Finding Bugs in Exceptional Situations of JNI Programs

Siliang Li
Department of Computer Science and Engineering
Lehigh University
sil206@cse.lehigh.edu

Gang Tan
Department of Computer Science and Engineering
Lehigh University
gtan@cse.lehigh.edu

ABSTRACT

Software flaws in native methods may defeat Java's guarantees of safety and security. One common kind of flaws in native methods results from the discrepancy on how exceptions are handled in Java and in native methods. Unlike exceptions in Java, exceptions raised in the native code through the Java Native Interface (JNI) are not controlled by the Java Virtual Machine (JVM). Only after the native code finishes execution will the JVM's mechanism for exceptions take over. This discrepancy makes handling of JNI exceptions an error prone process and can cause serious security flaws in software written using the JNI.

We propose a novel static analysis framework to examine exceptions and report errors in JNI programs. We have built a complete tool consisting of exception analysis, static taint analysis, and warning recovery. Experimental results demonstrated this tool allows finding of mishandling of exceptions with high accuracy (15.4% false-positive rate on over 260k lines of code). Our framework can be easily applied to analyzing software written in other foreign function interfaces, including the Python/C interface and the OCaml/C interface.

Categories and Subject Descriptors

D.2.4 [**Software**]: Software Engineering—*Software/Program Verification*; D.2.12 [**Software**]: Software Engineering—*Interoperability*

General Terms

Reliability, Security, Verification

1. INTRODUCTION

The increasing popularity of programming languages with managed runtime systems is a boon to computer security. Runtime services such as security sandbox, garbage collection, and exception handling provided by Java and .NET virtual machines make program execution in these platforms much less vulnerable. As another example, Perl's taint mode prevents attacks based on malicious user input. In both cases, managed environments provide a natural and extensible way of enforcing relevant security policies.

To interoperate with software components in other languages, most managed programming languages also support foreign function interfaces (FFIs). The Java Native Interface (JNI) allows Java components to interoperate with native components developed in C, C++, or assembly languages. Similarly, .NET provides the P/Invoke interface for invoking library functions.

Native components are usually the security dark corner of software applications. They are outside of managed environments and relevant security policies cannot be enforced on them. In Sun's JDK 1.6, there are over 800,000 lines of C/C++ code.[1] Any vulnerability in this trusted native code can compromise the security of the JVM. Several vulnerabilities have been discovered [24, 30, 29]. A recent empirical security study [28] on Sun's JDK 1.6 found over 126 software errors in a mere 38,000 lines of C code. 59 of them are security critical.

One of the most revealing aspects of the security study is that many of the discovered errors are due to a discrepancy on how exceptions are handled between Java and the JNI. Managed environments such as the JVM provide runtime support for exception handling, which native components cannot rely on. We next explain why this discrepancy may lead to security vulnerabilities and why it is common in foreign function interfaces.

1.1 Mishandling JNI exceptions

The JNI is Java's mechanism for interfacing with native code written in C, C++, or assembly languages. Programmers use the `native` modifier to declare native methods in Java classes. For example, the `bcopy` method in Figure 1 is declared as a native method. Once declared, native methods can be invoked in Java in the same way as how ordinary Java methods are invoked. Programmers then provide an implementation of the declared native methods (e.g., in C). The implementation can use various JNI functions provided by the JNI interface to cooperate with the Java side. Through these functions, native methods can inspect, modify, and create Java objects, invoke Java methods, catch and throw Java exceptions, and so on. Of interests to this paper are those JNI API functions that are related to exceptions.

[1]Over the time, the size of C/C++ code has been on the increase: JDK 1.4.2 has around 500,000 lines; JDK 1.5 has around 700,000 lines; and JDK 1.6 has around 800,000 lines.

CCS'09, November 9–13, 2009, Chicago, Illinois, USA.

Both Java code and native code may throw exceptions. Native code can throw exceptions through JNI functions such as `Throw` and `ThrowNew`; we call such throws *explicit throws*. Furthermore, many JNI functions throw exceptions to indicate failures. For example, `NewCharArray` throws an exception when the allocation fails; we call such throws *implicit throws*. Both explicit and implicit throws result in a pending exception on the native side. 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.

