Finding Bugs in Exceptional Situations of JNI Programs

Siliang Li    Gang Tan

Lehigh University

Wednesday, November 10, 2009
@ ACM CCS ’09 Chicago, IL
What is the JNI?

- JNI stands for Java Native Interface
  - The interface between Java and native C/C++
- JNI Programs are software written using the JNI
An Example JNI Program

```java
class HelloWorld {
    private native void print();
    public static void main(String[] args) {
        new HelloWorld().print();
    }
    static {
        System.loadLibrary("HelloWorld");
    }
}
```

```c
#include <jni.h>
#include <stdio.h>
#include "HelloWorld.h"
JNIEXPORT void JNICALL Java_HelloWorld_print(JNIEnv *env, jobject obj) {
    printf("Hello World!\n");
    return;
}
```

Java code

C code
Why Use the JNI?

- Many reasons
  - Legacy code, e.g., zlib
  - Efficiency
  - ...
- The JNI is used in many Java software applications
  - E.g., OpenGL, Java-Gnome, Java-Posix, ...

Over the years, the size of C/C++ code has been on the increase with each JDK release.
Native Code is Security Critical

- Native components: the security dark corner
  - Relevant security policies cannot be enforced on them
- Any vulnerability in trust native code can compromise the security of the JVM.
  - Furr & Foster [ESOP ’06]: 155 bugs
  - Kondoh & Onodera [ISSTA ’08]: 87 bugs
  - Tan & Croft [Security ’08]: 126 bugs
  - Other bug reports in Sun’s Vulnerability database (US-CERT. VU#138545, ...)
Bug Finding in the JNI

- A recent empirical study (Security ’08) finds there are 59 security critical bugs in just 38K lines of JDK6’s C code.
- Among them, 35 bugs are due to mishandling of exceptions in native code.
- This is because exception handling is quite different between Java and native code.
Java Exceptions

When an exception is thrown

- The JVM transfers the control to the nearest enclosing catch statement

```java
try {
    if (checkFails()) {
        throw …;
    }
    doDangerousOp();
} catch (Exception e) {
    ...
}
```

The dangerous operation skipped
However, JNI Exceptions are Different

- The JNI exception won’t be thrown until the C method returns

```java
public class A {
    public native void c_fun();
    void j_fun () {
        c_fun();
    }
}
```

```c
void c_fun (…) {
    if (checkFails()) {
        Throw(…); return;
    }
    doDangerousOp();
}
```

The dangerous operation still executed!
Things become more complicated when function calls are involved

```c
void c_fun (...) {
    util_fun(); // Might throw a JNI exception
    if (ExceptionOccurred()) {...; return;}
    {...};
}
```

C code
Definition of a Bug

- We are looking for bugs in software written using the JNI
- Specifically, we are looking for bugs that match the following pattern:
  - A JNI Exception is pending AND
  - The next operation is a dangerous operation
Static Analysis for Finding Bugs in Exceptional Situations

- We propose a framework that looks for this type of bugs in JNI programs
  - By finding exceptional situations AND
  - By identifying dangerous operations
- Our work uses static analysis, mainly because:
  - Bugs in exceptional situations are hard to trigger using user inputs
  - Necessary for dynamic analysis
  - Static analysis reasons about all user inputs
Outline

- Background
- Proposed Framework
  - Exception Analysis
  - Whitelisting + Static Taint Analysis
    - To determine dangerous operations
  - Warning Recovering
    - To eliminate duplicate warnings
- Experiments and Results
- Related Work
- Future Work
- Conclusion
Overview of the Framework

<table>
<thead>
<tr>
<th>Determining Dangerous Ops</th>
<th>Eliminating Dup. Warnings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Taint Analysis</td>
<td>Warning Recovering</td>
</tr>
<tr>
<td>Whitelisting</td>
<td></td>
</tr>
</tbody>
</table>

Exception Analysis
Exception Analysis

- Goal:
  - We want to capture places in a JNI program where exceptions are pending
  - The framework employs a standard lattice augmented with semantic conditions
The Lattice with Conditions

NoExnIf $cond_1$  NoExnIf $cond_2$  ...  NoExn

<table>
<thead>
<tr>
<th>NoExn</th>
<th>No exceptions pending</th>
</tr>
</thead>
<tbody>
<tr>
<td>MayExn</td>
<td>Some exceptions may be pending</td>
</tr>
<tr>
<td>NoExnIf $cond$</td>
<td>No exceptions pending if $cond$ is true</td>
</tr>
</tbody>
</table>
An Example of Exception Analysis

\[ p = (*env) \rightarrow \text{GetIntArrayElements(env, arr, Null);} \]

if \((p == \text{NULL})\) /* exception thrown*/
   return;

\[ \text{NoExn} \]
\[ \text{NoExnIf} \]
\[ p!=\text{NULL} \]
\[ \text{MayExn} \]
\[ \text{NoExn} \]

GetIntArrayElements may throw OutOfMemoryError exception
Exception Analysis

- Inter-procedural dataflow analysis
- A simple algorithm that infers exception states after guards (e.g., introduced by `if` statements)
- At the end of the analysis, we obtain exception states for every program point
Determining Dangerous Operations

- Strategy
  - Whitelisting + Static Taint Analysis
- Whitelisting
  - A list of absolutely safe operations when exceptions are pending
    - E.g., ExceptionClear
  - The rest are considered dangerous
- Advantage: simple and low false negative
Static Taint Analysis: Overview

☐ Motivation
  ■ Whitelisting only looks at what an operation is and this leads to many false positives

☐ Intuition
  ■ If the operation does not involve any bad data, the operation should be regarded safe
An Example of False Warning by Whitelisting

- The following code should be safe but a warning would be issued if using whitelisting alone.

