmark programs, obtained the ground truth, and

[4] `java.lang.other` includes all native code in `java.lang` but not in `java.lang.strictmath`.

JNI Package	LOC	Warnings	True Bugs
java.io	2,057	3	1
java.lang.strictmath	10,819	0	0
java.lang.other[4]	3,153	1	1
java.net	9,748	4	0
java.nio	385	1	1
java.security	78	0	0
java.sql	19	0	0
java.util.concurrent.atomic	16	0	0
java.util.timezone	700	0	0
java.util.zip	14,214	2	2
libec	19,465	1	1
hpdf	135,039	0	0
libreadline 0.8.0	1,847	2	2
posix 1.0	2,178	1	1
spread	27,803	3	3
TOTAL	227,521	18	12

Table 1: Evaluation of accuracy of JET.

compared it with the results reported by JET. This served as a way to debug JET and to gain confidence.

Analysis time. Table 2 presents the analysis time of JET on the benchmark programs. As we can see, JET's coarse-grained exception analysis (first stage) is very efficient, taking only 400 μs for all packages. The second stage dominates the total time taken by JET. In all, it takes about 8 seconds for all packages.

Effectiveness of the two-stage system. Table 2 also presents metrics that are used to evaluate the effectiveness of the two-stage system. The third column (LOC retained/reduced) shows the numbers of lines of source code retained and reduced by the coarse-grained analysis. The numbers show that it is very effective in terms of separating the code that affects Java's state. It reduces the amount of code necessary to be analyzed by almost 99%. Only a small portion of code is left for the fine-grained analysis.

The column "interface/library LOC (manual separation)" shows the number of interface code and library code determined by manual separation. This is a quick, rough classification using files as the unit. That is, if a file contains some JNI function calls, we put it under the category of interface code. When compared to the column of "LOC retained/reduced" by the first stage, we see our automatic algorithm is better at separating interface code and library code; the main reason is that it uses functions as the classification unit.

Finally, the column "Time without 1st stage" presents the analysis time if the whole package is fed directly into the second stage without going through the first stage. This resulted in a dramatic slow down in runtime.

5. Related Work

Almost all widely used programming languages support a foreign function interface (FFI) for interoperating with program modules developed in low-level code (e.g., [1, 2, 6, 14, 16, 19]). Most early work on FFIs aimed to provide efficient implementations.

In recent years, researchers studied how to improve upon FFIs' safety, reliability, and security. FFI-based code is often a rich source of software errors; a few recent studies reported hundreds of interface bugs in JNI programs ([8, 11, 15]). Errors occur often in interface code because FFIs generally provide little or no support for safety checking, and also because writing interface code requires resolving differences (e.g., memory models and language features) between two languages. Past work on improving FFIs' safety can be roughly classified into several categories: (1) Static analysis has been used to identify specific classes of errors in FFI code [7, 8, 11, 15, 24]; (2) In another approach, dynamic checks are inserted at the language boundary and/or in the native code for catching interface errors (see [13]) or for isolating errors in native code so that they do not affect the host language's safety [25] and security [23]; (3) New interface languages are designed to help programmers write safer interface code (e.g., [9]). JET takes the static-analysis approach, following how checked exceptions are enforced in Java's type system.

We next compare in more detail with three closely related work on using static analysis to find bugs in JNI programs [8, 11, 15]. Our previous system [15] and a system by Kondoh and Onodera [11] identify situations of mishandling JNI exceptions in native code. Both systems compute at each program location whether there is a possible JNI

