Static Program Analysis

Soundiness

Nanjing University

Yue Li

2020

1. Soundness and Soundiness
2. Hard Language Feature: Java Reflection
3. Hard Language Feature: Native Code

Soundness

Soundness:

Conservative approximation: the analysis captures all program behaviors, or the analysis result models all possible executions of the program

Soundness

Soundness:

Conservative approximation: the analysis captures all program behaviors, or the analysis result models all possible executions of the program

Can we achieve a fully sound analysis for real-world programs?

Academia

Virtually all published whole-program analyses are unsound when applied to real programming languages

Industries

Virtually all realistic whole-program static analysis tools (e.g., bug detection, security analysis, etc.) have to make unsound choices

Soundness

Soundness:

Conservative approximation: the analysis captures all program behaviors, or the analysis result models all possible executions of the program

Can we achieve a fully sound analysis for real-world programs?

Academia

Virtually all published whole-program analyses are unsound when applied to real programming languages

Why?

Industries

Virtually all realistic whole-program static analysis tools (e.g., bug detection, security analysis, etc.) have to make unsound choices

Hard Language Features for Static Analysis

- Java

  Reflection, native code, dynamic class loading, etc.

- JavaScript

  eval, document object model (DOM), etc.

- C/C++

  Pointer arithmetic, function pointers, etc.

Hard-to-analyze features: an aggressively conservative treatment to these features will likely make the analysis too imprecise to scale, rendering the analysis useless

… As a Result

- Generally, a claimed sound static analysis has a sound core in its implementation, i.e., most language features are over-approximated while some specific and/or hard ones are under-approximated
- Treatments to hard language features are usually omitted or only mentioned in an off-hand manner in some impl/eval parts in papers
- Not handling certain hard language features properly, e.g., Java reflection, may have a profound impact on analysis results

… As a Result

- Generally, a claimed sound static analysis has a sound core in its implementation, i.e., most language features are over-approximated while some specific and/or hard ones are under-approximated
- Treatments to hard language features are usually omitted or only mentioned in an off-hand manner in some impl/eval parts in papers
- Not handling certain hard language features properly, e.g., Java reflection, may have a profound impact on analysis results

Then claiming soundness in papers may mislead readers:

- For non-experts, they may erroneously conclude that the analysis is sound and confidently rely on the analysis results
- For experts, it is still hard for them to interpret the analysis results (how sound, fast, precise is the analysis) without a clear explanation about how they treat those important and hard language features

Soundiness

In Defense of Soundiness: A Manifesto

Ben Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Sam Guyer, Uday Khedker, Anders Møller, and Dimitrios Vardoulakis
Microsoft Research, Samsung Research America, University of Athens, University of Waterloo, University of Alberta, University of Colorado Boulder, Tufts University, IIT Bombay, Aarhus University, Google

COMMUNICATIONS OF THE ACM

2015

Soundiness

Truthiness: a "truth" that sb. believes to be true intuitively, without any fact or evidence

In Defense of Soundiness: A Manifesto

Ben Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Sam Guyer, Uday Khedker, Anders Møller, and Dimitrios Vardoulakis

Microsoft Research, Samsung Research America, University of Athens, University of Waterloo, University of Alberta, University of Colorado Boulder, Tufts University, IIT Bombay, Aarhus University, Google

COMMUNICATIONS OF THE ACM

2015

Soundiness

Truthiness: a "truth" that sb. believes to be true intuitively, without any fact or evidence

In Defense of Soundiness: A Manifesto

Ben Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Sam Guyer, Uday Khedker, Anders Møller, and Dimitrios Vardoulakis

Microsoft Research, Samsung Research America, University of Athens, University of Waterloo, University of Alberta, University of Colorado Boulder, Tufts University, IIT Bombay, Aarhus University, Google

COMMUNICATIONS OF THE ACM

2015

A soundy analysis typically means that the analysis is mostly sound, with well-identified unsound treatments to hard/specific language features

Soundness, Soundiness and Unsoundness

- A **sound** analysis requires to capture all dynamic behaviors
- A **soundy** analysis aims to capture all dynamic behaviors with certain hard language features unsoundly handled within reason
- An **unsound** analysis deliberately ignores certain behaviors in its design for better efficiency, precision or accessibility

Why hard language features are hard to analyze?

