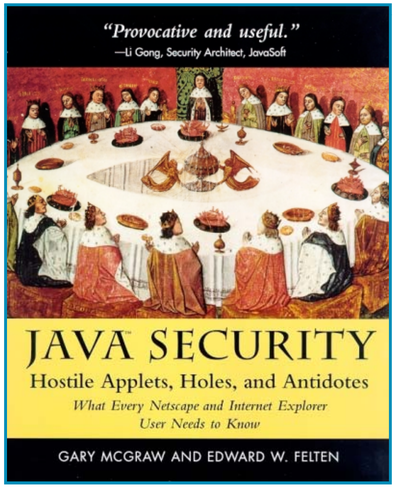
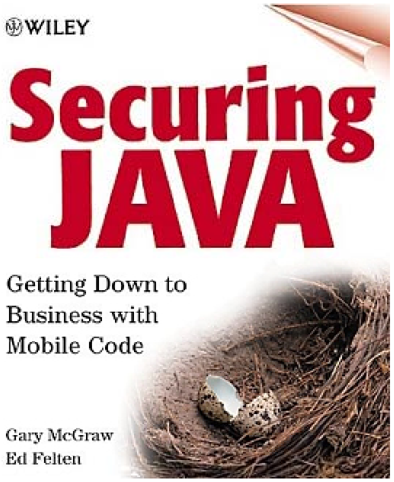
An Empirical Security Study of the Native Code in the JDK

Gang Tan, Boston College $\Rightarrow$ Lehigh University
Jason Croft, Boston College

Java Security

- Various holes identified and fixed
  - [Dean, Felten, Wallach 96]; [McGraw & Felten 99]; [Saraswat 97]; [Liang & Bracha 99]; …
- Formal models of various aspects of Java
  - Stack inspection [Wallach & Felton 98]
  - JVML model [Freund & Mitchell 03] [Stata & Abadi 99]
  - …
- Machine-checked theorems and proofs [Klein & Nipkow 06]





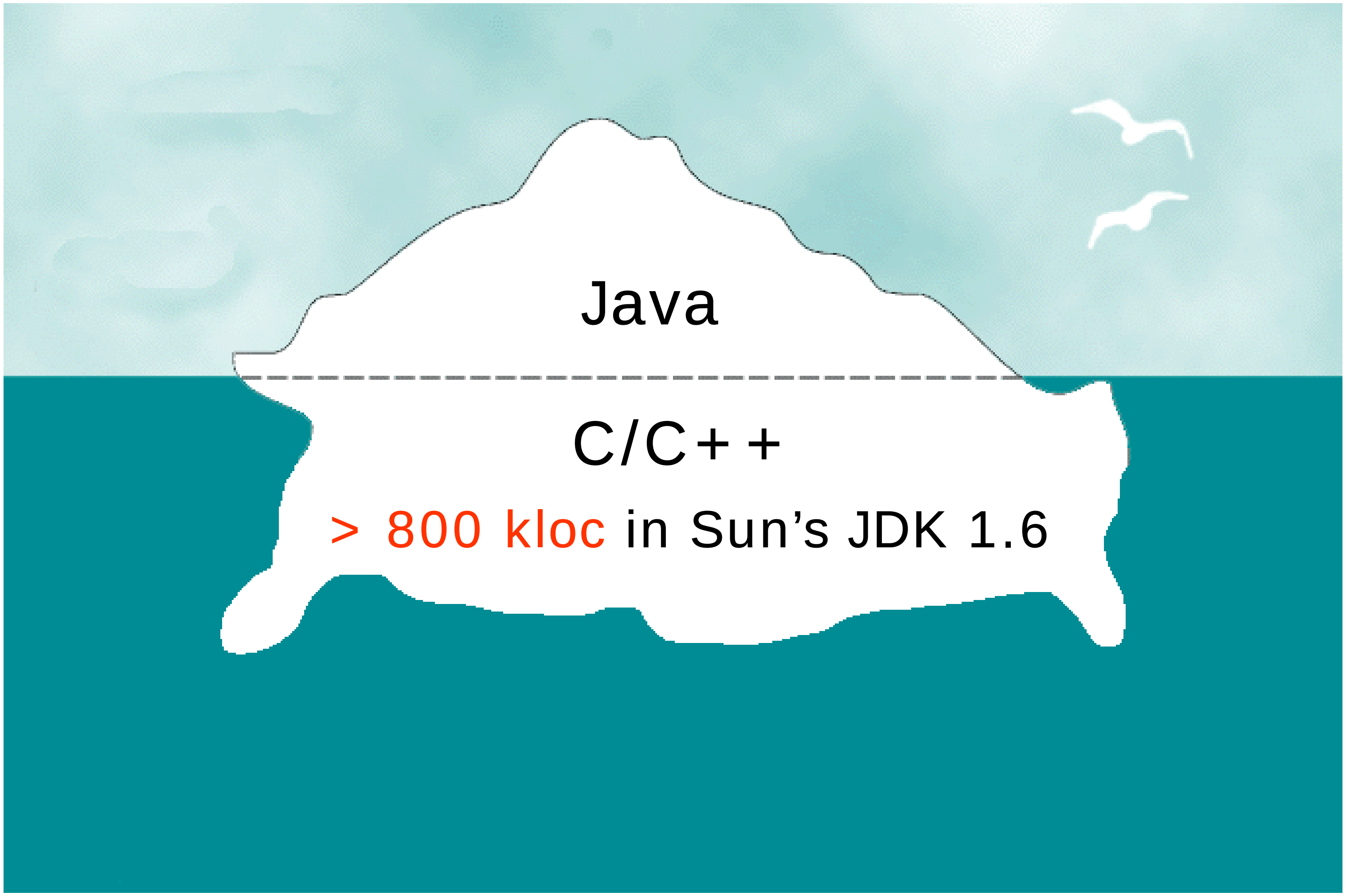
What About Native Methods?