When an exception is pending on the Java side, a Java Virtual Machine (JVM) automatically transfers the control to the nearest enclosing try-catch block that matches the exception type. In contrast, 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 this subtle difference between the runtime semantics of JNI exceptions and Java exceptions, it is easy for JNI programmers to make mistakes.

Figure 1 presents a toy example that shows how mishandling JNI exceptions may lead to 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 Figure 1 can be fixed quite easily—just insert a return statement after the throwing-exception 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.

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 of 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. A recent empirical study [28] has identified 35 cases of mishandling of JNI exceptions in 38,000 lines of C code.

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 [21] and the OCaml/C interface [12].

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

```
#include <jni.h>
#include "Vulnerable.h"
//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(env, jarr) > 512) {
    JNU_ThrowArrayIndexOutOfBoundsException(env,0);
  }
  jbyte *carr = (*env)->GetByteArrayElements
                        (env, jarr, NULL);
  //Dangerous operation
  strcpy(buffer, carr);
  (*env)->ReleaseByteArrayElements(env,arr,carr,0);
}
```

Figure 1: An example of mishandling JNI exceptions.

1.2 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 mishandling JNI exceptions:

DEFINITION 1. *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*

(a) a JNI exception is pending,

(b) and the next operation is an unsafe operation.

By this definition, the example in Figure 1 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, it leaves open the notion of "unsafe operations". Our strategy of determining unsafe operations will be described in Section 3.2. Second, 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 Figure 1 is a case of buffer overflows and a bug finder targeting buffer overflows would suffice. This is indeed true for that

(a)

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

(b)

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

Figure 2: Two more examples of mishandling JNI exceptions.

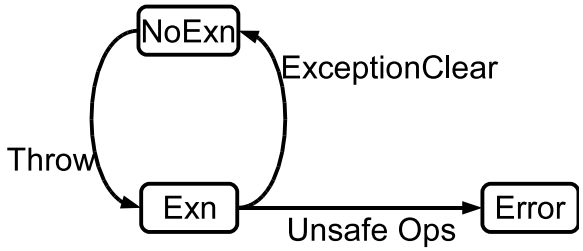


Figure 3: An FSM specification of mishandling JNI exceptions. Only the transitions of `Throw`, `ExceptionClear`, and unsafe operations are included.

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 Figure 2.

In Figure 2(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 Figure 2(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.

1.3 Overview of our static analysis framework

The benefit of static analysis is it examines every control flow path in a program and in general will not miss any security vulnerabilities. It is especially appropriate for catching defects in exceptional situations because they are hard to trigger with normal input and events.

The framework consists of three main components: 1) a dataflow analysis is employed for tracking exception states; the result of the analysis is used to check whether condition a in Definition 1 holds for a given location; 2) a second static analysis is used for identifying unsafe operations; this analysis targets condition b; 3) a "warning recovery" mechanism attempts to generate just one warning message for each distinct bug. The most important technical aspects of each component are summarized as follows:

- To get a high precision algorithm for tracking exception states, we equipped a standard interprocedural dataflow analysis algorithm with a lattice extended with *semantic conditions*. The semantic conditions exploit correlation between exception states and execution states. This idea is general and can be applied to checking any safety properties specified by finite state machines.

- Our strategy for identifying unsafe operations is based on a whitelist plus *static taint analysis*. We use taint to model faults in exceptional situations and employ taint analysis to simulate fault propagation. Our static taint analysis depends on a module that constructs pointer graphs, which approximate taint propagation and cope with aliases.

- To suppress duplicate warnings, we designed and implemented an effective warning recovery system. After issuing a warning, it transforms the input program and recomputes dataflow solutions so that duplicate warnings for distinct bugs are suppressed.

2. RELATED WORK

We discuss related work in four categories: checking temporal safety properties, JNI bug finding, exception analysis, and taint analysis.