JNI Package	1st stage time (μs)	LOC retained/reduced	2nd stage time (s)	Total time (s)	interface/library LOC (manual separation)	Time without 1st stage (s)
java.io	30	768/1,289	2.56	2.56	2,057/0	22.4
java.lang.strictmath	30	0/10,819	0	30μ	2,314/8,505	1.11
java.lang.other	30	536/2,617	0.08	0.08	3,153/0	30.91
java.net	100	527/8,968	3.13	3.13	9,748/0	94.69
java.nio	10	43/281	0.24	0.24	385/0	0.20
java.security	0	0/78	0	0	78/0	0.05
java.sql	0	0/19	0	0	19/0	0.00166
java.util.concurrent.atomic	0	0/16	0	0	16/0	0.08
java.util.timezone	0	0/700	0	0	700/0	1.90
java.util.zip	20	364/13,850	0.87	0.87	797/13417	63.73
libec	20	126/19,339	0.61	0.61	416/19,049	55.45
hpdf	60	165/134,874	0.45	0.45	327/134,712	187.13
libreadline0.8.0	20	148/1,699	0.02	0.02	654/1,193	12.47
posix1.0	20	261/1,917	0.05	0.05	2,178/0	8.50
spread	60	311/27,492	0.34	0.34	2,395/25,408	9.43
TOTAL	400	3,249/223,958	8.34	8.34	25,237/202,284	488.02

Table 2: Evaluation of efficiency of JET.

exception pending. However, they do not compute specific classes of pending exceptions. To do that, JET has to use a much more complicated static analysis. The analysis tracks information of C variables in native code and correlates them with the exception state; it also takes Java-method signatures into account for tracking exceptions that may be pending when invoking a Java method. Both are necessary for computing exception effects. J-Saffire [8] identifies type misuses in JNI-based code but does not compute exception effects for native methods. J-Saffire also finds it necessary to track information of C variables that hold Java references. J-Saffire performs polymorphic type inference that is based on semi-unification, while JET uses interprocedural dataflow analysis. There are cases where J-Saffire's type inference can infer more precise information. However, we did not find this is a problem in practice.

Many systems perform additional exception analysis, stronger than the built-in static exception checking of languages such as Java. For example, the Jex tool [21] and others (e.g., [3, 17]) can compute what exceptions might be pending at each program point. This information is essential for understanding where exceptions are thrown and caught. JET computes this information for JNI programs.

6. Conclusions

Exceptions are commonly used in Foreign-Function Interfaces (FFIs) as a way for native code to report error conditions to the host language. Java adopts the notion of checked exceptions, which are useful for statically identifying unhandled exceptional situations. JET is a system that extends Java's rules for checked exceptions to native code. It is both scalable and has high precision thanks to its careful engineering trade-offs. Evaluation demonstrates JET is effective at identifying exception-related bugs in native code. We are turning JET into an Eclipse plug-in tool, which will be useful for Java software developers.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments.

References

[1] M. Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[2] E. M. Chakravarty. The Haskell 98 foreign function interface 1.0: An addendum to the Haskell 98 report. `http://www.cse.unsw.edu.au/~chak/haskell/ffi/`, 2005.

[3] B.-M. Chang, J.-W. Jo, K. Yi, and K.-M. Choe. Interprocedural exception analysis for Java. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 620–625, New York, NY, USA, 2001. ACM.

[4] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 57–68, 2002.

[5] C. Elford. Integrated debugger for Java/JNI environments. `http://software.intel.com/en-us/articles/integrated-debugger-for-javajni-environments/`, Oct. 2010.

[6] K. Fisher, R. Pucella, and J. H. Reppy. A framework for interoperability. *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[7] M. Furr and J. S. Foster. Checking type safety of foreign function calls. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 62–72, 2005.

[8] M. Furr and J. S. Foster. Polymorphic type inference for the JNI. In *15th European Symposium on Programming (ESOP)*, pages 309–324, 2006.

[9] M. Hirzel and R. Grimm. Jeannie: Granting Java Native Interface developers their wishes. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 19–38, 2007.

[10] JavaLib. JavaLib. http://javalib.gforge.inria.fr/.

[11] G. Kondoh and T. Onodera. Finding bugs in Java Native Interface programs. In *ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 109–118, New York, NY, USA, 2008. ACM.