- Java Reflection
- Native Code

Why hard language features are hard to analyze?

- Java Reflection
- Native Code

The notorious feature of Java
for static analysis

re re re … reflection!

Open Hard Problem

- "*Reflection makes it difficult to analyze statically.*" [1]
- "*Static analysis of object-oriented code is an exciting, ongoing and challenging research area, made especially challenging by dynamic language features, a.k.a. reflection.*" [2]
- "*Reflection usage … make it very difficult to scale points-to analysis to modern Java programs.*" [3]
- "*In our experience [4], the largest challenge to analyzing Android apps is their use of reflection.*" [5]

[1] *Rastogi et al. DroidChameleon: evaluating Android anti-malware against transformation attacks. Asia CCS'13.*
[2] *Landman et al. Challenges for Static Analysis of Java Reflection: Literature Review and Empirical Study. ICSE'17*
[3] *WALA. IBM T.J. Watson Libraries for Analysis*
[4] *Ernst et al. Collaborative Verification of Information Flow for a High-Assurance App Store. CCS'14.*
[5] *Barros et al. Static Analysis of Implicit Control Flow: Resolving Java Reflection and Android Intents. ASE'15.*

Java Reflection

```
Class Person {
  String name;
  void setName(String nm) {…};
}

Person p = new Person();
p.setName("John");
```

Java Reflection

```
Class Person {
  String name;
  void setName(String nm) {…};
}

Person p = new Person();
p.setName(“John”);
```