- A Java class can have native methods
  - Implemented in C/C++
  - Interact with Java through the Java Native Interface (JNI)
- Outside of the Java security model
  - No type safety
  - Outside of the Java sandbox
- By default, Java applets does not allow loading non-local native code

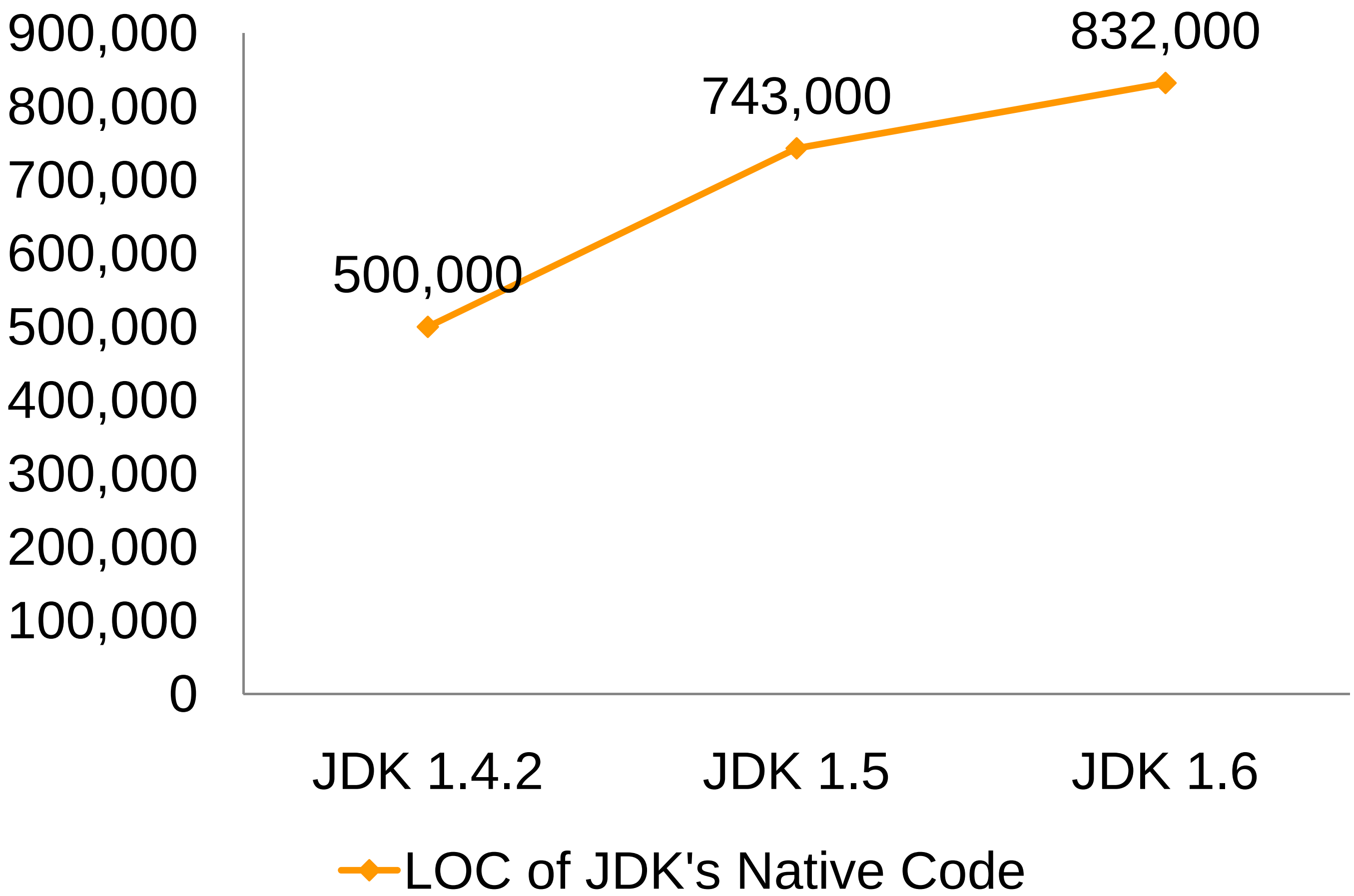
What About the Native code in the Java Development Kit (JDK)?