Checking temporal safety properties. Mishandling JNI exceptions is a violation of a safety property, which can be specified by a Finite State Machine (FSM). Figure 3 presents a partial FSM specification of mishandling JNI exceptions. Given such an FSM, a number of frameworks can be used to check for its violations, including interprocedural dataflow analysis [22], model checking (see MOPS [4] for an example in the security context), and typestate analysis [26].

However, if these algorithms are naively applied to the JNI context, they would suffer high false-positive rates because they do not have the ability to correlate exception and execution states. To track the correlation, we augment a dataflow lattice with semantic conditions. Property simulation [5] also allows such kind of correlation. Our exception analysis can be thought of as an instance of property simulation, but we encode the correlation between exception and execution states directly as elements in the lattice.

JNI bug finding. A number of systems in the recent years targeted the JNI interface. JSaffire [6] by Furr and Foster is a static tool that can identify type misuses of data passed from Java to C; it does not address the pattern of mishandling JNI exceptions.

A bug finder recently developed by Kondoh and Onodera [11] uses a typestate analysis to identify common JNI mistakes. In particular, it searches for bugs of mishandling JNI exceptions through a typestate analysis based on an FSM similar to the one in Figure 3. Unfortunately, their system is incomplete for the following reasons. First, it models only exceptions thrown by `Call`⟨`Type`⟩`Method` (a JNI function that calls back a Java method). Yet an exception can

be thrown by many other ways: by invoking the JNI function `Throw/ThrowNew`, or by invoking JNI functions such as `GetIntArrayElements`. Second, "unsafe" operations in their analysis are hardcoded to be operations that invoke `Call⟨Type⟩Method`. Under this model, errors in Figure 1 and Figure 2 would not be caught. Finally, their model allows checking exception states only through JNI functions, not through checking other execution states such as the functions' return values. Checking return values is quite common in JNI programming.

SafeJNI [27] is a mostly dynamic mechanism for checking the safety of JNI programs. With respect to JNI exceptions, SafeJNI inserts dynamic checks before JNI functions that should not be invoked with a pending exception. Common to all dynamic mechanisms, SafeJNI incurs runtime overhead and catches errors only if they are manifested during runtime.

Exception analysis. Many systems perform exception analysis for languages that provide built-in support for exceptions. For example, the Jex tool [23] and others (e.g., [15, 2]) can compute what kinds of exceptions can reach which program point for Java programs. This information is essential for understanding where exceptions are thrown and caught. Our system analyzes JNI programs, which has no built-in exception support. The system is concerned with ensuring correct control flow when an exception is pending. Furthermore, our setting is simpler in that we only need to distinguish between exception from non-exception states, while other systems also identify exception types.

Taint analysis. Our system employs static taint analysis to track "bad" data in exceptional situations. Taint analysis, either static (e.g., [14, 10, 32, 1, 33]) or dynamic (e.g., [18, 34, 19]), has been successfully applied to preventing a range of attacks (e.g., format string attacks [25]). Static taint analysis does not incur runtime overhead as dynamic analysis does, but may report false errors. A promising hybrid strategy recently proposed by Chang *et al.* [3] performs static taint analysis as much as it can, but leaves difficult cases for dynamic analysis. One technical difference between our static taint analysis and previous ones is that it depends on a pointer graph. The pointer graph (described in Section 3.2) both approximates taint propagation and is used to cope with aliases; previous systems [14, 3] have separate taint propagation and pointer-analysis modules.

3. STATIC ANALYSIS

Figure 4 presents pseudo code that highlights the major steps of our static analysis framework. First, it performs exception analysis on an input program. Second, 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 recomputes exception analysis. We next discuss the three components in detail.

3.1 Exception analysis

To infer the exception state at every program location, we started with a standard interprocedural dataflow analysis framework with a lattice whose elements are in $2^{\{\texttt{Exn},\texttt{NoExn}\}}$. In this lattice, `Exn` and `NoExn` are the non-error states in the FSM of Figure 3.