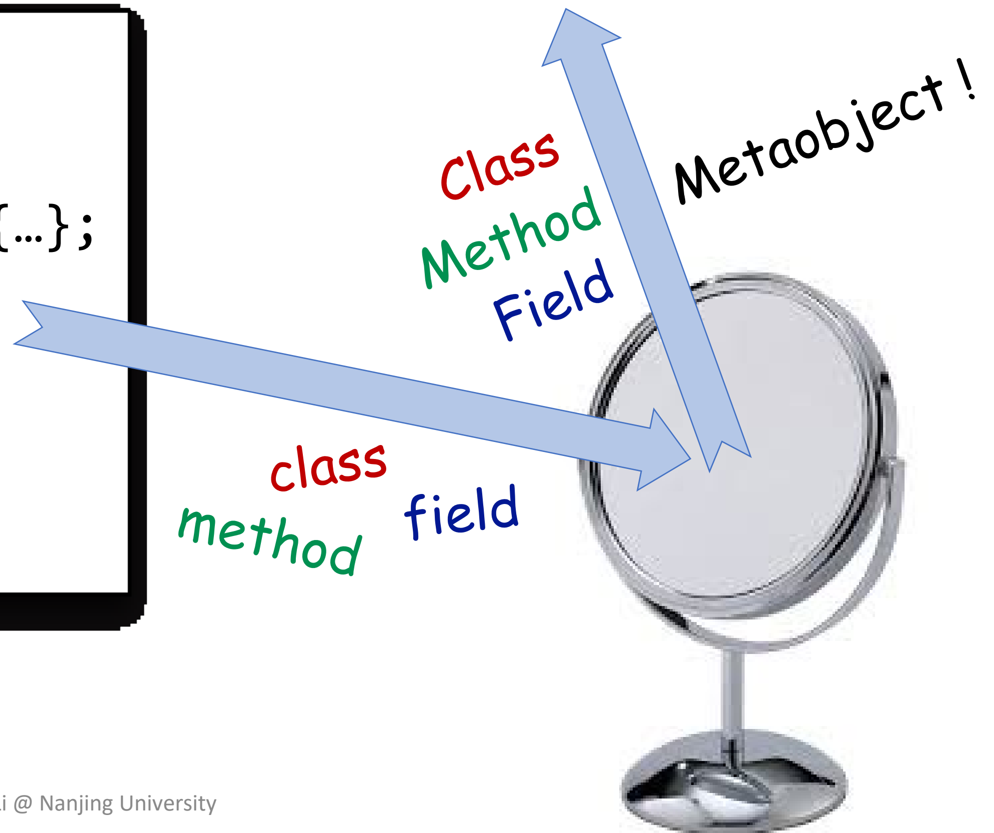
class

method

field

Java Reflection

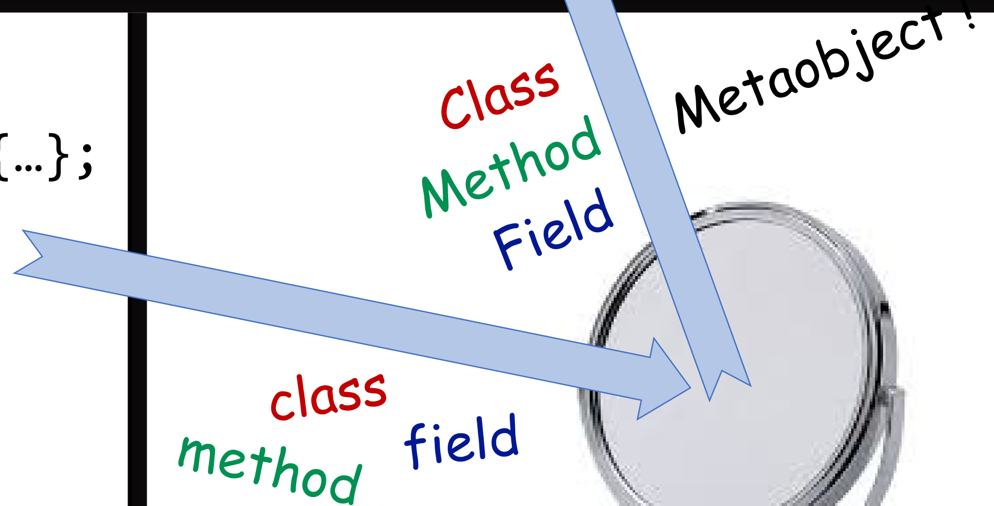
```
Class Person {
  String name;
  void setName(String nm) {…};
}

Person p = new Person();
p.setName("John");
```



Java Reflection

```
Class c = Class.forName("Person");
Method m = c.getMethod("setName", …);
Field f = c.getField("name");
Object p = c.newInstance();
m.invoke(p, "John");
f.set(p, …); s = (String) f.get(p);
```

```
Class Person {
  String name;
  void setName(String nm) {…};
}

Person p = new Person();
p.setName("John");
```



Java Reflection

```
Class c = Class.forName("Person");
Method m = c.getMethod("setName", …);
Field f = c.getField("name");
Object p = c.newInstance();
m.invoke(p, "John");
f.set(p, …); s = (String) f.get(p);
```