- java.io.FileInputStream
  - A Java wrapper for C code that invokes system libraries
- java.util.zip.*
  - Java wrappers that invoke the Zlib C compression/decompression library
- The JDK's native code is trusted by default

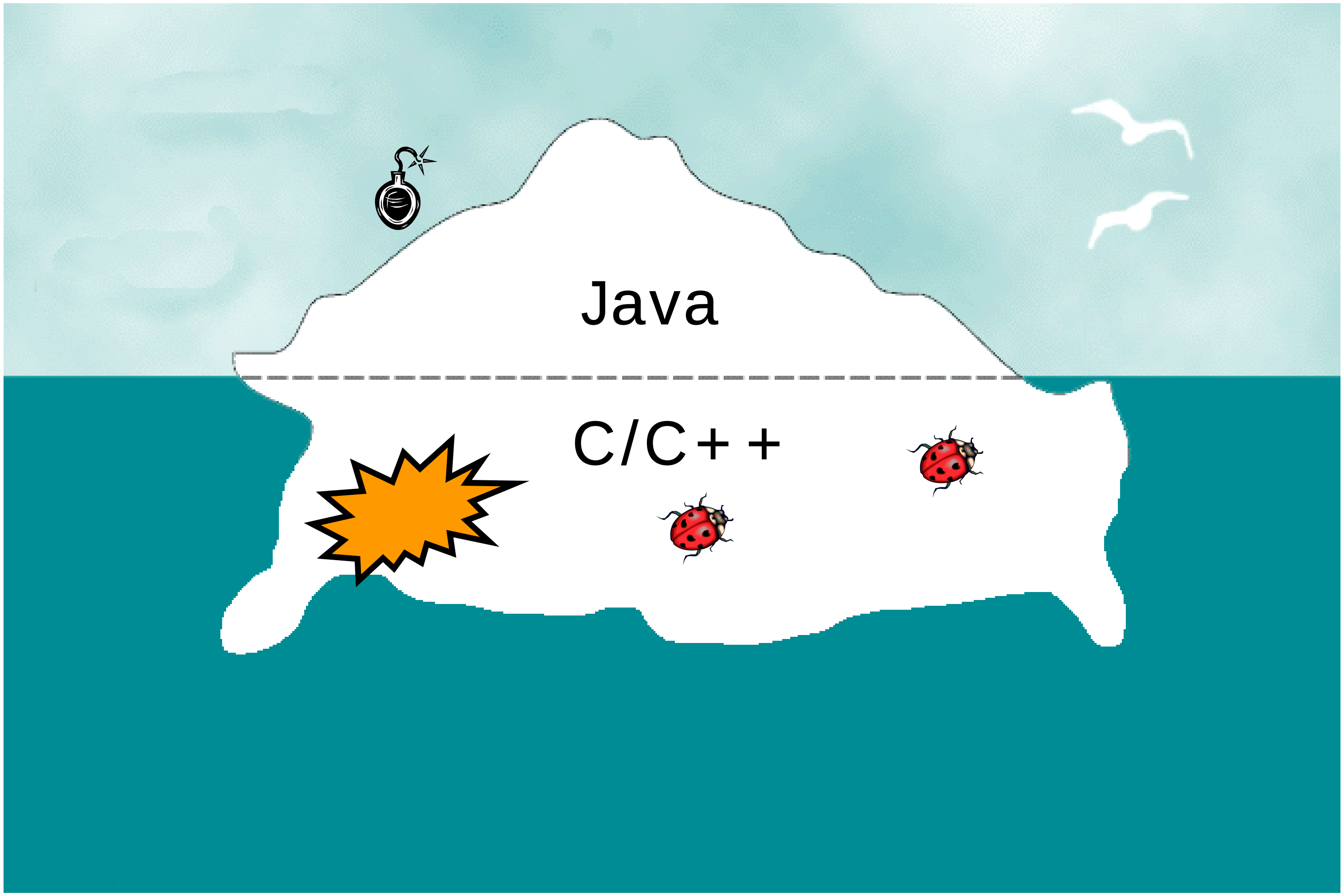
How Large Is This Trust?