[12] B. Lee, M. Hirzel, R. Grimm, and K. McKinley. Debug all your code: A portable mixed-environment debugger for Java and C. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 207–226, 2009.

[13] B. Lee, M. Hirzel, R. Grimm, B. Wiedermann, and K. S. McKinley. Jinn: Synthesizing a dynamic bug detector for foreign language interfaces. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2010. To appear.

[14] X. Leroy. *The Objective Caml system*, 2008. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`.

[15] 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 (CCS)*, pages 442–452, 2009.

[16] S. Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0201325772.

[17] D. Malayeri and J. Aldrich. Practical exception specifications. In *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 200–220. Springer, 2006.

[18] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *International Conference on Compiler Construction (CC)*, pages 213–228, 2002.

[19] Python/C FFI. Python/C API reference manual. `http://docs.python.org/c-api/index.html`, Apr. 2009.

[20] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *22nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 49–61, 1995.

[21] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Programming Languages and Systems*, 12(2):191–221, 2003. ISSN 1049-331X.

[22] M. Sharir and A. Pnueli. Two approaches to inter-procedural dataflow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall Inc., 1981.

[23] J. Siefers, G. Tan, and G. Morrisett. Robusta: Taming the native beast of the JVM. In *Proceedings of the 17th ACM conference on Computer and communications security (CCS)*, pages 201–211, 2010.

[24] G. Tan and G. Morrisett. ILEA: Inter-language analysis across Java and C. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 39–56, 2007.

[25] 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*, pages 97–106, 2006.

[26] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

A. Inter-procedural exception analysis

We first describe some notations. A symbolic state $S = D \times X$, where a property state $D = \{\, \texttt{NoExn}, \texttt{ChkedExn}(E_1), \ldots, \texttt{ChkedExn}(E_n), \texttt{UnChkedExn}, \texttt{Error} \,\}$, and an execution state X is a map from variables to values in a constant-propagation lattice or Java types. Given a symbolic state s, $ps(s)$ is its property state and $es(s)$ is its execution state.

A global control-flow graph = $[N, E, F]$, where N is a set of nodes, E is the set of edges, and F is the set of functions. Notation $src(e)$ denotes edge e's source node and $dst(e)$ the destination node. For node n, notation $In_0(n)$ stands for its first incoming edge and $In_1(n)$ the second (if there is one). A merger node is assumed to have two incoming edges. For a non-branch node, $Out_T(n)$ stands for its only outgoing edge. For a branch node, $Out_T(n)$ stands for the true branch and $Out_F(n)$ the false branch. We assume each function has a distinguished entry node, denoted by $entryNode(f)$. Notation $fn(n)$ denotes the function that node n belongs to. When n stands for a function-call node, $callee(n)$ denotes the callee function.

As in ESP, α denotes a function that merges a set of symbolic states according to the property state:

$$\alpha(ss) = \{\, [d, \bigsqcup_{s \in ss[d]} es(s)] \mid d \in D \wedge ss[d] \neq \emptyset \,\}$$

where $ss[d] = \{\, s \mid s \in ss \wedge d = ps(s) \,\}$.

We use F to denote transfer functions for nodes. For instance, F_{Merge} is the transfer function for merger nodes.

- $F_{Merge}(n, ss_1, ss_2) = \alpha(ss_1 \bigcup ss_2)$.
- $F_{Call}(f, ss, \bar{a}) = \alpha(\{s' | s' = f_{Call}(f, s, \bar{a}) \wedge s \in ss\})$ where f_{Call} binds parameters of f to the symbolic-evaluation results of arguments $\bar{a}$ in symbolic state s;

it also take cares of scoping by removing bindings for variables not in the scope of f.