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

NoExnIf

\( p \neq NULL \)

False Warning
Motivation for Static Taint Analysis

- We observe from the previous example that
  - Even though $p$ might be dangerous if dereferenced, but it is not used
  - We cannot put ‘+’ and ‘=’ operations on the whitelist, because that would allow statement 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.
Static Taint Analysis: Overview

- Two ideas:
  - Use taint source to model where faults may happen
  - Use static analysis to track fault propagation
- Constructs a pointer graph
  - Nodes -- value of pointers
  - Edges -- how pointer values flows in the graph
The Pointer Graph for Static Taint Analysis

- Two types of Nodes
  - Taint Source
    - Used to model where faults may happen (e.g. a JNI function that may throw exceptions)
    - Where taint is generated (e.g. user input)
  - Taint Sink
    - Dangerous ways in which tainted data may be used in the program (e.g. dereferencing operation)
The Pointer Graph for Static Taint Analysis

- Two types of Edges
  - Flow-to edge
  - Contains edge

- Special edges
  - Bidirectional
  - Propagate flow-to edges along contains edges

- We use static analysis to model taint propagation
  - How taint is propagated in the program

- Requires user defined (customizable) taint source
  - E.g., user file input, incoming network packet, C and JNI functions
An Example Program and its Pointer Graph

```c
int *p;
int **q = &p;
p = (*env) → GetIntArrayElement(env, arr, NULL);

int len = (*env) → GetArrayLength(env, arr);

for(i=0; i<len; i++){
    if((*q)[i]>0){
        sum += (*q)[i];
    }
}
```

Warning!!

return(GetIntArrayElement)
Eliminating Duplicate Warnings: Warning Recovering

- Motivation
  - False positives caused by warning propagation that originates from the same pending exception

- Intuition
  - We should correct the warning when it first occurs

- Technique
  - Record exception state with respect to its location in the code
  - Insert ExceptionClear immediately after a warning has been issued
An Example of Warning Recovering

```c
p = (*env) → GetIntArrayElements(env, arr, NULL);

(*env) → ExceptionClear();
int len = (*env) → GetArrayLength(env, arr);
for (i=0; i<len; i++) {
    if((*q)[i] > 0 {
        sum += (*q)[i];
    }
}
```

NoExnIf
p!=NULL

Warning

Warning
Suppressed!
Experiment Setup

- The framework is implemented in
  - CIL – a tool set for analyzing and transforming C programs
  - we primarily focused on examining C code
  - 3700 lines of OCaml code
- Consulted findings in the recent empirical study published in Security ‘08 as ground truth
- Conducted manual inspection of reported bugs
Experimental Results

- **Strategy:**  
  Whitelisting + Static Taint Analysis + Warning Recovering

- Relative low false positive rate

- Efficient over large amount of code
Experimental Results

<table>
<thead>
<tr>
<th>Program Name</th>
<th># of Warnings</th>
<th>V1</th>
<th>FP%</th>
<th>V2</th>
<th>FP%</th>
</tr>
</thead>
<tbody>
<tr>
<td>JDK 6 directories</td>
<td>1072</td>
<td>38.2</td>
<td></td>
<td>2235</td>
<td>70.4</td>
</tr>
<tr>
<td>java-posix</td>
<td>66</td>
<td>45.5</td>
<td></td>
<td>108</td>
<td>66.7</td>
</tr>
<tr>
<td>java-gnome</td>
<td>24</td>
<td>45.8</td>
<td></td>
<td>83</td>
<td>84.3</td>
</tr>
<tr>
<td>java.opengl</td>
<td>5</td>
<td>0</td>
<td></td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>1167</strong></td>
<td><strong>38.6</strong></td>
<td></td>
<td><strong>2,431</strong></td>
<td><strong>70.5</strong></td>
</tr>
</tbody>
</table>

- **V1**: Whitelisting + Static Taint Analysis
- **V2**: Whitelisting
- False positive rates in both schemes are lot higher (than 15%)
Related Work

- Checking temporal safety properties
  - Interprocedural dataflow analysis by Reps et al. (*POPL ’95*)
  - Model checking (e.g. Chen and Wagner *CCS’02*)
  - Typestate analysis by Strom and Yemini (*IEEE Trans. on SE ‘86*)
  - Property simulation by Das et al. (*PLDI’02*)

- JNI bug finding
  - JSaffire by Furr and Foster (*ESOP’06*)
  - A bug finder by Kondoh and Onodera (*ISSTA’08*)
  - SafeJNI by Tan et al. (*ISSSE’06*)
Related Work (continued)

- Exception Analysis
  - “Jex tool” by Robillard and Murphy (ACM Trans. on PLS ’03)
  - Exception specification by Malayeri and Aldrich (Lecture Notes in Computer Science ‘06)
  - Java exception analysis by Chang et al (SAC’01)

- Taint analysis
  - Static taint analysis (e.g. Jovanovic et al. S&P’06)
  - Dynamic taint analysis (e.g. Newsome and Song NDSS’05)
  - Hybrid model (e.g. Chang et al. SAC’01)
  - Pointer graph (e.g. Livshits and Lam Security ‘05)
Future Work

☐ To apply the framework to other Foreign Function Interfaces (FFIs)
   ■ E.g., Python/C, OCaml/C

☐ To enforce proper resource release when exceptions are pending in native code

☐ To derive specific types of JNI exceptions
Conclusion

- We introduced a framework that uses static analysis to find bugs caused by mishandling of exceptional situations in JNI programs
  - Efficient and relative low false positive rates
- Use static analysis is an effective technique for bug finding in multi-lingual software
Thank you!

Questions?