Run-time

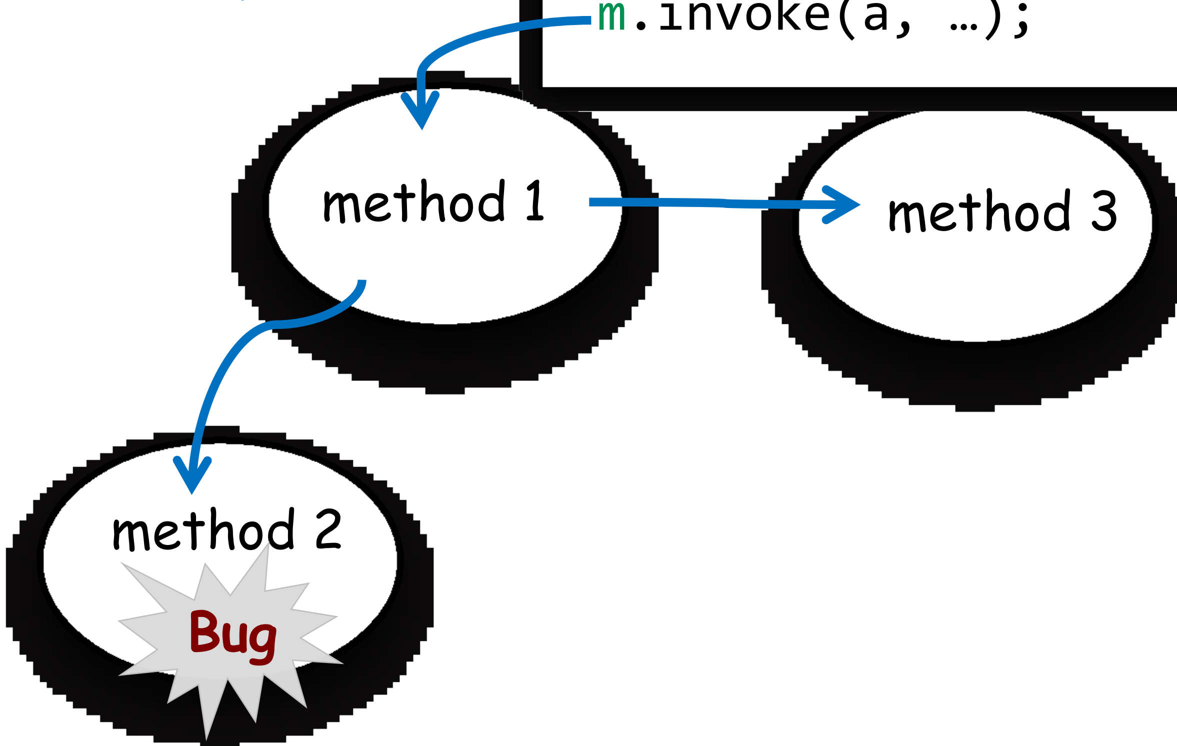
```
Class Person {
  String name;
  void setName(String nm) {…};
}

Person p = new Person();
p.setName("John");
```

Compile-time

Why We Need to Analyze Java Reflection

Why We Need to Analyze Java Reflection

```java
Class c = Class.forName(cName);
Method m = c.getMethod(mName, …);
A a = new A();
m.invoke(a, …);
```



Why We Need to Analyze Java Reflection

```
Class c = Class.forName(cName);
Method m = c.getMethod(mName, …);
A a = new A();
m.invoke(a, …);
```

