Exception Analysis in the Java Native Interface

Siliang Li, Gang Tan

P.C. Rossin College of Engineering & Applied Science
Computer Science and Engineering
Packard Laboratory, 19 Memorial Drive West
Lehigh University, Bethlehem PA 18015

Abstract

A Foreign Function Interface (FFI) allows one host programming language to interoperate with another foreign language. It enables efficient software development by permitting developers to assemble components in different languages. One typical FFI is the Java Native Interface (JNI), through which Java programs can invoke native-code components developed in C, C++, or assembly code. Although FFIs bring convenience to software development, interface code developed in FFIs is often error prone because of the lack of safety and security enforcement. This paper introduces a static-analysis framework, TurboJet, which finds exception-related bugs in JNI applications. It finds bugs of inconsistent exception declarations and bugs of mishandling JNI exceptions. TurboJet is carefully engineered to achieve both high efficiency and accuracy. We have applied TurboJet on a set of benchmark programs and identified many errors. We have also implemented a practical Eclipse plug-in based on TurboJet that can be used by JNI programmers to find errors in their code.

Keywords:
Foreign Function Interface, Java Native Interface, Exception Checking, Static Analysis

1. Introduction

Today's software development rarely uses just one language or framework but often uses a mix of several languages: programmers design and implement some functionality in one language, and complete the building of the software by snapping together legacy systems or library code written in other languages. This is considered a more efficient practice because it allows programmers to focus on developing new features without "re-inventing the wheel". More importantly, the practice makes it possible to deliver a software product quickly to the competitive market place.

A Foreign Function Interface (FFI) is a mechanism that permits software written in one *host* programming language to interoperate with another *foreign* language in the form of invoking functions across language boundaries. In this paper, we focus on the Java Native Interface (JNI), which allows Java programs to interface with low-level C/C++/assembly code (i.e., native code). 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`. The actual implementation of the `open` method is implemented in C. Once declared, native methods are invoked in Java in the same way as how Java methods are invoked. We define a *JNI application* as a set of Java class files together with the native code that implements the native methods declared in the class files.

While the use of FFIs can bring convenience and efficiency in software development, it also has its downsides. Beyond the obvious differences in code syntax and semantics, different programming languages can have different type systems, exception-handling mechanisms, memory-management schemes, multithreading programming models, and so on. Because of these differences, programming with FFIs requires extreme care, making it an error-prone process. Misuse of FFIs can introduce issues that are difficult to debug, and make the software less safe and less reliable.

Specifically in this paper, we study the differences of exception-handling mechanisms between Java and native code. Java has two features related to exceptions that help improve program reliability.

- *Compile-time exception checking.* 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). While the usefulness of checked exceptions for large programs is not universally agreed upon by language designers, proper use of checked exceptions improve program robustness by enabling the compiler to identify unhandled exceptional situations during compile time.

- *Runtime exception handling.* When an exception is pending in Java code, the Java Virtual Machine (JVM) automatically transfers the control to the nearest enclosing try-catch block that matches the exception type.

Native methods can also throw, handle, and clear Java exceptions through a set of interface functions provided by the JNI. Consequently, an exception may be pending when the control

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(env);
  ...
}
```

Figure 1: A simple JNI example. This example also demonstrates how a native method can violate its exception declaration. `ThrowIOException` is a utility function that throws a Java `IOException`.

is in a native method. For 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. JNI exceptions are treated differently from Java exceptions.

- A Java compiler does not perform compile-time exception checking on native methods, in contrast to how exception checking is performed on Java methods.
- The JVM does not provide runtime exception handling for JNI exceptions. An exception pending on the native side does not immediately disrupt the native-code execution, and only after the native code finishes execution will the JVM mechanism for exceptions start to take over.

Because of these differences, it is easy for JNI programmers to make mistakes. First, since there is a lack of compile-time exception checking on native methods, the exceptions declared in a native-method type signature might differ from those exceptions that can actually happen during runtime. Second, since there is no support for runtime exception handling in native methods, JNI programmers have to write their own code for implementing the correct control flow for handling and clearing exceptions—an error-prone process. Both kinds of mistakes might lead to unexpected crashes in Java programs and even security vulnerabilities. More detailed discussion and concrete examples of such mistakes will be presented in Section 3.

The major contribution of this paper is TurboJet, a complete framework that performs exception analysis on JNI applications. Specifically, TurboJet statically analyzes and examines JNI code for finding inconsistencies between native-method exception declarations and implementations. Furthermore, it statically analyzes native-method implementations to check whether exceptions are handled properly. The benefit of static analysis is that it examines every control-flow path in a program and in general does not miss mistakes. It is especially appropriate for catching defects in exceptional situations because they are hard to trigger with normal input and events. Having said that, we note that the focus of TurboJet is not the creation of new static-analysis algorithms, but rather the building of a scalable system that performs efficient and accurate analysis over large Java packages of mixed Java and native code. In this process, TurboJet adapts several static-analysis algorithms, which are combined to achieve favorable results.

Our experimental results show that TurboJet delivers high accuracy in catching bugs with relatively low false-positive rates and short execution time. For 16 benchmark programs with nearly 100K lines of code, TurboJet finds 147 bugs caused by mishandling inconsistent exceptions and 17 bugs of inconsistent exception declarations, with the false positive rates 23% and 32%, respectively. The total time took to examine all benchmark programs is only 14 seconds.

TurboJet is built on the basis of our two previous systems [1, 2] but improves upon them in several aspects. Foremost, TurboJet provides a unified framework capable for finding both types of bugs due to exceptional situations in the JNI; the framework introduced in this paper is more complete and more general than the previous systems. Details on the construction of the unified framework are discussed in this paper. Second, TurboJet's exception-analysis framework improves the previous exception analysis [1] in terms of precision and efficiency. Sections 7.1 and 7.2 show the improved results due to the refined analysis. Third, to better evaluate TurboJet, we conducted extensive experiments covering more benchmark programs than before and performed additional comparison with recent works. Results are presented in Section 7.3.1. Finally, we put TurboJet into practical use by implementing an Eclipse plug-in tool based on TurboJet. Software developers wish to check their JNI development can find this tool very useful.

The rest of the paper is structured as follows. We present in Section 2 some background information. In Section 3, we discuss in detail the bug patterns studied in this paper, followed by an overview of TurboJet in Section 4. In Section 5, we present details of TurboJet's exception-analysis framework. We then discuss how TurboJet detects bugs of mishandling JNI exceptions in Section 6. Prototype implementation and evaluation are presented in Section 7. We have developed an Eclipse plug-in based on TurboJet; its details are discussed in Section 8. We discuss closely related work in Section 9, possible future work in Section 10, and conclude in Section 11.

2. Background: the JNI

The JNI is Java's foreign function interface that allows Java code to interoperate with native code. Figs. 1 and 2 provide two example JNI programs. A Java class declares a native method using the `native` keyword. The native method can be implemented in C code. Once a native method is declared, Java code can invoke the native method in the same way as how a Java method is invoked.

Interactions between Java and C can be involved and are made possible through the use of JNI functions. In general, Java can pass C code a set of Java object references. In addition, a JNI environment pointer of type "`JNIEnv *`" is passed to C. The environment pointer contains entries for JNI function pointers, through which C code can invoke JNI functions. For instance, the C code in Fig. 2 invokes `GetArrayLength` through the environment pointer to get the length of a Java array. Through these JNI functions, native methods can read, write, and create Java objects, raise exceptions, invoke Java methods, and so on.

2.1. How JNI exceptions can be thrown

Both Java code and native code may throw exceptions. There are several ways that an exception may become pending on the native side:

- Native code can throw exceptions directly through JNI functions such as `Throw` and `ThrowNew`. We call such throws *explicit throws*.
- Many JNI functions throw exceptions to indicate failures. For instance, `NewCharArray` throws an exception when the allocation fails. Such throws are considered as *implicit throws*.
- 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 on the native side. We treat this type of throws also as implicit throws.

Both explicit and implicit throws result in pending exceptions on the native side.

2.2. Checked exceptions vs. unchecked exceptions

In Java, 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 Java exceptions are not statically checked. Following Java's practice, TurboJet issues warnings about inconsistent exception declarations only for checked exceptions. Nevertheless, TurboJet's exception analysis tracks both checked and unchecked exceptions (unlike our previous work [2], which tracks only checked exceptions). The tracking of unchecked exceptions is necessary for finding the second kind of bugs (i.e., mishandling JNI exceptions).

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

2.3. Interface code 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; this fact is taken advantage of by TurboJet for better efficiency, as we will show. 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 IO operations and does not go through a native library.

3. Defining bug patterns

TurboJet looks for two kinds of mistakes in native methods of JNI applications: inconsistent exception declarations and mishandling JNI exceptions. We next give a detailed discussion of these two kinds of mistakes and present concrete examples.

3.1. Inconsistent exception declarations

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

Since a Java compiler does not perform static exception checking on native methods, this type of bugs might lead to unexpected crashes in Java programs. Let us use the program in Fig. 1 as an example. The `ZipFile` class declares a native method named `open`. 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. 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`.

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., [3, 4]) can help, but not all bugs manifest themselves during particular program runs.

3.2. Mishandling JNI exceptions

As we have discussed, JVM provides no runtime support for exception handling when a native method is running. A pending JNI exception does not immediately disrupt the native code execution. Because of the subtle difference between the runtime semantics of JNI exceptions and Java exceptions, it is easy for JNI programmers to make mistakes.

Fig. 2 presents a toy example that shows how mishandling JNI exceptions may lead to security vulnerabilities. In the example, the Java class `Vulnerable` declares a native method, which is realized by a C function. The C code checks for some condition (a bounds check in this case), and when the check fails, a JNI exception is thrown. It appears that the following `strcpy` is safe as it is after the bounds check. However, since the JNI exception does not disrupt the control flow, the `strcpy` will always be executed and may result in an unbounded string copy. Consequently, an attacker can craft malicious input to the public Java `byteCopy()` method, and overtake the JVM.

The error in Fig. 2 can be fixed quite easily—just insert a return statement after the exception-throwing statement. In general, when a JNI exception is pending, native code should do one of the following two things:

- 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.

In short, JNI programmers are required to implement the control flow of exception handling correctly. This is a tedious and error-prone process, especially when function calls are involved. For example, imagine a C function, say `f`, invokes another C function, say g, and the function g throws an exception when an error occurs. The `f` function has to explicitly deal with two cases after calling g: the successful case, and the exceptional case. Mishandling it may result in the same type of errors as the one in the example. An empirical study has identified 35 cases of mishandling JNI exceptions in 38,000 lines of C code [5].

The error pattern of mishandling exceptions is not unique to the JNI. Any programming language with managed environments that allows native components to throw exceptions faces the same issue. Examples include the Python/C API interface [6] and the OCaml/C interface [7].

Furthermore, mishandling Java exceptions in JNI-based C code is a special case of mishandling errors in C. Since the

Java code