- $F_{Branch}(n, ss, v) = \alpha(\{ s' | s' = f_{Branch}(n, s, v) \wedge s \in ss \wedge es(s') \neq \bot \})$, where f_{Branch} takes advantage of the fact that the result of the branching condition is v and adjusts the symbolic state s.
- $F_{Exit}(f, ss) = \alpha(\{ s' \mid s' = f_{Exit}(f, s) \wedge s \in ss \})$, where f_{Exit} takes care of scoping by removing variables that are only in the scope of f from the symbolic state.
- We use F_{JNI} to denote the transfer functions for JNI functions and we use F_{Other} to denote the transfer function for all other nodes.

Algorithm 1: Auxiliary procedures

```
procedure $Add(e, n_c, d, ss)$
begin
    if $Info(e, n_c, d) \neq ss$ then
        $Info(e, n_c, d) := ss$;
        $Worklist := Worklist \bigcup \{[dst(e), n_c, d]\}$;
procedure $AddTrigger(n, n_c, d, ss, \bar{a})$
begin
    $ss' := F_{Call}(fn(n), ss, \bar{a})$;
    $e := Out_T(n)$;
    $ss' := \alpha(ss' \bigcup Info(e, n_c, d))$;
    $Add(e, n_c, d, ss')$;
procedure $AddToSummary(n, n_c, d, ss)$
begin
    $ss' = F_{Exit}(fn(n), ss)$;
    if $Summary(fn(n), n_c, d) \neq ss'$ then
        $Summary(fn(n), n_c, d) := ss'$;
        foreach
        $n'_c, d' \in D$ such that $Info(In_0(n_c), n'_c, d') \neq \emptyset$
        do
            $Worklist := Worklist \bigcup \{[n_c, n'_c, d']\}$;
```

Algorithm 2: Inter-procedural exception analysis

```
Input
    Global control-flow graph=$[N, E, F]$;
    $f_{entry} \in F$ is an entry function to be analyzed
Globals
    $Worklist : 2^{N \times N \times D}$;
    $Info : (E \times N \times D) \rightarrow 2^S$;
    $Summary : (F \times N \times D) \rightarrow 2^S$;
procedure $solve$
begin
    $\forall\, e, n, d, Info(e, n, d) = \emptyset$;
    $\forall\, f, n, d, Summary(f, n, d) = \emptyset$;
    $e := Out_T(entryNode(f_{entry}))$;
    $Info(e, \_m, \texttt{NoExn}) := \{[\texttt{NoExn}, \top]\}$;
    $Worklist := \{[dst(e), \_m, \texttt{NoExn}]\}$;
    while $Worklist \neq \emptyset$ do
        Remove $[n, n_c, d]$ from $Worklist$;
        $ss_{in} := Info(In_0(n), n_c, d)$;
        switch $n$ do
            case $n \in Merge$
                $ss_{out} :=$
                $F_{Merge}(n, ss_{in}, Info(In_1(n), n_c, d))$;
            case $n \in Branch$
                $Add(Out_T(n), n_c, d, F_{Branch}(n, ss_{in}, T))$;
                $Add(Out_F(n), n_c, d, F_{Branch}(n, ss_{in}, F))$;
            case $n \in JNIFun$
                $Add(Out_T(n), n_c, d, F_{JNI}(n, ss_{in}, n_c, d))$;
            case $n \in Call(\bar{a})$
                $ss_{out} := \emptyset$;
                foreach $d' \in D$ such that $ss_{in}[d'] \neq \emptyset$
                do
                    $sm :=$
                    $Summary(callee(n), n, d')$;
                    if $sm \neq \emptyset$ then
                        $ss_{out} := ss_{out} \bigcup sm$;
                    $AddTrigger(entryNode(callee(n)),$
                                $n, d', ss_{in}[d'], \bar{a})$;
                $Add(Out_T(n), n_c, d, \alpha(ss_{out}))$;
            case $n \in Exit$
                $AddToSummary(n, n_c, d, ss_{in})$;
            case $n \in Other$
                $ss_{out} := F_{Other}(n, ss_{in}, n_c, d)$;
                $Add(Out_T(n), n_c, d, ss_{out})$;
    return $Info$
```