Bug detection
Security Analysis

Call Graph Edge

method 1

method 3

method 2

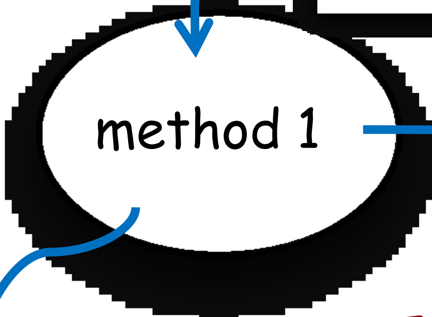
Bug

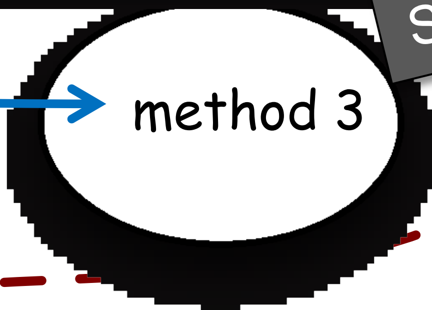
Why We Need to Analyze Java Reflection

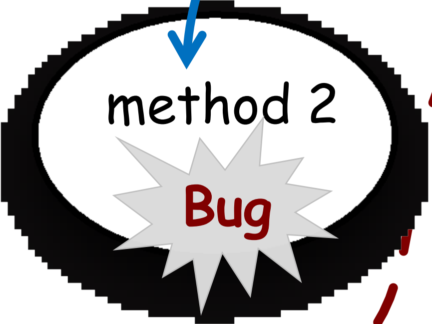
```
Class c = Class.forName(cName);
Method m = c.getMethod(mName, …);
A a = new A();
m.invoke(a, …);
```

Call Graph Edge

Bug detection
Security Analysis








```
B b = new B();
a.fld = b;
```

```
Field f = c.getField("fld");
f.set(a, a);  // a.fld = a
```

```
B b' = (B) a.fld;
```

Not Safe

Why We Need to Analyze Java Reflection

```
Class c = Class.forName(cName);
Method m = c.getMethod(mName, …);
A a = new A();
m.invoke(a, …);
```

Call Graph Edge

method 1

method 3

method 2

Bug

Bug detection
Security Analysis

```
B b = new B();
a.fld = b;
```

```
Field f = c.getField("fld");
f.set(a, a);   // a.fld = a
```

```
B b' = (B) a.fld;
```

Not Safe

Verification
Optimization

How to Analyze Java Reflection?

- String Constant analysis + Pointer Analysis

  Reflection Analysis for Java (APLAS 2005)

  Benjamin Livshits, John Whaley, Monica S. Lam. Stanford University

How to Analyze Java Reflection?

- String Constant analysis + Pointer Analysis

```
String cName = "Person";
String mName = "setName";
String fName = "name";
… …
Class  c = Class.forName(cName);
Method m = c.getMethod(mName, …);
Field  f = c.getField(fName);
… …
m.invoke(p, …);
```

How to Analyze Java Reflection?

- String Constant analysis + Pointer Analysis

```
String cName = “Person”;
String mName = “setName”;
String fName = “name”;
… …
Class  c = Class.forName(cName);
Method m = c.getMethod(mName, …);
Field  f = c.getField(fName);
… …
m.invoke(p, …);
```

- Configuration files
- Internet
- Command lines
- Complex string manipulations
- Dynamically generated
- Encrypted

How to Analyze Java Reflection?

- String Constant analysis + Pointer Analysis

```
String cName = "Person";
String mName = "setName";
String fName = "name";
… …
Class  c = Class.forName(cName);
Method m = c.getMethod(mName, …);
Field  f = c.getField(fName);
… …
m.invoke(p, …);
```

- Configuration files
- Internet
- Command lines
- Complex string manipulations
- Dynamically generated
- Encrypted

Problem: Reflection targets cannot be resolved if the string values are statically unknown

How to Analyze Java Reflection?

- String Constant analysis + Pointer Analysis

```
String cName = "Person";
String mName = "setName";
String fName = "name";
… …
Class  c = Class.forName(cName);
Method m = c.getMethod(mName, …);
Field  f = c.getField(fName);
… …
m.invoke(p, …);
```

- Configuration files
- Internet
- Command lines
- Complex string manipulations
- Dynamically generated
- Encrypted

Problem: Reflection targets cannot be resolved if the string values are statically unknown

End of Story?

How to Analyze Java Reflection?

- String Constant analysis + Pointer Analysis

  Reflection Analysis for Java (APLAS 2005)

  Benjamin Livshits, John Whaley, Monica S. Lam. Stanford University

- Type Inference + String analysis + Pointer Analysis

  Self-Inferencing Reflection Resolution for Java (ECOOP 2014)

  Yue Li, Tian Tan, Yulei Sui, Jingling Xue. UNSW Sydney

When string arguments cannot be resolved statically,

infer the reflective targets at their usage points!