```
class Vulnerable {
  //Declare a native method
  private native void bcopy(byte[] arr);
  public void byteCopy(byte[] arr) {
    //Call the native method
    bcopy(arr);}
  static {
    System.loadLibrary("Vulnerable");
  }
}
```

C code

```
//Get a pointer to the Java array,
//then copy the Java array
//to a local buffer
void Java_Vulnerable_bcopy
  (JNIEnv* env, jobject obj, jbyteArray jarr) {
  char buffer[512];
  //Check for the length of the array
  if ((*env)->GetArrayLength(jarr)>512) {
    JNU_ThrowArrayIndexOutOfBoundsException(env,0);
  }
  jbyte *carr =
    (*env)->GetByteArrayElements(jarr,NULL);
  //Dangerous operation
  strcpy(buffer, carr);
  (*env)->ReleaseByteArrayElements(arr,carr,0);
}
```

Figure 2: An example of mishandling JNI exceptions. The function `JNU_ThrowArrayIndexOutOfBoundsException` is a JNI utility function. It first uses `FindClass` to get a reference to class `ArrayIndexOutOfBoundsException` and then uses `ThrowNew` to throw the exception.

C language does not provide language support for exceptions, C functions often use integer return codes to report errors to callers. Callers must use these return values to check for error conditions and perform appropriate actions such as handling errors or propagating errors to their callers. This process is tedious and can be many programming mistakes, as reported by a study on how errors are handled in an embedded software system [8]. Previous systems [9, 8] used static analysis to automatically detect error-handling errors in C code. These systems rely on manual patterns to determine where errors are generated, propagated, and handled. In the JNI context, where Java exceptions are generated, propagated, and handled can be identified automatically based on relevant JNI-function invocations; this makes our problem more tractable.

Defining the pattern of mishandling JNI exceptions. When a JNI exception is pending, native code should either return to the Java side after performing some clean-up tasks, or handle and clear the exception itself. Intuitively, there is a set of "safe" operations that are allowed when an exception is pending. For example, a return-to-Java operation is safe, so are calling JNI functions such as `ExceptionOccurred` or `ExceptionClear`. Given this intuition, we next define the defect pattern of mis-

(a)

```
int len=(*env)->GetArrayLength(arr);
int *p=(*env)->GetIntArrayElements(arr,NULL);
for (i=0; i<len; i++) {
  sum += p[i];
}
```

(b)