The JDK's Native Code: On the Increase



Triggering a Bug in the Native Code



An Obvious Example

```
class Vulnerable {
  public native void bcopy(byte[] arr);
  …
}
```

Java code

```
void Java_Vulnerable_bcopy (…, jobject jarr) {
  char buffer[512];
  jbyte *arr = GetByteArrayElements(jarr, 0);
  strcpy(buffer, arr);
}
```

C code

Unbounded string copy!

An Empirical Security Study

- Folklore: bugs in the JDK's native code is a threat to Java security
  - All 800,000 lines are too big to be trusted
- Problem: how to alleviate the threat?
- An empirical study is a first and important step
- Goals of the study:
  - Collect evidence that the native code is a realistic threat to Java security
  - Collect data to understand the extent
  - Characterize bug patterns

Approach to Characterizing Bug Patterns

- Static analysis tools + manual inspection
  - Common C vulnerabilities
    - Splint, ITS4, Flawfinder
  - Bug patterns particular to the JNI
    - Custom built scanners: grep-based scripts; CIL-based scanners
    - Bug patterns inferred from the JNI manual
  - Manual inspection to rule out false positives
    - An HTML interface for browsing the code: GNU Global source code tag system; htags

Approach and Scope of the Study

- Pros
  - Can cover many bug patterns
  - The scanning results are fairly complete: good for collecting empirical evidence
- Cons
  - Lots of manual work: cannot cover all 800,000 lines
- Limiting the scope: target directories
  - Native code under share/native/java and solaris/native/java
  - They implement the native methods of the classes under java.*
  - 38,000 LOC of C code

A Taxonomy of Bugs in the Native Code of the JDK

A Summary of the Bugs Identified

	Bugs	Security Critical	Tools used
Mishandling JNI exceptions	11	Y	grep-based scripts
C pointers as Java integers	38	N	Our CIL scanner
Race conditions in file accesses	3	Y	ITS4, Flawfinder
Buffer Overflows	5*	Y	Splint, ITS4, Flawfinder
Mem. Management Flaws	29	N	Splint, grep-based scripts
Insufficient error checking	40	Y	Splint, grep-based scripts
TOTAL	**126**	**59**	

Java Exceptions

```
try {
  if checkFails() {
    throw …;
  }
  doSensitiveOp();
} catch (Exception e) {
    …
}
```

The sensitive operation skipped

Java code

- When an exception is thrown
  - The JVM transfers the control to the nearest enclosing catch statement

JNI Exceptions Are Different!

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

Java code

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

C code

The sensitive operation still executed!