How to Analyze Java Reflection?

- Type Inference + String Analysis + Pointer Analysis

Application: Eclipse (v4.2.2)

Class:org.eclipse.osgi.framework.internal.core.FrameworkCommandInterpreter

```
123  public Object execute(String cmd) {...
155    Object[] parameters = new Object[] {this}; ...
167    for (int i = 0; i < size; i++) {
174      method = target.getClass().getMethod("_" + cmd, parameterTypes);
175      retval = method.invoke(target, parameters); ...}
228  }
```

How to Analyze Java Reflection?

- Type Inference + String Analysis + Pointer Analysis

Application: Eclipse (v4.2.2)

Class:org.eclipse.osgi.framework.internal.core.FrameworkCommandInterpreter

```
123  public Object execute(String cmd) {...
155    Object[] parameters = new Object[] {this}; ...
167    for (int i = 0; i < size; i++) {
174      method = target.getClass().getMethod("_" + cmd, parameterTypes);
175      retval = method.invoke(target, parameters); ...}
228  }
```

Creation Point

?

How to Analyze Java Reflection?

- Type Inference + String Analysis + Pointer Analysis

Application: Eclipse (v4.2.2)

```
Class:org.eclipse.osgi.framework.internal.core.FrameworkCommandInterpreter
123  public Object execute(String cmd) {...
155    Object[] parameters = new Object[] {this}; ...
167    for (int i = 0; i < size; i++) {
174      method = target.getClass().getMethod("_" + cmd, parameterTypes);
175      retval = method.invoke(target, parameters); ...}
228  }
```

Creation Point

Usage point

How to Analyze Java Reflection?

- Type Inference + String Analysis + Pointer Analysis

```
Application: Eclipse (v4.2.2)
Class:org.eclipse.osgi.framework.internal.core.FrameworkCommandInterpreter
123  public Object execute(String cmd) {...
155    Object[] parameters = new Object[] {this}; ...
167    for (int i = 0; i < size; i++) {
174      method = target.getClass().getMethod("_" + cmd, parameterTypes);
175      retval = method.invoke(target, parameters); ...}
228  }
```

Creation Point

Usage point

The reflective target method at line 175 must have one parameter and its declared type must be FrameworkCommandInterpreter or its sub/supertypes

How to Analyze Java Reflection?

- Type Inference + String Analysis + Pointer Analysis

Application: Eclipse (v4.2.2)

```
Class:org.eclipse.osgi.framework.internal.core.FrameworkCommandInterpreter
123  public Object execute(String cmd) {...
155    Object[] parameters = new Object[] {this}; ...
167    for (int i = 0; i < size; i++) {
174      method = target.getClass().getMethod("_" + cmd, parameterTypes);
175      retval = method.invoke(target, parameters); ...}
228  }
```