```
jcharArray elemArr=(*env)->NewCharArray(len);
(*env)->SetCharArrayRegion(elemArr,0,len,chars);
```

Figure 3: Two more examples of mishandling JNI exceptions.

handling JNI exceptions:

Definition 2. *JNI-based native code has a defect of* mishandling JNI exceptions *if there is a location in the code that can be reached during runtime with a state such that*

1. *a JNI exception is pending,*
2. *and the next operation is an unsafe operation.*

By this definition, the example in Fig. 2 has a defect because it is possible to reach the location before `strcpy` with a pending exception, and `strcpy` is in general unsafe.

We make a few clarifications about the definition. First, a JNI exception can be either a checked or an unchecked exception. Second, it leaves open the notion of "unsafe operations". Our strategy of determining unsafe operations will be described in Section 6.1. Finally, readers may wonder whether it is wise to look for defects of mishandling JNI exceptions in the first place. After all, one could argue that the example in Fig. 2 is a case of buffer overflows and a bug finder targeting buffer overflows would suffice. This is indeed true for that example. However, we would argue that there are many other scenarios that do not involve buffer overflows, but are similar to the example in terms of incorrect control flow following JNI exceptions. We give two examples in Fig. 3.

In Fig. 3(a), `GetIntArrayElements` may throw an exception and return NULL when it fails to get a pointer to an input Java integer array. In this case, the subsequent array access in `p[i]` results in a null-pointer dereference.

In Fig. 3(b), `NewCharArray` may throw an exception when allocation fails. The JNI reference manual states that "it is extremely important to check, handle, and clear a pending exception before calling any subsequent JNI functions. Calling most JNI functions with a pending exception—with an exception that you have not explicitly cleared—may lead to unexpected results." Therefore, it is unsafe to invoke the next JNI function `SetCharArrayRegion` in the example.

These examples demonstrate that mishandling JNI exceptions may result in a variety of runtime failures. Rather than constructing a set of bug finders targeting each of these failures, it is beneficial to have a single framework targeting the general pattern of mishandling JNI exceptions.

4. Overview of TurboJet

Fig. 4 presents a high-level diagram that depicts the system flow of TurboJet. At a high-level, TurboJet takes a JNI application as input and generates warnings for native methods in terms of inconsistent exception declarations and mishandling JNI exceptions. A JNI application is composed of both Java class files and native code that implements the native methods declared in the class files. The application is fed into an exception-analysis component, which performs static analysis to determine possible pending JNI exceptions at each program point. We use the term *exception effects* to refer to the set of exceptions that may escape a method. The exception-effect information is then used by two warning generators that search for bugs of (1) inconsistent exception declarations, and (2) mishandling JNI exceptions.

The exception-analysis component is designed to be both efficient and precise. These goals are achieved by a two-stage analysis: the first stage is a coarse-grained analysis, which quickly prunes a large chunk of irrelevant code; the second stage performs fine-grained analysis that employs both path sensitivity and context sensitivity to perform precise exception analysis on the remaining code. Details of the exception-analysis component is discussed in Section 5.

The warning generator for inconsistent exception declarations is straightforward and directly use the information generated by exception analysis. The exception analysis determines possible pending JNI exceptions at every program point. The possible pending checked exceptions at the end of a native method is considered the actual exception effects of the native method. The actual exception effects are then compared with the declared exception effects extracted from Java class files. If there is a mismatch, the warning generator issues a warning. In particular, if a checked exception can possibly be pending at the end of a native method and if the class of the exception is not a subclass of one of the exception classes listed in the native method's signature, then a warning is issued.

The warning generator for mishandling JNI exceptions also uses the information from exception analysis and includes two additional components: (1) a static analysis for identifying unsafe operations; (2) a "warning recovery" mechanism that attempts to generate just one warning message for each distinct bug. Section 6 presents the technical details of the warning generator for mishandling JNI exceptions.

5. Scalable and Precise Exception Analysis

The major stages in TurboJet's exception analysis is depicted in Fig. 4. As mentioned before, the goal of exception analysis is to find out the set of possible pending JNI exceptions at every program point of native methods. Its core is a two-stage design, motivated by the following two observations.

1. Library code in a JNI package does not affect Java's exception 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

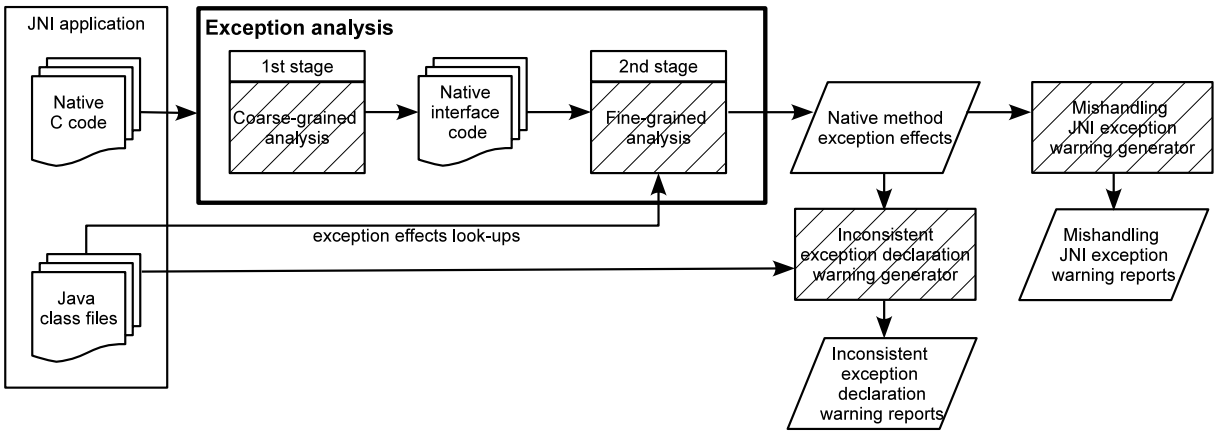

Figure 4: System architecture of TurboJet.

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.

TurboJet's exception analysis uses a two-stage design to take advantage of these observations. The first stage 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 (path and context sensitivity) for accurate analysis. This stage takes into considerations of JNI function calls and performs exception-effect look-ups in Java class files since native code can call back Java methods, which may throw exceptions. The second stage is at the heart of TurboJet's exception analysis.

5.1. Separating interface and library code

The first stage of TurboJet'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] In particular, native code interacts with Java through the JNI functions accessible from a global JNI environment pointer passed from Java. This design makes identification of interface code straightforward.

Accordingly, a native function is defined as part of the interface code if

1. it invokes a JNI function through the environment pointer, 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. We have implemented a straightforward worklist algorithm that iterates through all functions and identifies invocations of JNI functions via the JNI environment pointer. The algorithm is used 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.

Having the first-stage exception analysis helps TurboJet achieve high efficiency, as our experiments demonstrate (see Section 7). Briefly, without the first-stage analysis, TurboJet's analysis time would be increased by more than an order of magnitude 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 application 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.

[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 for a well-defined native method.

(a)

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

(b)

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

Figure 5: Examples for illustrating the need for path and context sensitivity.

5.2. Fine-grained tracking of exception states

The second stage of exception analysis is a fine-grained analysis that is path-sensitive and context-sensitive. By applying the fine-grained analysis on the remaining interface code, TurboJet can precisely identify the exception effects at every program point.

Before presenting our solution, we first discuss requirements of performing accurate tracking.

5.2.1. Requirements of TurboJet'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. 5(a) presents such an example involving `GetIntArrayElements`, which returns the null value and throws an exception when it fails.

Invoking `GetIntArrayElements` results in two possible cases: an exception is thrown and `p` equals `NULL`; no exception is thrown and `p` gets a non-null value. That is, the value of `p` is 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. 5(b) uses a two-step process for throwing an exception: first obtaining a reference to the class of the exception and then throwing 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. 5(b) depends on what the context passes in as the argument.

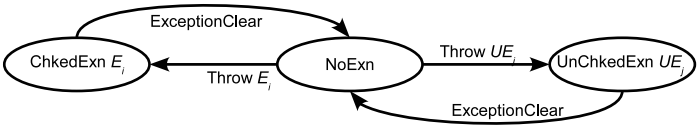

Figure 6: An incomplete FSM specification of exception-state transitions.

Clearly, it is not possible for TurboJet to infer in every case the exact exception effects. For instance, suppose a JNI program calls back a Java method and TurboJet cannot determine which Java method it is. In such cases, TurboJet has to be conservative and assumes 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.

5.3. An FSM specification of exception-state transitions

TurboJet uses an FSM (Finite State Machine) to track how exception states can be changed by JNI functions. For instance, an exception becomes pending after `Throw`; after `ExceptionClear`, the current pending exception is cleared. Fig. 6 presents an FSM specification. It is incomplete as only a few JNI functions are included for brevity. The following table summarizes the meaning of the states in the FSM:

`NoExn`	No JNI exception is pending
`ChkedExn` E_i	A checked exception E_i is pending
`UnChkedExn` UE_j	An unchecked exception UE_j is pending

The FSM has a state for each specific type of exceptions, including both checked and unchecked exceptions.

5.4. Path-sensitive analysis

Static-analysis algorithms that capture full path sensitivity (e.g., [10]) are rather slow since they track all execution states. Fortunately, we are interested only in exception states and transitions between exception states are described by an FSM. TurboJet adopts ESP, proposed by Das *et al.* [11]. It 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 more detailed description. In the end, ESP infers information of the following format *at each control-flow edge*:

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

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

Figure 7: Syntax of Java types that TurboJet tracks.

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, TurboJet uses the FSM specified in Fig. 6. That is, a property state is an exception state. For the execution state, TurboJet tracks two kinds of information: (1) constant values of variables of simple types; and (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, TurboJet tracks their constant values using an interprocedural and conditional constant propagation [12]. 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 for other operations.

Take the program in Fig. 5(a) as an example. After `GetIntArrayElements`, there are two symbolic states encoded as follows.

$$
\begin{array}{l}
\{\langle \texttt{NoExn}, \{\, p = \top \,\}\rangle, \\
\ \langle \texttt{UnChkedExn OutOfMemoryError}, \{\, p = 0 \,\}\rangle\}
\end{array}
$$

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 `OutOfMemoryError` 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 $\bot$ 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 accurately track exception states, it is necessary to infer more Java-side information about these references. For instance, since the JNI function `Throw` takes a `jthrowable` reference, TurboJet has to know the class of the Java object to infer the exact exception class it throws. TurboJet uses a simple type system to capture this kind of information. The type system is presented in Fig. 7. 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 an 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. 8 presents a more elaborate example demonstrating how path sensitivity works in TurboJet. The JNI program invokes a callback method in class `Demo`. We assume that the callback 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) finding a reference to the class `Demo`; (2) allocating an object; (3) obtaining the method ID of the callback 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. 8 contains annotation that specifies what TurboJet's exception analysis infers. Notice that after `FindClass`, there are two symbolic states representing two cases. The following if-statement 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`”. TurboJet 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 TurboJet 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. In this case, it is assumed to declare an `IOException`.

5.5. 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 [11] and is not sufficient for JNI programs. The need for more fine-grained contexts is because of a common programming pattern in JNI programs: for convenience, JNI programs typically use a set of utility functions for accessing fields and throwing exceptions. These utility functions take names of fields, methods, classes, or exceptions as

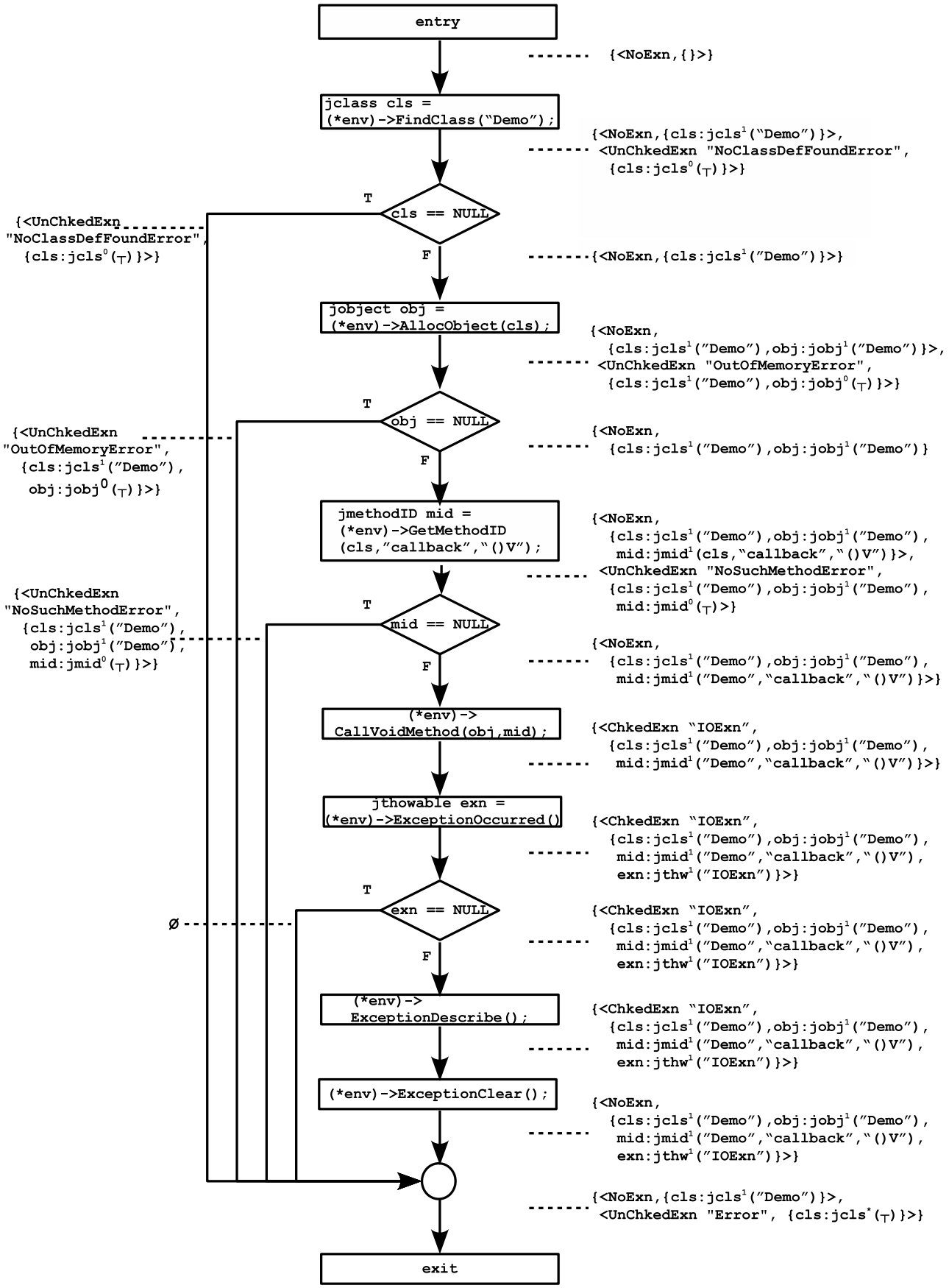

Figure 8: Example of TurboJet path sensitivity.

parameters and are called multiple times with different names. To accurately track Java types in native code, our system has to take the context into account.

Figs. 9 and 10 present a typical JNI program for demonstrating the need. In the example, `ThrowByName` is a utility function, which takes the name of an exception as a parameter and uses a sequence of JNI functions to throw the exception. The function is used by the `Java_Demo_main` function in two contexts, denoted by `L1` and `L2`. At `L1`, exception `E1` is thrown and exceptions `E2` at `L2`.

ESP as shown Fig. 9 merges the execution states of the two call sites when analyzing `ThrowByName` since their property states are the same; both are in the `NoExn` state. As a result, the value of parameter `cn` in `ThrowByName` gets $\top$. Consequently, the analysis determines that `ThrowByName` has the "`ChkedExn Exception`" effect. But in reality, `E1` can only be thrown at `L1`, and `E2` at `L2`. Some unchecked exception may be thrown as well. For brevity, the figure includes only "`NoClassDefFoundError`".

TurboJet as shown in Fig. 10 improves the context by using the call-string approach (of length one) [13]. 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, TurboJet is able to infer the exception effects accurately for the program in Fig. 9. In the figure, we use "_m" to represent a special (unknown) context that invokes the `Java_Demo_main` function. Another note about the figure is that the "`NoClassDefFoundError`" unchecked exception disappears because for each context the analysis determines the exact name of the exception and our system is able to infer `FindClass` always succeeds for a specific name.

The overall interprocedural algorithm is given in the appendix, using notation similar to 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.

5.6. Transfer functions for JNI functions

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 [14].

5.7. Merging symbolic states

At a merge point of a control flow graph, TurboJet groups symbolic states in the following fashion. It merges two symbolic states if the exception state of the first is a subclass of the exception state of the second. Since the subclass relation is reflexive, TurboJet's merge function extends ESP's, which merges two symbolic states if their exception states are the same. Grouping symbolic states by using a parent class of the exceptions is conservative. This may introduce imprecision. The advantage is that by having less symbolic states, the analysis runs faster.

Input: Program P
Output: A list of warning locations in P
Notation: 1. $op(l)$ stands for the operation at location l in P
2. $\text{ss}_\circ(l)$ stands for the symbolic state at the entry of node for l