Input: Program P
Output: A list of warning locations in P
Notation: a) op_l stands for the operation at location l in P
b) $\text{AS}_\circ(l)$ stands for the abstract state at the entry of op_l

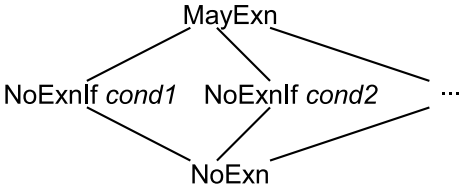
BEGIN: $\text{AS}_\circ$ = exceptionAnalysis(P);
for each location l in P **do**
  if ($\text{AS}_\circ(l)$ implies a possible exception) and (unSafe(op_l)) **then**
    Issue a warning for location l;
    P=warningRecovery(P);
    goto BEGIN
  end if
end for

Figure 4: High-level steps of static analysis

It was quickly discovered, however, this lattice was insufficient because JNI programs often exploit correlation between exception states and execution states. Our first system was imprecise because of not tracking the correlation. For instance, many JNI functions return an error value and at the same time throw an exception to signal failures (similar situation can be found in Python/C APIs). As a result of this correlation between the return value and the exception state, 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. The following program shows such an example involving `GetIntArrayElements`, which returns the null value and throws an exception when it fails:

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

To correlate exception and execution states, our idea is to have a more refined lattice that have extra elements that carry *semantic conditions*. In particular, we use a dataflow lattice described below:



The following table summarizes the semantics of the elements in the lattice:

`MayExn`	Some exceptions may be pending
`NoExnIf` *cond*	No exceptions pending if *cond* is true
`NoExn`	No exceptions pending

The abstract element "`NoExnIf` *cond*" in the lattice captures correlation between exception and execution states. For the above example, the abstract state after line 1 will be "`NoExnIf` $p \neq 0$", reflecting that when $p \neq 0$, then there is

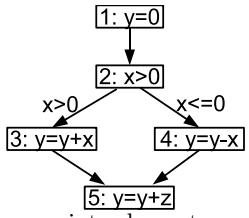
no exception. Given this and taking the following **if** statement into account, our system can derive the state before line 4 is `NoExn`.

For computational efficiency, our lattice is of finite height. The result of joining two "`NoExnIf` *cond*" elements is `MayExn`, when the two conditions are different. Another note is that our lattice is specially designed for inferring no-exception states and does not have an element that corresponds to must-exception states. One design objective of our system is *soundness*, that is, it should detect all true errors. If the abstract state at a location is `NoExn`, it is certain that the location does not constitute an error of mishandling JNI exceptions according to the definition in section 1.2. Therefore, tracking no-exception states accurately has the benefit of eliminating false positives, while maintaining soundness.

Having introduced the lattice, we next discuss various aspects of our interprocedural dataflow algorithm for inferring exception states.

Augmented control flow graphs. We augment a control flow graph with boolean guards on edges. A boolean guard is a condition introduced by, for example, an if statement. Below we give an example program and its augmented control flow graph.

```
1  y = 0;
2  if (x>0)
3    y = y + x;
4  else y = y - x;
5  y = y + z;
```



The if statement in the program introduces two guards: "$x > 0$" for the true branch and "$x <= 0$" for the false branch. We actually associate with every edge in the graph a boolean guard. The ones without explicit guards have implicit always-true guards.

Guards help derive no-exception states. For example, assuming the abstract state at location 2 of the above example is "`NoExnIf` $x \neq 0$", we can then derive at location 3 there is no exception because the guard $x > 0$ on the edge $(2, 3)$ is a stronger condition than $x \neq 0$. This reasoning process is formally captured by the following function, which takes an input state s and a guard g and derives a new state:

$$\text{doGuard}(s, g) = \begin{cases} \texttt{NoExn} & \text{if } s ='' \texttt{NoExnIf } cond'' \text{ and } g \Rightarrow cond \\ s & \text{Otherwise} \end{cases}$$

where $\Rightarrow$ stands for the logical implication relation.

Deciding the validity of "$g \Rightarrow cond$" for general guards and conditions would require some form of constraint solving. Our implementation instead takes a simple but safe approach. In the implementation, conditions in "`NoExnIf` *cond*" are basic conditions of the form "$\text{op}(v, c)$", where op is a binary operator (including $=, \neq, >, <, \geq, \leq$), v is a variable name, and c is a constant. We also implemented a simple algorithm which conservatively decides whether a guard implies a condition. The simple form of conditions and the conservative algorithm are sufficient to suppress most false positives (demonstrated by experiments in Section 4).