Creation Point

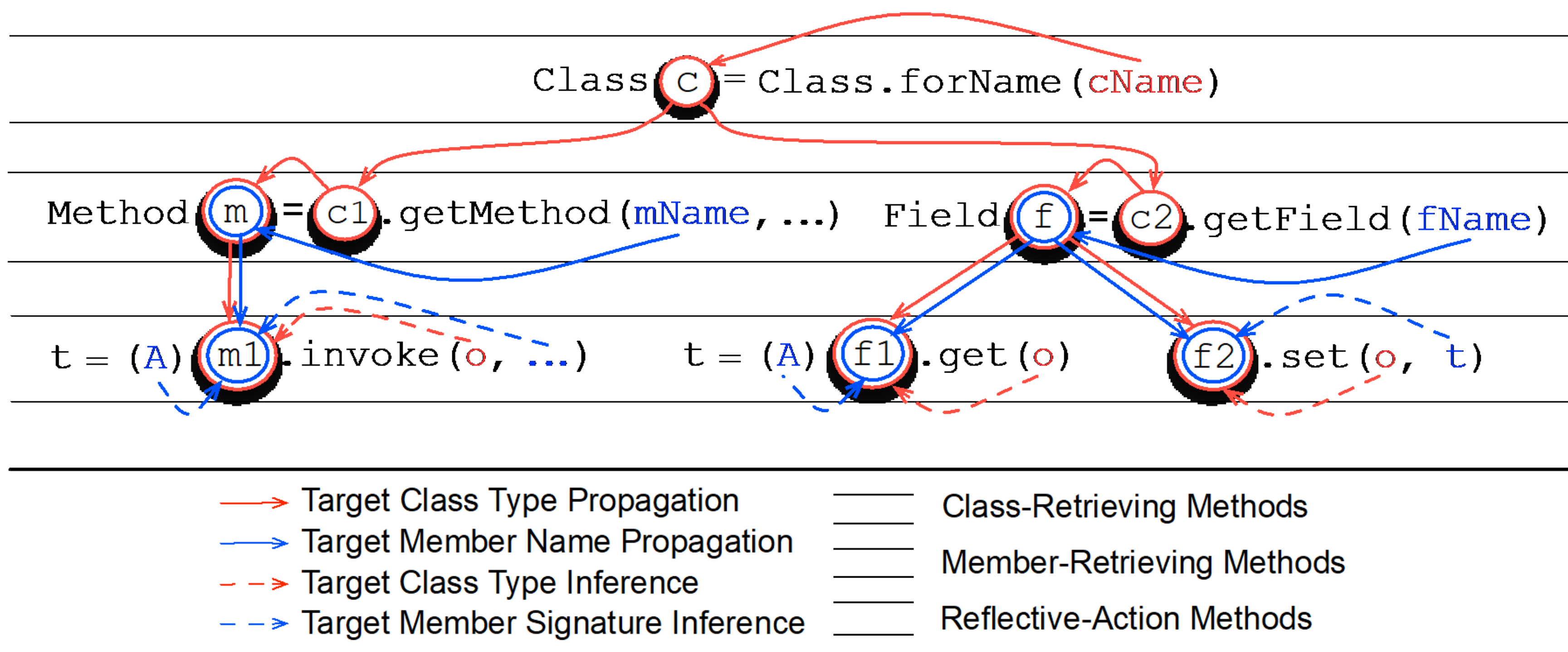
Usage point

The reflective target method at line 175 must have one parameter and its declared type must be FrameworkCommandInterpreter or its sub/supertypes

Infer 50 reflective target methods and 48 methods are true!

How to Analyze Java Reflection?

- Type Inference + String Analysis + Pointer Analysis



Understanding and Analyzing Java Reflection (TOSEM 2019)

Yue Li, Tian Tan, Jingling Xue

Introduce more advanced handlings

How to Analyze Java Reflection?

- String Constant analysis + Pointer Analysis

  Reflection Analysis for Java (APLAS 2005)

  Benjamin Livshits, John Whaley, Monica S. Lam. Stanford University

- Type Inference + String analysis + Pointer Analysis

  Self-Inferencing Reflection Resolution for Java (ECOOP 2014)

  Yue Li, Tian Tan, Yulei Sui, Jingling Xue. UNSW Sydney

- Assisted by Dynamic Analysis

  Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders (ICSE 2011)

  Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, Mira Mezini. Technische Universität Darmstadt

Why hard language features are hard to analyze?

- Java Reflection
- Native Code

What Happens When Print in Java?