```
BEGIN: ss∘ = exceptionAnalysis(P);
for each location l in P do
  if (⟨ps, es⟩ ∈ ss∘(l)) and (ps ≠ NoExn) and (unSafe(op(l)))
  then
    Issue a warning for location l;
    P=warningRecovery(P);
    goto BEGIN
  end if
end for
```

Figure 11: High-level steps of warning generation for mishandling JNI exceptions.

6. Finding Bugs of Mishandling JNI Exceptions

In this section, we discuss how TurboJet utilizes the information generated by exception analysis to find bugs of mishandling JNI exceptions. By definition 2 (on page 5), two conditions should be met for such kind of bugs: (1) a JNI exception is pending; and (2) the next operation is an unsafe operation. The first condition is determined by the result from exception analysis.

Fig. 11 presents pseudo code that highlights the major steps of warning generation for mishandling JNI exceptions. First, it performs exception analysis on an input program. We use notation $\text{ss}_\circ$ to stand for the result of exception analysis. It is a function mapping locations of P to the symbolic states calculated by exception analysis. In particular, $\text{ss}_\circ(l)$ is the symbolic state at the entry of location l. It is calculated as the join of symbolic states of all edges that flow into the node for l in the control-flow graph.

After exception analysis, a warning is issued for a location if the exception analysis indicates a possible pending exception *and* next to that location an unsafe operation is identified. Finally, after a warning is issued, it performs "warning recovery" which transforms the old program and re-computes exception analysis. We next discuss how TurboJet decides whether an operation is unsafe and how it performs warning recovery.

6.1. Determining unsafe operations

Our strategy for determining unsafe operations is an algorithm of whitelisting plus static taint analysis.

```
int JNICALL Java_Demo_main(JNIEnv* env){
 char* x = "E1";
 ThrowByName(env,x);//L1
 (*env)->ExceptionClear();
 X = "E2";
 ThrowByName(env,x);//L2
 return 0;
}

void ThrowByName(JNIEnv* env, const char*cn){
 jclass cls = (*env)->FindClass(cn);
 if(cls != NULL){
  (*env)->ThrowNew(cls);
 } else{
   // do nothing
 }
}
```

```
-------NoExn:{<NoExn,{}>}
-------NoExn:{<NoExn,{x = "E1"}>}
       NoExn:{
-------<ChkedExn "Exception",{x = "E1"}>,
       <UnChkedExn "NoClassDefFoundError",{x = "E1"}>}
-------NoExn:{<NoExn,{x = "E1"}>}
-------NoExn:{<NoExn,{x = "E2"}>}
       NoExn:{
-------<ChkedExn "Exception",{x = "E2"}>,
       <UnChkedExn "NoClassDefFoundError",{x = "E2"}>}
       NoExn:{
-------<ChkedExn "Exception",{x = "E2"}>,
       <UnChkedExn "NoClassDefFoundError",{x = "E2"}>}
-------NoExn:{<NoExn,{cn = ⊤}>}
       NoExn:{
-------<NoExn,{cn = ⊤,cls:jcls^1(⊤)}>
       <UnChkedExn "NoClassDefFoundError",{cn = ⊤,cls:jcls^0}>}
-------NoExn:{<NoExn,{cn = ⊤,cls:jcls^1(⊤)}>}
       NoExn:{
-------<ChkedExn "Exception",{cn = ⊤,cls:jcls^1(⊤)}>,
       <UnChkedExn "NoClassDefFoundError",{cn = ⊤,cls:jcls^1(⊤)}>}
       NoExn:{
-------<UnChkedExn "NoClassDefFoundError",{cn = ⊤,cls:jcls^0}>}
       NoExn:{
-------<ChkedExn "Exception",{cn = ⊤,cls:jcls^1(⊤)}>,
       <UnChkedExn "NoClassDefFoundError",{cn = ⊤,cls:jcls^*}>}
```

Figure 9: Exception analysis using ESP.

6.1.1. Whitelisting

The whitelist is comprised of those operations that are absolutely safe to use when an exception is pending. In general, after an exception is thrown, a JNI program either (1) cleans up resources and returns the control to Java, or (2) handles and clears the exception itself using JNI functions such as `ExceptionClear`. Appendix A lists the set of operations that are on the whitelist.

6.1.2. Static taint analysis

A pure whitelist strategy, however, would result in too many false positives (see the experiments section). As an example, the following code is considered safe, but a warning would be issued by the whitelisting strategy since "`a=a+1`" is not on the whitelist.

```
int *p = (*env)->GetIntArrayElements(arr,NULL);
if ((*env)->ExceptionOccurred()) {
       a=a+1; return a;}
```

We cannot put plus and assignment operations on the whitelist because that would allow statements like "`a=(*p)+1`" to escape detection.

Our idea is to use static taint analysis to decide the safety of operations that are not on the whitelist. In static taint analysis, a *taint source* specifies where taint is generated. A *taint sink* specifies unsafe ways in which data may be used in the program. A *taint propagation* rule specifies how taint is propagated in the program.

Our system uses taint sources to model where faults may happen and use static analysis to track fault propagation. In general, a fault is an accidental condition that can cause a program to malfunction. One benefit of static taint analysis is that it can accommodate various sources of faults. For example, in an application that can receive network packets that are controllable by remote attackers, all network packets can be considered being tainted because their contents may be arbitrary values. In the JNI context, data passed from Java to C can be considered being tainted because attackers can write a Java program and affect those data, as the example in Fig. 2 shows. Our framework

```
int JNICALL Java_Demo_main(JNIEnv* env){
                              --------(_m, NoExn):{<NoExn,{}>}
 char* x = "E1";
                              --------(_m, NoExn):{<NoExn,{x = ″E1″}>}
 ThrowByName(env,x);//L1
                              --------(_m, NoExn):{<ChkedExn "E1",{x = ″E1″}>}
 (*env)->ExceptionClear();
                              --------(_m, NoExn):{<NoExn,{x = "E1″}>}
 X = "E2";
                              --------(_m, NoExn):{<NoExn,{x = "E2″}>}
 ThrowByName(env,x);//L2
                              --------(_m, NoExn):{<ChkedExn "E2″,{x = "E2″}>}
 return 0;
                              --------(_m, NoExn):{<ChkedExn "E2″,{x = "E2″}>}
}

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

Figure 10: Exception analysis using TurboJet.

is flexible about how taint is generated and propagated and can thus accommodate various fault models.

In our implementation, our fault model is comprised of the following parts:

- Certain JNI functions may fail to return desired results and are sources of faults. For instance, `NewIntArray` may fail to allocate a Java integer array.

- Certain JNI functions may return direct pointers to Java arrays or strings that can be controlled by attackers. For example, `GetIntArrayElements` may return a direct pointer to a Java integer array when successful. Therefore, the fault model also considers the results of these functions as sources of faults.

- Some library function calls may fail. For example, `malloc` may fail to allocate a buffer. In general, for those external functions that may generate faults, our system requires manual annotation to specify they are fault sources.

All faults in our fault model are associated with pointers. Intuitively, a tainted pointer means that it points to data that may be affected by faults specified in the fault model. For example, the pointer result of `GetIntArrayElements` is tainted because its value may be null or a pointer to a Java buffer (controlled by attackers). Given this intuition, a taint sink (i.e., unsafe ways of using taint) in our setting is an operation where a tainted pointer value is dereferenced for either memory reading or memory writing. Note that the operation of copying a tainted pointer variable to another pointer variable does not constitute a taint sink, as there are no dereferences. On the other hand, the second variable also becomes tainted because of the copy and a dereference of it would be unsafe.

An operation is unsafe if it is not on the whitelist and it may dereference tainted pointers. Now come back to the example earlier in this section. `p` is marked as being tainted as it is the result of `GetIntArrayElements`. As a result, the operation "`a=a+1`" is considered safe because it does not involve the use of any tainted data. On the other hand, "`a=(*p)+1`" would be unsafe. Note that finding an unsafe operation is not a sufficient

condition for issuing a warning. By the algorithm in Fig. 11, a warning is issued only when performing an unsafe operation with a pending exception.

To track taint propagation statically, we have implemented an interprocedural, flow-insensitive algorithm. The flow-insensitivity makes our static taint analysis scalable. The algorithm consists of two steps:

1. A pointer graph is constructed for the whole program. For every pointer in the program, there is a node in the graph to represent the value of the pointer. The edges in the graph approximate how pointer values flow in the program. Our pointer graph is similar to CCured's pointer graph [15], except that our graph has different kinds of edges, which will be discussed shortly.
2. Nodes in the graph that correspond to taint sources are marked as being tainted, and then our algorithm propagates taint along edges of the graph. After this process, any marked nodes are considered being tainted.

Next we discuss more details of our pointer graph. For every pointer in the program, we have a node in the graph to represent the value of the pointer. For example, if a program has a declaration

```
int *p;
```

then its pointer graph has a node for the address of p, or &p, and another node for p. This is because both p and &p are pointers. Similarly, if the program has

```
int **q;
```

then the graph has a node for the address of q, a node for q, and a node for *q; all three are pointer values.

There are two kinds of edges in the graph. The first kind is *flow-to* edges. A directed flow-to edge from node one to node two exists if the pointer value of node one may directly flow to the pointer value for node two. For example, an assignment from pointer one to pointer two would result in a flow-to edge.

The second kind of edges is *contains* edges. It allows our pointer graph to be sound in the presence of aliases. A contains edge exists from node one to node two if the storage place pointed to by the pointer for node one may contain the pointer for node two. For example, given the declaration "`int *p`", there is a contains edge from the node for &p to the node for p. In this example, a contains edge is like a points-to edge in alias analysis. Contains edges allow us to handle C structs. A pointer to a C struct contains all the pointers inside the struct.