Dataflow equations. Given an input program, we use the notation op_l for the operation at location l. We use $\text{AS}_\circ(l)$ and $\text{AS}_\bullet(l)$ for the abstract state at the entry and exit of the operation at location l, respectively.

Figure 5 presents dataflow equations for our exception analysis. The entry abstract state at l is the join of those exit abstract states with an edge ending at l, after consultation with guards. The consultation with guards is where the power of our exception analysis is—a "`NoExnIf` *cond*" may be refined to `NoExn`.

The exit abstract state at l is computed in four cases:

- If the operation is an invocation of a JNI function, then its effect is presented in Table 4 in the appendix. The table is an encoding of the semantics of JNI functions in terms of their effects on exception states [13]. For example, after `Throw`, the abstract state becomes `MayExn`, and after `ExceptionClear`, the state becomes `NoExn`.

- When the operation at location l is not an invocation of a JNI function, and the abstract state before is either `NoExn` or `MayExn`, then the abstract state keeps the same after the operation; the operation does not change runtime exception states.

- When the abstract state before is "`NoExnIf` *cond*", and the operation does not affect the condition, then the abstract state keeps the same. In the figure, we use the notation $\text{Inv}(cond, op_l)$ to denote that the condition is an invariant with respect to the operation. The relation $\text{Inv}(cond, op_l)$ is computed through a may-alias analysis, which can estimate what variables may be affected for a given operation. If *cond* does not depend on any of the variables affected, then it holds on the state after the operation and thus is an invariant with respect to the operation.

- When the abstract state before is "`NoExnIf` *cond*", and the operation *does* affect the condition, then the abstract state after becomes `MayExn`. For example, if the abstract state is "`NoExnIf` $x \neq 0$", and the next operation updates variable x, then there is in general no relationship between $x \neq 0$ and the exception state after the update. This is an example where an operation does not affect runtime exception states, but affects abstract states.

It can be easily verified that the transfer functions in our dataflow equations are monotone with respect to the dataflow lattice, satisfying the general requirement of dataflow analysis framework.

Inter-procedural analysis. We have implemented a standard worklist algorithm that takes an entire JNI program as input and computes abstract states at each program point [22]. To speed up the process of computing the fixed point, the worklist algorithm iterates in reverse post order of the reverse call graph of the program [20, Ch6.2]. For each function in the program, the algorithm obtains a summary that tells its ending state when given an input state. Our algorithm also performs simple inference to conclude "`NoExnIf` *cond*" ending states. For instance, the ending state for the following function is `NoExnIf` *ret* $\neq -1$, where we use *ret* for the return value.

```
...
if (...) {
  (*env)->Throw(...);
  return -1;
```

$$\mathrm{AS}_\circ(l) \;=\; \bigsqcup\{\, \mathrm{doGuard}(\mathrm{AS}_\bullet(l'), g) \mid (l', l) \text{ is a control-flow edge with guard } g \,\}$$

$$\mathrm{AS}_\bullet(l) \;=\; \begin{cases} \mathit{jnifun}(\mathrm{AS}_\circ(l)) & \text{if } op_l = \mathit{jnifun} \\ \mathrm{AS}_\circ(l) & \text{if } op_l \neq \mathit{jnifun}, \mathrm{AS}_\circ(l) = \texttt{NoExn} \text{ or } \texttt{MayExn} \\ \mathrm{AS}_\circ(l) & \text{if } op_l \neq \mathit{jnifun}, \mathrm{AS}_\circ(l) ='' \texttt{NoExnIf } \mathit{cond}'' \text{ and } \mathrm{Inv}(\mathit{cond}, op_l) \text{ holds} \\ \texttt{MayExn} & \text{if } op_l \neq \mathit{jnifun}, \mathrm{AS}_\circ(l) ='' \texttt{NoExnIf } \mathit{cond}'' \text{ and } \mathrm{Inv}(\mathit{cond}, op_l) \text{ does not hold} \end{cases}$$

Figure 5: Dataflow equations for exception analysis

```
} else {
  ... //No exceptions pending
  return 0;
}
```

3.2 Determining unsafe operations

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

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.