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

Mishandling JNI Exceptions

- Things become more complicated when function calls are involved

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

C code

- Our study found 11 cases of mishandling JNI Exceptions
  - Mismatch between the programming models
  - Blame the programmers or the API designers

Another Bug Pattern: C Pointers as Java Integers

- Often, need to store C pointers at the Java side
  - However, how to declare the types of the C pointers in Java?
- Commonly used pattern
  - Cast the C pointers to Java integers
  - When passed back to C, they are cast back to pointers
- Example:
  - Zlib maintains a z_stream struct for keeping state info
  - A Java Deflater object needs to store a pointer to this C struct

Bogus Pointers to C

- The pattern is unsafe if the Java side can inject arbitrary integers to C
- Example [Greenfieldboyce & Foster]: GTK

```
class GUILib {
  public native static void setFocus (int windowPtr);
  ...
}
```

  - A public method that anybody can invoke with bogus pointers
- Some cases will enable reading/writing arbitrary memory locations

Bogus Pointers to C in the JDK

- The target directories in the JDK
  - 38 native methods that accept Java integers and cast them to pointers
  - Not security critical: they are declared as private
  - Attackers cannot invoke private methods, without Java Reflection

Still type safe

- Should still be fixed
  - Java Reflection: can invoke private methods
  - Java Reflection + C pointers as Java integers: read/write arbitrary memory locations

Type unsafe!

A Summary of Bug Patterns

- We found a range of bugs: buffer overflows, misusing JNI exceptions, …
  - O(100) bugs in 38 kloc code
- Other bug patterns (we did not find violations)
  - Type misuses
  - Deadlocks
  - Violating the Java sandbox security model

Remedies, Limitations, and Future Directions

Remedy: Static Analysis

- Find and remove bugs
- The static tools used in the study do not scale
  - High proportions of false positives (FP)

Off-the-shelf tools	FP rates
ITS4 -c1	97.5%
Flawfinder	98.3%
Splint	99.8%

  - Same story for our own scripts and scanners
  - A large amount of time on manual inspection
    - Prone to human errors

Reducing False Positives

- Advanced static analysis techniques can help
  - Software model checking; abstract interpretation; type qualifiers; theorem proving techniques
- Mishandling JNI exceptions: dataflow analysis
  - How many more bugs can we expect to find?
    - 11 violations out of 337 Throws
    - 2471 Throws => $\approx$ 80 violations

Reducing False Positives: Inter-Language Analysis

- During our manual inspection, we often went back and forth between Java and C side to decide if a warning is a bug

```
jint deflatebytes(…, jarray b, jint len, jint off) {
  …
  out_buf = (jbyte *) malloc (len);
  …
  SetByteArrayRegion(b, off, len, out_buf)
  …
}
```

No range checks on len and off!

Is this a buffer overrun?

Well, it depends on how the Java side invokes it

Static Analysis on Multi-Lingual Applications

- Most existing source-code analysis tools are limited a priori to code written in a single language
- Extending the horizon of analysis
  - Saffire [Furr & Foster, PLDI ’05, ESOP ‘06]
  - APLT [Zhang et al., ISSTA ’06]
  - ILEA [Tan & Morrisett, OOPSLA ’07]
    - Enable Java analysis to also understand the behavior of C code

Remedy: Dynamic Mechanisms

- SafeJNI [Tan *et al.* ISSSE '06]: dynamic checks + static pointer type system
  - Statically reject or dynamically stop ill-behaved C programs
  - Leverage CCured [Necula *et al.*] to provide internal memory safety to C code
  - Checkings at the boundary between Java and C
  - Performance slowdown: Microbenchmark: 14%-119%; Zlib: 74%
  - Limitations: concurrency; efficiency
- Assembly level monitoring: SFI, XFI

Remedy: Rewrite the Native Code in Safer Languages

- Java
- Cyclone
- Better interfaces between Java and C
  - Jeannie [Hirzel and Grimm OOPSLA ‘07]
  - Janet

In Summary

- Native code in the JDK is a time bomb to Java security
- In the short term
  - Develop scalable static analysis tools to eliminate bugs
  - Efficient dynamic mechanisms
- In the long term
  - Most of the C code should be converted into Java code---CLASSPATH's long term goal
- Same problem with .NET

The End