Fig. 12 presents an example program and its pointer graph. Dotted edges are contains edges; solid edges are flow-to edges. The node "`ret(GIAE)`" represents the pointer value returned by `GetIntArrayElements`. The flow-to edge from this node to the node for "p" exists because of the assignment on line 3. The flow-to edge from &p to q is because of the assignment on line 2.

Our algorithm also propagates flow-to edges along contains edges. This is because when a pointer value flows to another pointer value, the storage places pointed to by these two values

```
1  int *p;
2  int **q = &p;
3  p = (*env)->GetIntArrayElements(arr,NULL);
4  int len = (*env)->GetArrayLength(arr);
5  for (i=0; i<len; i++) {
6    if ((*q)[i]>0) {
7      sum += (*q)[i];}
8  }
```

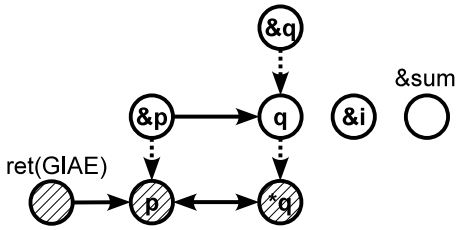

Figure 12: An example program and its pointer graph. The program takes a Java integer array and computes the sum of all positive elements. The nodes with shading are tainted nodes.

may be aliases. As a result, implicit flows exist between the aliases. An example of the propagation of flow-to edges along contains edges in Fig. 12 is the propagation of the flow-to edge from &p to q along the contains edges that are from &p to p and from q to *q. As a result, the flow-to edges from p to *q and from *q to p are added.

Having constructed the pointer graph, it is easy to compute what pointer values in the example program may be tainted. First, our fault model specifies that the return value of `GetIntArrayElements` may be tainted. This taint is then propagated in the pointer graph along flow-to edges, which in turn taints the nodes for p and *q. Given this result, operations on line 6 and 7 are unsafe as they dereference tainted pointer "*q".

For external library functions without source code, the algorithm accepts hand annotations about taint propagation as their models. A library function without annotation is assumed to have the default taint propagation rule: the output is tainted if and only if any of its arguments is tainted. The default rule can be overwritten with hand annotations. We use CIL's attribute language for such manual annotation. For instance, an annotation can say that the output is always tainted and independent of the arguments; in this case, the function is a taint source. A different annotation might say that whether the output is tainted depends on a specific argument.

Calling any external functions that take tainted pointers as parameters is considered unsafe—they might dereference those pointers. This is a source of inaccuracy in our analysis. We could enhance the precision by allowing hand annotation describing how parameters are used in external functions; this is left as future work. We treat JNI functions as external library functions. Therefore, if a JNI function is not on the white list, an invocation of the JNI function is considered unsafe if it takes tainted pointers as parameters.

Our algorithm for constructing pointer graphs handles most C features, including function calls and returns, structs, unions,

and others. One limitation of the pointer-graph construction at this point is it does not construct flow-to edges for inlined assemblies.

6.2. Warning Recovery

The purpose of warning recovery is to suppress duplicate warnings. To illustrate its necessity, we use the example in Fig. 12. At both lines 6 and 7, an exception thrown by `GetIntArrayElements` may be pending and operations at these two places are unsafe since they dereference the tainted pointer "`*q`". A naïve system would issue warnings for both lines 6 and 7. However, both warnings are due to the error of mishandling JNI exceptions possibly thrown at line 3, and can be considered duplicates. Ideally, only one warning should be issued.

To suppress this kind of duplicates, we have implemented a warning-recovery strategy. First, our system remembers information about which exceptions can reach which location. For line 6 of the example problem, the exception raised at line 3 can reach line 6. Our system records location information by augmenting exception states. In particular, `ChkedExn` E becomes "`ChkedExn` E from $\{l_1, ..., l_n\}$". If the exception state at a location l is "`ChkedExn` E from $\{l_1, ..., l_n\}$", it means that exception E thrown at l_1 to l_n might reach l. We augment "`UnChkedExn` E" similarly.

Next, after a warning is issued, our system inserts an `ExceptionClear` statement immediately after all locations included in the abstract state. For example, after a warning for line 6 is issued, an `ExceptionClear` statement is inserted after line 3, whose exception reaches line 6.

After inserting the `ExceptionClear` statements, our system re-computes exception analysis and continues issuing warnings. Because of the inserted statements, duplicate warnings due to the same exception source are suppressed. This strategy allows us to suppress the warning at line 7.

Note that our system inserts `ExceptionClear` purely for warning recovery, not for transforming the program into a correct or semantically equivalent one.

7. Prototype Implementations and Evaluation

To evaluate TurboJet, we conducted experiments to evaluate its accuracy, effectiveness and efficiency. Our static analysis is implemented in the CIL framework [16], a tool set for analyzing and transforming C programs. Before our analysis is invoked, the CIL front end converts C to the CIL intermediate language. The conversion compiles away many complexities of C, thus allowing our system to concentrate on a relatively clean subset of C. CIL's parser does not support C++ programs, so our analysis is limited to C programs only. There are also a few limitations in CIL's parser, and we had to tweak a few programs' syntax so they are acceptable to CIL's parser. Our system has a total about 6,600 lines of OCaml code, including 2,000 lines for constructing pointer graphs. In the exception analysis, we used a may-alias analysis module and a call-graph module included in the CIL. Our system also used JavaLib [17] to access information from Java class files. All experiments were carried out on a Linux Ubuntu 9.10 box with Intel Core2 Duo CPU at 3.16GHz and with 512MB memory.

We compiled a set of JNI packages for experimentation. The packages and their numbers of lines of C source code are listed in Table 1. Statistics for Java class files are not included because TurboJet needs only Java type signatures for its analysis. 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. (1) `java-gnome` [18] is a package that allows Java to use the GNOME desktop environment; (2) `jogl` [19] is a library that allows OpenGL to be used in Java. (3) `libec` is a library for elliptic curve cryptography; OpenJDK [20] provides JNI bindings for interfacing with the library; (4) `libreadline` [21] is a package that provides JNI wrappers for accessing the GNU readline library; (5) `posix 1.0` [22] is a package that provides JNI wrappers for accessing OS system calls; (6) `spread` [23] 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. Other packages are also selected as they are well-known Java applications that consist of JNI code and native components.

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

1. How accurate is TurboJet at uncovering errors of inconsistent exception declarations and mishandling JNI exceptions? Does it generate too many false positives along the process?
2. How efficient is TurboJet in terms of analysis time? Does it scale to large programs?
3. Is the two-stage exception analysis necessary?

7.1. Accuracy

Table 1 presents the number of warnings and true bugs of both types of bugs in the set of benchmark programs.

7.1.1. Results on inconsistent exception declarations

Conceptually, there are two categories of bugs. The first category contains those bugs when a native method does not declare any exceptions but its implementation can actually throw some exceptions. The second category contains those bugs 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 similar 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.

JNI Package	LOC	Inconsistent exception declarations			Mishandling JNI exceptions		
	(K)	Warnings	True Bugs	FP(%)	Warnings	True Bugs	FP(%)
java.io	2	3	1	67	0	0	0
java.lang.math	11	0	0	0	0	0	0
java.lang.non-math	3	1	1	0	8	5	38
java.net	10	4	0	100	88	60	32
java.nio	0.4	1	1	0	8	8	0
java.security	0.01	0	0	0	0	0	0
java.sql	0.02	0	0	0	0	0	0
java.util.timezone	0.7	0	0	0	0	0	0
java.util.zip	14	2	2	0	24	20	17
java.util.other[a]	0.02	0	0	0	0	0	0
java-gnome	6	5	3	40	17	13	24
jogl	0.6	2	2	0	5	5	0
libec	19	1	1	0	0	0	0
libreadline	2	2	2	0	0	0	0
posix	2	1	1	0	40	36	10
spread	28	3	3	0	0	0	0
TOTAL	99	25	17	32	190	147	23

[a] `java.util.other` includes all native code in `java.util` but not in `java.util.timezone` and `java.util.zip`.

Table 1: Accuracy evaluation of TurboJet.

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

Figure 13: A typical example of false positives.

False positives. There are 8 false positives. All false positives are caused by the imprecision when tracking the execution state. Fig. 13 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 TurboJet 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 TurboJet'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.

7.1.2. Results on mishandling JNI exceptions

The warnings of mishandling JNI exceptions in the table are the results of applying strategies of whitelisting, static taint analysis and warning recovery. The overall false positive rate among all benchmark JNI programs is 23%. Our system achieves a relatively high precision. Of the 147 true bugs, 129 of them are because of implicit throws, while the rest are because of explicit throws. That is, the majority of the errors are because programmers forgot to check for exceptions after calling JNI functions. This result is consistent with our expectation.

JNI packages	V1	FP(%)	V2	FP(%)
java.io	0	0	0	0
java.lang.math	0	0	0	0
java.lang.non-math	17	71	38	87
java.net	132	55	318	81
java.nio	16	50	30	73
java.security	0	0	0	0
java.sql	0	0	0	0
java.util.timezone	0	0	0	0
java.util.zip	35	43	48	58
java.util.other	0	0	0	0
java-gnome	24	46	83	84
jogl	5	0	5	0
libec	0	0	0	0
libreadline	0	0	0	0
posix	66	46	108	67
spread	0	0	0	0
TOTAL	295	50	630	77

Table 2: Experimental results for assessing effectiveness of warning recovery and static taint analysis

False positives. False positives are mainly of two kinds. The first kind includes those places where external library functions are invoked with tainted pointer parameters. For soundness, our system issued warnings for them because they might dereference the tainted pointers. For future work, this kind of false positives could be removed by either including the source code of the library functions, or by adding additional manual annotation about how parameters are used. The second kind of false positives are because of flow insensitivity of our static taint analysis. Our design favored scalability and we believe the overall FP rate supports this tradeoff.