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

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

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 Figure 1 shows. Our framework 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 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 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 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 becomes also 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 Figure 4, 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 [16], 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 value 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. 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.

Figure 6 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 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 Figure 6 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 as their models, both in terms of taint source and taint propagation. For soundness, calling any external functions that take tainted pointers as parameters is considered being unsafe—they might dereference those pointers. This is a source of inaccuracy in our analysis.

Our algorithm for constructing pointer graphs handles most C features, including function call 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.

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

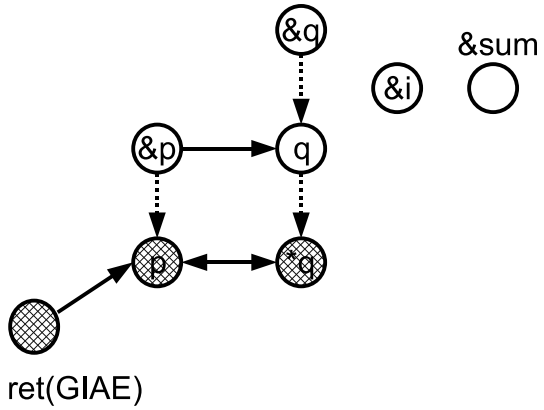


Figure 6: An example program and its pointer graph. The program takes a Java integer array and computes the sum of all positive elements.

3.3 Warning recovery

The purpose of warning recovery is to suppress duplicate warnings. To illustrate its necessity, we use the example in Figure 6. At both line 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 naive system would issue warnings for both line 6 and 7. However, both warnings are due to the error of mishandling 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 elements in our dataflow lattice. In particular, `MayExn` now becomes `MayExn`$\{ l_1, ..., l_n \}$. If the abstract state at a location l is `MayExn`$\{ l_1, ..., l_n \}$, it means that exceptions thrown at l_1 to l_n might reach l. We augment "`NoExnIf` *cond*" 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 recomputes dataflow 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 and semantically equivalent one.

4. EVALUATION

In this section, we present details of the implementation, experimental results and discussions.

4.1 Implementation and experiments

Our static analysis is implemented in the CIL framework [17], which is 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 about 3,700 lines of OCaml code, including 2,000 lines for constructing pointer graphs. In the exception analysis, we also used a may-alias analysis module and a call-graph module included in the CIL.

We started experiments by running our system on the JDK directories targeted by the previous empirical study [28]. It included 38K lines of C code. The results in that study were used to perform a sanity check on our system implementation. We then extended the experiments to also cover the `share/native/sun` directory in the JDK. In the end, our experiments checked approximately 260k lines of C code in the JDK. We also tested three other non-JDK programs: java-posix[7], java-gnome[8] , and JOGL[9] . All three programs are typical JNI programs.

4.2 Experimental results

Experimental results are presented in Table 1. For each program, the table lists the number of lines of C code, the total execution time (average over three runs), and the total number of warnings. For each warning, we manually determined whether it is a true error or a false positive. This process took several days to complete. The number of true errors and false-positive (FP) rates are also shown in the table. Experiments were carried out on a Linux Ubuntu (version 2.6.27-11-generic) box with Intel Core2 Duo CPU at 3.16 GHz.

Our system is efficient and scalable. For over 268,000 lines of code, it took about 20 seconds to examine all of them. The outputs contain warnings in terms of specific locations in the code. In the future we plan to add a more developer-friendly interface so that a programmer can rely on the tool to perform quick and accurate code review. We also expect that our tool can be easily applied to many other JNI programs with similar performance.

The overall false positive rate among all benchmark JNI programs is 15.4%. Our system achieves a relatively high precision, especially considering it strives to be sound (i.e., no false negatives). Of the 716 true errors, 689 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. We next discuss the cause of false positives.

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. This kind of false positives could be removed by either including the source code of the library functions, or by adding additional hand 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.