To assess the effectiveness of warning recovery and static taint analysis, we have carried out two additional sets of experiments. First, we tested a version of the system without warning recovery; we denote this version by V1. We also tested a version with neither warning recovery nor static taint analysis; we denote this version by V2. Table 2 presents the results. Without warning recovery, the overall FP rate would be 50%, as compared to 23%. Further removing the component of static taint analysis would hike the FP rate to 77%. These experiments demonstrated that warning recovery and static taint analysis are very effective in terms of reducing the number of false positives.

7.1.3. False negatives

TurboJet is designed to be conservative, though it is possible for TurboJet to have false negatives due to errors in its implementation. So we first conducted manual audit in half of the packages of our benchmark programs to obtain the ground truth, and compared it with the results reported by TurboJet. In this way, we found and fixed several bugs in early implementations of TurboJet. Clearly, this is not a proof that TurboJet is sound. For the claim of soundness, we would need to (1) take a formalized semantics of C and the JNI, (2) formalize TurboJet, and (3) show that it can catch all relevant bugs. Although there has been efforts on formalizing C [24, 25] and the JNI [26], they target only subsets of C and the JNI and omit many important language features. Therefore, a formal proof of soundness remains a difficult task. On the other hand, there are many places where the system would be obviously unsound if we did not make a specific design choice. One example we have discussed before is issuing warnings when an external function takes tainted pointers as parameters. Without doing that, the system would be unsound as the external function might dereference the pointers.

7.2. Efficiency

Table 3 presents the analysis time of the TurboJet system (for detecting both types of bugs) on the benchmark programs. As we can see, TurboJet's coarse-grained exception analysis (first stage) is very efficient, taking only 420 μs for all packages. We group the time for the second-stage exception analysis and the time for warning generation into the column named "2nd stage time". It dominates the total time taken by TurboJet. In all, it takes about 14 s for all packages.

Effectiveness of the two-stage system. Table 3 also presents statistics that are used to evaluate the effectiveness of the two-stage design of TurboJet's exception analysis. For each package, the third column (LOC (K) retained/reduced) shows the lines of source code retained and reduced by the coarse-grained analysis. The sum of the two numbers is the total lines of source code of the package. 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 97%. Only a small portion of code is left for the fine-grained analysis.

The column "interface/library LOC (K) (manual separation)" shows the lines of interface code and library code determined by a manual separation. For each package, the sum of the two numbers determined by the manual separation is the size of the package and is therefore the same as the sum in the third column (LOC (K) retained/reduced). The manual separation 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 (K) 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 rest of TurboJet without going through the first stage. This resulted in a dramatic slowdown in runtime.

The experiments demonstrate that our system is efficient and scalable. For nearly 100K lines of code, it took about 14 s to examine all of them.

7.3. Comparison with previous studies

To further validate the design of TurboJet, we compared our work with our two previous studies that used alternative design approaches for exception analysis.

7.3.1. Comparison with an alternative exception analysis

In our previous work on finding bugs of mishandling JNI exceptions [1], we implemented an interprocedural exception analysis. It calculates whether an exception is pending at every program point, but does not determine the set of possible pending exceptions. Since its goal is simpler, it uses a specially designed lattice to get a primitive form of path sensitivity. Furthermore, it is not context sensitive. TurboJet's exception analysis was started with that algorithm but we quickly discovered its imprecision led to too many false positives when calculating exception effects. The previous system has an acceptable false-positive rate only because its exception analysis is combined with other kinds of analysis (in particular, static taint analysis); this strategy worked well when identifying bugs because of mishandling JNI exceptions but would lead to high false-positive rates when calculating exception effects. As a result, we gradually moved to a path-sensitive and context-sensitive system. The need for context sensitivity was also pointed out by J-Saffire [27], which is why they used polymorphic type inference rather than a monomorphic version.

Table 4 presents comparison results between TurboJet and a version of TurboJet that uses the crude exception analysis as

JNI Package	1st stage time (μs)	LOC (K) retained/reduced	2nd stage time (s)	Total time (s)	interface/library LOC (K) (manual separation)	Time without 1st stage (s)
java.io	30	0.7/1	3.07	3.07	2/0	26.7
java.lang.math	30	0/11	0	30μs	2/9	2.09
java.lang.non-math	30	0.5/3	0.1	0.1	3/0	37.11
java.net	100	0.5/9	3.43	3.43	10/0	101.13
java.nio	10	0.04/0.3	0.50	0.50	0.4/0	0.31
java.security	0	0/0.08	0	0	0.08/0	0.05
java.sql	0	0/0.02	0	0	0.02/0	0.00171
java.util.timezone	0	0/0.7	0	0	0.7/0	2.12
java.util.zip	20	0.4/14	1.13	1.13	0.8/13	72.88
java.util.other	0	0/0.02	1.00	1.00	0.02/0	0.10
java-gnome	15	0.3/5	1.71	1.71	0.6/5	45.27
jogl	5	0.3/0.3	1.09	1.09	0.3/0.3	17.37
libec	20	0.1/19	0.61	0.61	0.4/19	70.21
libreadline	20	0.1/2	0.05	0.05	0.7/1	19.31
posix	20	0.3/2	0.08	0.08	2/0	11.28
spread	60	0.3/27	0.51	0.51	2/25	12.97
TOTAL	360	4/94	13.28	13.28	25/72	418.9

Table 3: Efficiency evaluation of TurboJet.

in [1]. `CrudeExn` stands for the version with the crude exception analysis. The table presents false-positive rates and running times of the two versions for finding inconsistent exception declarations.

It is clear from the results that TurboJet's exception analysis is more accurate. The improvement in terms of reducing false positive rate is more than 50%. Since `CrudeExn` only determines whether or not there is an exception pending from the native side, a warning is issued if a native method's type signature declares an exception class that is a strict subclass of `java.lang.Exception` and `CrudeExn` determines that an exception is pending from the native side. As a result, it causes higher false-positive rates.

7.3.2. Comparison with an empirical study

A previous empirical study on Sun's JDK 1.6 found a number of errors of mishandling JNI exceptions [5]. The study used grep-based scripts to examine 38,000 lines of C code and found 35 counts of mishandling JNI exceptions. Table 5 shows the comparison between the results of TurboJet and the empirical study on the set of JDK directories that are common in both studies. TurboJet uncovered all of the errors that were found in the previous study; it also discovered more errors. We like to note, however, that in deciding whether a warning is a true bug, we exercised a more conservative approach and adhered to the JNI's specification more closely than the one taken in the previous study. For example, TurboJet assumes JNI function `GetFieldID` can throw `NoSuchFieldError`, `ExceptionInInitializerError`, and `OutOfMemoryError` exceptions, and issues a warning when insufficient error checking occurs. The previous study took a more liberal measure on such cases; for example, it ignored the possibility of `OutOfMemoryError` exception in the case of `GetFieldID`. This discrepancy contributes to the majority of the difference in errors shown in the table (roughly about two thirds). Another benefit of our tool is its much lower false-positive rate, which means much less manual work to sift through warnings for identifying true bugs.

Results	The previous study	TurboJet
Warnings	556	132
True bugs	35	93
FP %	93.7	29.5

Table 5: Comparison with the previous study [5] (of the 35 errors in the previous study, 11 are due to explicit throws and 24 due to implicit throws).

8. An Eclipse Plug-In Tool

We implemented TurboJet as an Eclipse plug-in tool. This tool identifies both types of bugs described in this paper, and provides a developer-friendly graphic user interface. The plug-in is built as an additional checker that extends Codan [28], an open source project that performs a variety of static code analyses and checking on C/C++ code in Eclipse.

The TurboJet plug-in has a user-interface frontend and a code-analysis backend. For the frontend, the plug-in leverages Codan to extract information from source code, to configure settings (including the path to Java class files and which types of bugs should be identified), and to perform code navigation. The plug-in passes information on both the Java class files and C source code to the backend, which invokes the exception analysis system.

JNI package	FP (%)		Time	
	CrudeExn	*TurboJet*	*CrudeExn*	*TurboJet*
java.io	80	67	1.25	3.07
java.lang.math	0	0	5μ	30μ
java.lang.non-math	50	0	0.05	0.1
java.net	100	100	1.25	3.43
java.nio	75	0	0.14	0.50
java.security	0	0	0	0
java.sql	0	0	0	0
java.util.timezone	0	0	0	0
java.util.zip	60	0	0.68	1.13
java.util.other	0	0	0.81	1.00
java-gnome	50	40	0.87	1.71
jogl	75	0	0.45	1.09
libec	67	0	0.44	0.61
libreadline	50	0	0.01	0.05
posix	83	0	0.01	0.08
spread	50	0	0.22	0.51
TOTAL	67	32	6.18	13.28

Table 4: Comparing TurboJet with an alternative exception analysis on finding inconsistent exception declarations.

Like any other checker in Codan, the TurboJet plug-in can be easily configured inside Eclipse. A JNI developer may disable the checker, configure bug levels (e.g., warnings or errors), and select which particular type of bug (i.e., mishandling JNI exceptions or inconsistent exception declarations) to check.

Fig. 14 and Fig. 15 show examples of the TurboJet plug-in's warnings on inconsistent exception declarations and mishandling JNI exceptions, respectively. The problematic code is highlighted with warnings. Developers are provided with warning messages, and can navigate to the buggy code by double-clicking on the messages.

9. 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., [14, 6, 7, 29, 30, 31]). Early work on FFIs were mostly concerned with how to design an FFI and how to provide efficient implementation.

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 ([27, 32, 1]). 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 [33, 27, 34, 32, 1]; (2) In another approach, dynamic checks are inserted at the language boundary and/or in the native code for catching interface errors (see [35]) or for isolating errors in native code so that they do not affect the host language's safety [36, 37] and security [38]; (3) New interface languages are designed to help programmers write safer interface code (e.g., [39]). TurboJet takes the static-analysis approach, which is well-suited for finding bugs in exceptional cases.

We next compare in more detail with three closely related work on using static analysis to find bugs in JNI programs [27, 32, 1]. Our previous system [1] and a system by [32] identify situations of mishandling JNI exceptions in native code. Both systems compute at each program location whether there is a possible JNI exception pending. However, they do not compute specific classes of pending exceptions. To do that, TurboJet has to use a much more complicated static analysis. The analysis tracks information of C variables in native code and correlates them with exception states; 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 [27] 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. To deal with context sensitivity required for analyzing JNI utility functions, J-Saffire performs polymorphic type inference that is based on semi-unification, while TurboJet uses a context-sensitive dataflow analysis. Since TurboJet's context sensitivity uses call strings of length one, there are cases when J-Saffire's type inference can infer more precise information than TurboJet's. For instance, if a string constant is passed two levels down in a function-call chain and used as an argument to `FindClass`, then TurboJet cannot infer the exact Java type of the resulting reference. In practice, however, we did not find this results in noticeable imprecision in TurboJet.

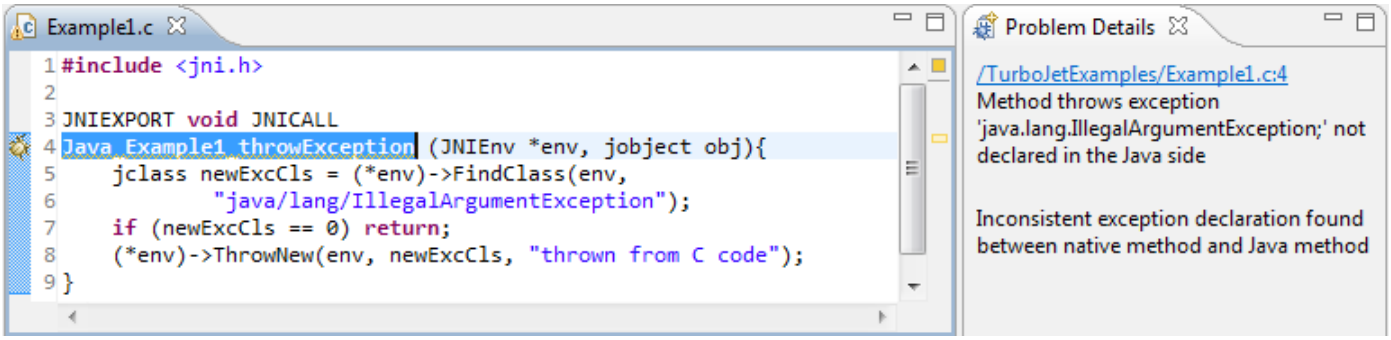

Figure 14: An example of TurboJet plug-in's warning on inconsistent exception declarations.

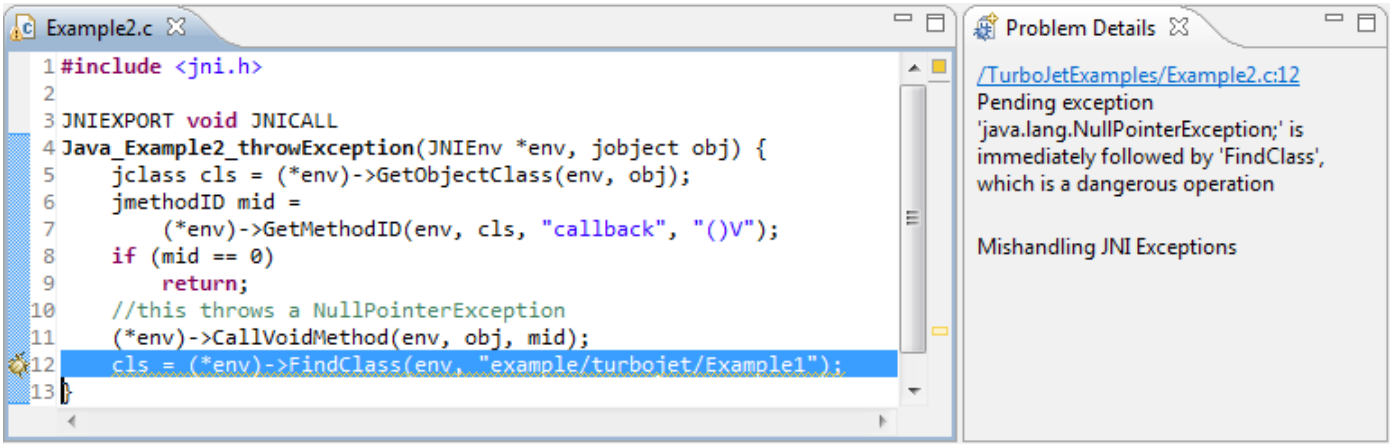

Figure 15: An example of TurboJet plug-in's warning on mishandling JNI exceptions.

Exception analysis. Many systems perform exception analysis for languages that provide built-in support for exceptions. For example, the Jex tool [40] and others (e.g., [41, 42]) can compute what kinds of exceptions can reach which program points for Java programs, which is essential for understanding where exceptions are thrown and caught. A system that performs exception analysis for C++ has also been proposed [43]. Bravenboer and Smaragdakis [44] demonstrate that exception analysis should be performed together with points-to analysis as they are mutually dependent (similar to the mutual dependency between points-to analysis and call-graph analysis). TurboJet performs exception analysis for JNI programs, which has no built-in exception support.

Taint analysis. Our system employs static taint analysis to track "bad" data in exceptional situations. Taint analysis can be performed either dynamically or statically. Dynamic taint analysis (e.g., [45, 46, 47, 48]) incurs substantial runtime overhead (typically over 100%). The benefit is that it usually has higher granularity (e.g., bit-level taint tracking) and it is also more precise than static taint tracking. A recent success was a dynamic taint tracking system in Android that prevents confidentiality violations [49]. Static taint analysis does not incur runtime overhead, but may report false errors. It has been used for identifying vulnerabilities [50, 51, 52, 53, 54], for helping symbolic execution [55], and for identifying where authorization hooks should be placed in an access-control system [56]. One technical difference between our static taint analysis and others is that it depends on a pointer graph. The pointer graph (described in Section 6.1) both approximates taint propagation and is used to cope with aliases; previous systems [50, 57] have separate taint propagation and pointer-analysis modules.

10. Future Work

Although this paper studies the JNI, we believe many of its ideas can be transferred to other Foreign Function Interfaces (FFIs). For instance, the error pattern of mishandling exceptions is not unique to the JNI. Any programming language with a managed environment that allows native components to throw exceptions faces the same issue. Examples include the Python/C API interface [6] and the OCaml/C interface [7].

Second, TurboJet finds mistakes of incorrect control flow when an exception is pending. For example, forgetting a return statement after throwing exceptions falls under its scope. On the other hand, even if the control flow is implemented correctly, native code may still forget releasing resources in some control flow paths. The Python/C interface provides an interesting example. Since native code uses reference counting to manage Python objects, it is easy for programmers to overlook updating reference counts after an exception is thrown, resulting in memory leaks. Weimer and Necula [58] studied how to ensure that resources are properly released according to an

FSM-based specification in the presence of exceptions. They proposed a language feature called compensation stacks, which can dynamically ensure resources are properly released even in exceptional situations. We believe one interesting future project is to statically enforce FSM-based resource usage protocols.

Third, TurboJet performs deep static analysis on C source code and does not work for libraries whose source code is unavailable. One possibility would be to perform static analysis over binary code, but it would be difficult as binary code loses many structured information in source code.

Finally, there remains many interesting and challenging research work in the area of improving multilingual software reliability and security. Previous work that focuses exclusively on one particular language often makes assumptions about foreign language components and treats them as either black boxes or ignore them all together. For example, concurrency safety in Java has been studied extensively, but to our best knowledge at the time of writing this paper, none of the work deals with the case of having native components.

11. Conclusions

Foreign Function Interfaces (FFIs) bring convenience and efficiency to software development but at the same time introduces software security and reliability issues if care is not taken. Exceptions are commonly used in FFIs as ways for the foreign language to report error conditions to the host language. FFI programmers often make mistakes related to exceptions since FFIs do not provide support for exception checking and exception handling.

Making a solid step toward safer and more reliable FFI code, we have designed and implemented a novel static analysis framework for finding bugs in exceptional situations in JNI programs. TurboJet is a system that statically analyze native code (1) to extend Java's rules for checked exceptions to native code and (2) to identify errors of mishandling JNI exceptions. It is both scalable and has high precision thanks to its carefully engineered trade-offs. Our experimental results demonstrated the effectiveness of our techniques. We have also built a practical Eclipse plug-in that can be used by programmers to catch errors in their JNI code. We believe that the techniques introduced in TurboJet are applicable to other environments such as Python, OCaml, and .NET.

Acknowledgment

We would like to thank Martin Hirzel and Kathryn McKinley for their helpful comments. We also want to thank the anonymous reviewers for their constructive comments.

Appendix A. Whitelist

- A list of JNI functions that are safe to call with a pending exception (specified in the JNI Manual [14]): `ExceptionOccurred`, `ExceptionDescribe`, `ExceptionClear`, `ExceptionCheck`, `ReleaseStringChars`, `ReleaseStringUTFchars`, `ReleaseStringCritical`, `Release⟨Type⟩ArrayElements`, `ReleasePrimitiveArrayCritical`, `DeleteLocalRef`, `DeleteGlobalRef`, `DeleteWeakGlobalRef`, `MonitorExit`, `PushLocalFrame`, `PopLocalFrame`.
- The return operation and memory free operation.

Appendix B. Inter-procedural exception analysis

We first describe some notations. A symbolic state $S = D \times X$, where a property state $D = \{$ `NoExn`, `ChkedExn` $E_1, \ldots,$ `ChkedExn` E_n, `UnChkedExn` $UE_1, \ldots,$ `UnChkedExn` $UE_m\}$, 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.

α denotes a function that merges a set of symbolic states:

$$\alpha(ss) = \{\, \langle d, \bigsqcup_{s \in ss[d]} es(s) \rangle \mid d \in roots(ss) \,\}$$

where

$ss[d] = \{\, s \mid s \in ss \wedge ps(s) <: d \,\}$
$roots(ss) = \{\, d \mid ss[d] \neq \emptyset \wedge \forall \langle d', es' \rangle \in ss.\ \neg(d <: d') \,\}$
and $<:$ denotes the subclass relation

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

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

The algorithm is formally described in Algorithms 1 and 2.