4.3 Warning recovery and static taint analysis

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 that is without warning recovery; we denote this version by V1. We also tested a version that is 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 38.6%, as compared to 15.4%. Further removing the component of static taint analysis would hike the FP rate to 70.5%. These experiments demonstrated that warning recovery and static taint analysis are very effective in terms of reducing the number of false positives.

Program Name	# of Warnings			
	V1	FP%	*V2*	FP%
JDK 6 directories	1072	38.2	2235	70.4
java-posix	66	45.5	108	66.7
java-gnome	24	45.8	83	84.3
java opengl	5	0	5	0
TOTAL	1167	38.6	2,431	70.5

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

4.4 Comparison with the previous study

As we have discussed, a previous study [28] carried out an empirical analysis on the security of native code in the JDK by using grep-based scripts. Table 3 shows the comparison between the results of our system and the previous study on the set of JDK directories covered in the previous study. Our new system not only included all of the errors that were found in the previous study, but also discovered more true errors. We do like to note, however, that in deciding whether a warning is a true error, 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, our system considers 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 true errors shown in the table (two thirds of the difference are due to this). 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 errors.

Program Name	Approx # lines of code	Execution Time (seconds)	Warnings	True Errors	FP%
JDK 6 directories	260,000	15.29	770	662	13.9
java-posix	1,800	0.16	50	36	28.0
java-gnome	5,800	5.64	21	13	38.1
java opengl	700	0.04	5	5	0.0
TOTAL	268,300	21.13	846	716	15.4

Table 1: Summary of experimental results

Results	The previous study	This work
Warnings	567	132
True Errors	35	93
FP %	93.7	29.5

Table 3: Comparison with the previous study [28] (Of the 35 errors in the previous study, 11 are due to explicit throws and 24 due to implicit throws)

5. FUTURE WORK

First, our framework can be easily applied to analyzing other foreign function interfaces (FFIs). The parts of the system that are JNI specific consists of the transfer functions and specification of taint sources, both of which can be straightforwardly adapted to analyze other FFIs, such as the Python/C and the OCaml/C interface.

Second, this work aims to capture 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 [31] 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 an interesting future work is to enforce FSM-based resource usage protocols statically.

6. CONCLUSION

Native components pose a significant security threat to managed environments such as the JVM. Given the increasing popularity of programming languages with managed environments, it is necessary to design mediation techniques to address the threat. Making a solid step toward this goal, we have designed and implemented a novel static analysis framework for finding bugs in exceptional situations in JNI programs. Our experimental results demonstrated the effectiveness of our techniques. They are applicable to other environments such as Python, OCaml, and .NET.

Acknowledgments

We would like to thank Martin Hirzel, Kathryn McKinley, and anonymous reviewers for their many helpful comments.

7. REFERENCES

[1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy (S&P)*, pages 143–159, Washington, DC, USA, 2002. IEEE Computer Society.

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

[3] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 39–50, 2008.

[4] H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244, 2002.

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

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

[7] S. D. Gathman. java-posix. http://www.bmsi.com/java/posix/package.html. Fetched on August 7, 2009.

[8] The java-gnome user interface library. http://java-gnome.sourceforge.net/. Fetched on August 7, 2009.

[9] JOGL API project. https://jogl.dev.java.net/. Fetched on August 7, 2009.

[10] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy (S&P)*, pages 258–263, 2006.

[11] G. Kondoh and T. Onodera. Finding bugs in Java Native Interface programs. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 109–118, New York, NY, USA, 2008. ACM.

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

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

[14] B. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *14th Usenix Security Symposium*, pages 271–286, 2005.

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

[16] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, 2005.

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

[18] J. Newsome and 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.

[19] A. Nguyen-tuong, S. Guarnieri, D. Greene, and D. Evans. Automatically hardening web applications using precise tainting. In *In 20th IFIP International Information Security Conference*, pages 372–382, 2005.

[20] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag Berlin, 1999.

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

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

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

[24] M. Schoenefeld. Denial-of-service holes in JDK 1.3.1 and 1.4.1_01. Retrieved Apr 26th, 2008, from `http://www.illegalaccess.org/java/ZipBugs.php`, 2003.