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 $AddTrigger(n, n_c, d, ss, \overline{a})$
begin
  $ss' := F_{Call}(fn(n), ss, \overline{a})$;
  $e := Out_T(n)$;
  $ss' := \alpha(ss' \bigcup Info(e, n_c, d))$;
  $Add(e, n_c, d, ss')$;

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
  $\forall\, e, n, d, Info(e, n, d) = \emptyset$;
  $\forall\, f, n, d, Summary(f, n, d) = \emptyset$;
  $e := Out_T(entryNode(f_{entry}))$;
  $Info(e, \_\mathrm{m}, \texttt{NoExn}) := \{[\texttt{NoExn}, \top]\}$;
  $Worklist := \{[dst(e), \_\mathrm{m}, \texttt{NoExn}]\}$;
  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 $n \in Call(\overline{a})$
        $ss_{out} := \emptyset$;
        foreach $d' \in D$ *such that* $ss_{in}[d'] \neq \emptyset$ **do**
          $sm := Summary(callee(n), n, d')$;
          if $sm \neq \emptyset$ **then**
            $ss_{out} := ss_{out} \bigcup sm$;
          $AddTrigger(entryNode(callee(n)), n, d', ss_{in}[d'], \overline{a})$;
        $Add(Out_T(n), n_c, d, \alpha(ss_{out}))$;
      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*

References

[1] S. Li, G. Tan, Finding bugs in exceptional situations of JNI programs, in: 16th ACM Conference on Computer and Communications Security (CCS), 2009, pp. 442–452.

[2] S. Li, G. Tan, JET: Exception checking in the Java Native Interface, in: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2011, pp. 345–358.

[3] B. Lee, M. Hirzel, R. Grimm, 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), 2009, pp. 207–226.

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

[5] G. Tan, J. Croft, An empirical security study of the native code in the JDK, in: 17th Usenix Security Symposium, 2008, pp. 365–377.

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

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

[8] M. Bruntink, A. van Deursen, T. Tourwé, Discovering faults in idiom-based exception handling, in: International Conference on Software engineering (ICSE), 2006, pp. 242–251.

[9] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, Error propagation analysis for file systems, in: ACM Conference on Programming Language Design and Implementation (PLDI), 2009, pp. 270–280.

[10] T. Reps, S. Horwitz, M. Sagiv, Precise interprocedural dataflow analysis via graph reachability, in: 22nd ACM Symposium on Principles of Programming Languages (POPL), 1995, pp. 49–61.

[11] M. Das, S. Lerner, M. Seigle, ESP: path-sensitive program verification in polynomial time, in: ACM Conference on Programming Language Design and Implementation (PLDI), 2002, pp. 57–68.
[12] M. N. Wegman, F. K. Zadeck, Constant propagation with conditional branches, ACM Transactions on Programming Languages and Systems 13 (2) (1991) 181–210.
[13] M. Sharir, A. Pnueli, Two approaches to inter-procedural dataflow analysis, in: S. S. Muchnick, N. D. Jones (Eds.), Program Flow Analysis: Theory and Applications, Prentice-Hall Inc., 1981.
[14] S. Liang, Java Native Interface: Programmer's Guide and Reference, Addison-Wesley Longman Publishing Co., Inc., 1999.
[15] G. Necula, S. McPeak, W. Weimer, CCured: type-safe retrofitting of legacy code, in: 29th ACM Symposium on Principles of Programming Languages (POPL), 2002, pp. 128–139.
[16] G. Necula, S. McPeak, S. P. Rahul, W. Weimer, CIL: Intermediate language and tools for analysis and transformation of C programs., in: International Conference on Compiler Construction (CC), 2002, pp. 213–228.
[17] http://javalib.gforge.inria.fr (2011).
[18] The java-gnome user interface library, http://java-gnome.sourceforge.net (2011).
[19] `https://jogl.dev.java.net` (2011).
[20] OpenJDK 1.7.0, http://openjdk.java.net/.
[21] Java-Readline, `http://java-readline.sourceforge.net` (2003).
[22] Posix for Java, `http://bmsi.com/java/posix/` (2009).
[23] JNI Binding to FlushSpread and Spread, `http://gsd.di.uminho.pt/members/jop/spread-jni` (2004).
[24] M. Norrish, Formalising c in hol, Ph.D. thesis, University of Cambridge (1998).
[25] X. Leroy, Formal certification of a compiler back-end or: programming a compiler with a proof assistant, in: 33rd ACM Symposium on Principles of Programming Languages (POPL), 2006, pp. 42–54.
[26] G. Tan, JNI Light: An operational model for the core JNI, in: Proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS '10), 2010, pp. 114–130.
[27] M. Furr, J. Foster, Polymorphic type inference for the JNI, in: 15th European Symposium on Programming (ESOP), 2006, pp. 309–324.
[28] `http://wiki.eclipse.org/CDT/designs/StaticAnalysis` (2011).
[29] M. Blume, No-longer-foreign: Teaching an ML compiler to speak C "natively"., Electronic Notes in Theoretical Computer Science 59 (1) (2001) 36–52.
[30] 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).
[31] K. Fisher, R. Pucella, J. H. Reppy, A framework for interoperability., Electronic Notes in Theoretical Computer Science 59 (1) (2001) 3–19.
[32] G. Kondoh, T. Onodera, Finding bugs in Java Native Interface programs, in: ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis, ACM, New York, NY, USA, 2008, pp. 109–118.
[33] M. Furr, J. Foster, Checking type safety of foreign function calls., in: ACM Conference on Programming Language Design and Implementation (PLDI), 2005, pp. 62–72.
[34] G. Tan, G. Morrisett, ILEA: Inter-language analysis across Java and C, in: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2007, pp. 39–56.
[35] B. Lee, M. Hirzel, R. Grimm, B. Wiedermann, K. S. McKinley, Jinn: Synthesizing a dynamic bug detector for foreign language interfaces, in: ACM Conference on Programming Language Design and Implementation (PLDI), 2010, pp. 36–49.
[36] G. Tan, A. Appel, S. Chakradhar, A. Raghunathan, S. Ravi, D. Wang, Safe Java Native Interface, in: Proceedings of IEEE International Symposium on Secure Software Engineering, 2006, pp. 97–106.
[37] D. Li, W. Srisa-an, Quarantine: A framework to mitigate memory errors in jni applications, in: Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ), 2011, pp. 1–10.
[38] J. Siefers, G. Tan, G. Morrisett, Robusta: Taming the native beast of the JVM, in: 17th ACM Conference on Computer and Communications Security (CCS), 2010, pp. 201–211.
[39] M. Hirzel, R. Grimm, Jeannie: Granting Java Native Interface developers their wishes, in: ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2007, pp. 19–38.
[40] M. P. Robillard, 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) (2003) 191–221.
[41] D. Malayeri, J. Aldrich, Practical exception specifications, in: Advanced Topics in Exception Handling Techniques, Vol. 4119 of Lecture Notes in Computer Science, Springer, 2006, pp. 200–220.
[42] B.-M. Chang, J.-W. Jo, K. Yi, K.-M. Choe, Interprocedural exception analysis for Java, in: SAC '01: Proceedings of the 2001 ACM symposium on Applied computing, ACM, New York, NY, USA, 2001, pp. 620–625.
[43] P. Prabhu, N. Maeda, G. Balakrishnan, Interprocedural exception analysis for C++, in: Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP), 2011, pp. 583–608.
[44] M. Bravenboer, Y. Smaragdakis, Exception analysis and points-to analysis: better together, in: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, 2009, pp. 1–12.
[45] J. Newsome, D. Song, Dynamic taint analysis for automatic dedection, analysis, and signature generation of exploits on commodity software, in: Network and Distributed System Security Symposium (NDSS), 2005.
[46] W. Xu, S. Bhatkar, R. Sekar, Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks, in: 15th Usenix Security Symposium, 2006, pp. 121–136.
[47] A. Nguyen-tuong, S. Guarnieri, D. Greene, D. Evans, Automatically hardening web applications using precise tainting, in: 20th IFIP International Information Security Conference, 2005, pp. 372–382.
[48] E. Bosman, A. Slowinska, H. Bos, Minemu: The world's fastest taint tracker, in: RAID, 2011, pp. 1–20.
[49] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, A. N. Sheth, Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones, in: USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2010.
[50] B. Livshits, M. Lam, Finding security vulnerabilities in Java applications with static analysis, in: 14th Usenix Security Symposium, 2005, pp. 271–286.
[51] N. Jovanovic, C. Kruegel, E. Kirda, Pixy: A static analysis tool for detecting web application vulnerabilities (short paper), in: IEEE Symposium on Security and Privacy (S&P), 2006, pp. 258–263.
[52] Y. Xie, A. Aiken, Static detection of security vulnerabilities in scripting languages, in: 15th Usenix Security Symposium, USENIX Association, Berkeley, CA, USA, 2006, pp. 179–192.
[53] K. Ashcraft, D. Engler, Using programmer-written compiler extensions to catch security holes, in: IEEE Symposium on Security and Privacy (S&P), IEEE Computer Society, Washington, DC, USA, 2002, pp. 143–159.
[54] W. Xinran, J. Yoon-Chan, Z. Sencun, L. Peng, Still: Exploit code detection via static taint and initialization analyses, in: ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference, IEEE Computer Society, Washington, DC, USA, 2008, pp. 289–298.
[55] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, D. Song, A symbolic execution framework for JavaScript, in: IEEE Symposium on Security and Privacy (S&P), 2010, pp. 513–528.
[56] D. Muthukumaran, T. Jaeger, V. Ganapathy, Leveraging "choice" to automate authorization hook placement, in: 19th ACM Conference on Computer and Communications Security (CCS), 2012, pp. 145–156.
[57] W. Chang, B. Streiff, C. Lin, Efficient and extensible security enforcement using dynamic data flow analysis, in: 15th ACM Conference on Computer and Communications Security (CCS), 2008, pp. 39–50.
[58] W. Weimer, G. Necula, Exceptional situations and program reliability, ACM Transactions on Programming Languages and Systems 30 (2) (2008) 1–51.