[25] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *In Proceedings of the 10th USENIX Security Symposium*, pages 201–220, 2001.

[26] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

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

[28] G. Tan and J. Croft. An empirical security study of the native code in the JDK. In *17th Usenix Security Symposium*, pages 365–377, 2008.

[29] US-CERT. Vulnerability note VU#138545: Java Runtime Environment image parsing code buffer overflow vulnerability, June 2007. Credit goes to Chris Evans.

[30] US-CERT. Vulnerability note VU#939609: Sun Java JRE vulnerable to arbitrary code execution via an unspecified error, Jan. 2007. Credit goes to Chris Evans.

[31] W. Weimer and G. Necula. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems*, 30(2):1–51, 2008.

[32] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *15th Usenix Security Symposium*, pages 179–192, Berkeley, CA, USA, 2006. USENIX Association.

[33] W. Xinran, J. Yoon-Chan, Z. Sencun, and L. Peng. Still: Exploit code detection via static taint and initialization analyses. In *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 289–298, Washington, DC, USA, 2008. IEEE Computer Society.

[34] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *15th Usenix Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

APPENDIX

A. WHITELIST

- A list of JNI functions that are safe to call with a pending exception (specified in the JNI Manual [13]): ExceptionOccurred, ExceptionDescribe, ExceptionClear, ExceptionCheck, ReleaseStringChars, ReleaseStringUTFchars, ReleaseStringCritical, Release⟨Type⟩ArrayElements, ReleasePrimitiveArrayCritical, DeleteLocalRef, DeleteGlobalRef, DeleteWeakGlobalRef, MonitorExit, PushLocalFrame, PopLocalFrame.
- The return operation and memory free operation.

jnifun =	*jnifun*(s) =	PRE-CONDITION	EXPLANATION
Throw, ThrowNew	`MayExn`	s = `NoExn`	A second exception cannot be thrown when one is already pending.
ExceptionOccurred	s		Return a reference to the pending exception, or null if there is no pending exception.
ExceptionCheck	s		Return false if there is no pending exception; otherwise, true.
ExceptionDescribe	`NoExn`		If an exception is pending, print a diagnostic message and *clear* it. Otherwise, no effect.
ExceptionClear	`NoExn`		Clear the pending exception if there is one. No effect otherwise.
Call⟨Type⟩Method	`MayExn`	s = `NoExn`	Call a Java method, which may throw exceptions. The same applies to Call⟨Type⟩MethodA, Call⟨Type⟩MethodV, CallNonvirtual⟨Type⟩Method, CallNonvirtual⟨Type⟩MethodA, CallNonvirtual⟨Type⟩MethodV, CallStatic⟨Type⟩Method, CallStatic⟨Type⟩MethodA, CallStatic⟨Type⟩MethodV, Set⟨Type⟩ArrayRegion, SetObjectArrayElement
AllocObject	`NoExnIf` ($ret \neq 0$)	s = `NoExn`	Return 0 on failure. The same applies to DefineClass, FindClass, FromReflectedField, FromReflectedMethod, ToReflectedField, ToReflectedMethod, Get⟨Type⟩ArrayElements, Get⟨Type⟩ArrayRegion, GetObjectArrayElement, GetPrimitiveArrayCritical, GetFieldID, GetMethodID, GetStaticFieldID, GetStaticMethodID, GetStringChars, GetStringCritical, GetStringRegion, GetStringUTFChars, GetStringUTFRegion, NewObject, NewObjectA, NewObjectV, NewObjectArray, New⟨Type⟩Array, NewString, NewStringUTF, NewWeakGlobalRef.
MonitorEnter	`NoExnIf` ($ret = 0$)	s = `NoExn`	Return 0 on success. The same applies to EnsureLocalCapacity, RegisterNatives, NewDirectByteBuffer.
MonitorExit	`NoExnIf` ($ret = 0$)		Return 0 on success. The same applies to PushLocalFrame, PopLocalFrame.

Table 4: Transfer functions for the JNI functions that may affect the exception state. The state before invocation of a JNI function is denoted by s, and the state after is *jnifun*(s). When a precondition is not satisfied, a warning is issued.