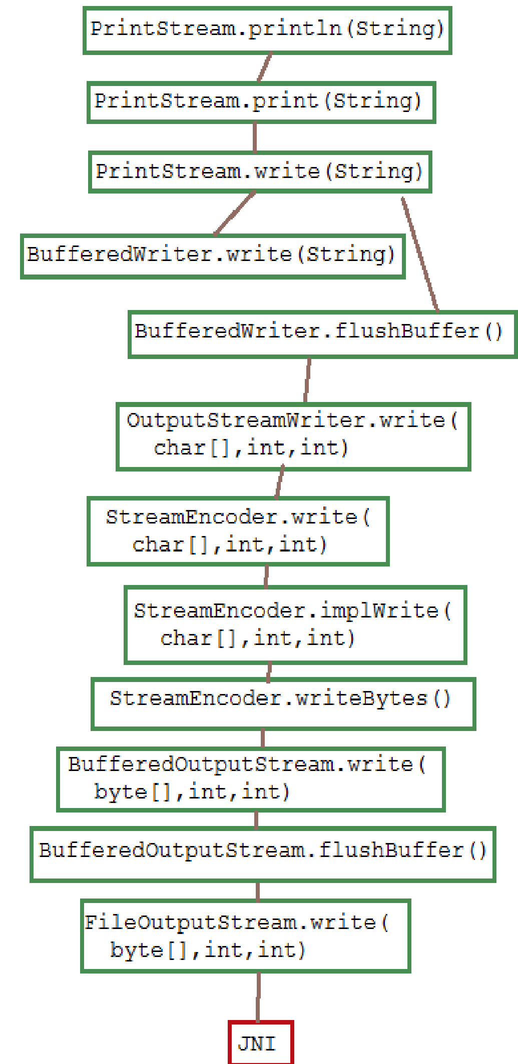
MyHello.java:

```
public static void main(String[] args) {
    System.out.println(“Hello World”);
}
```

What Happens When Print in Java?

MyHello.java:

```
public static void main(String[] args) {
    System.out.println("Hello World");
}
```

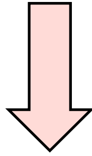


https://luckytoilet.wordpress.com/2010/05/21/how-system-out-println-really-works

What Happens When Print in Java?

MyHello.java:

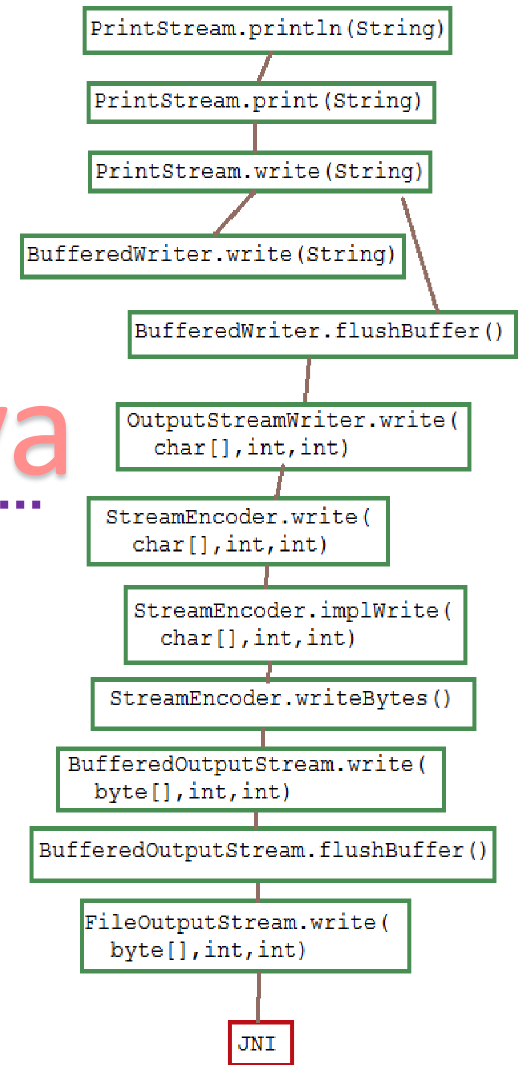
```
public static void main(String[] args) {
      System.out.println("Hello World");
}
```



java.io.FileOutputStream.java:

```
private native void writeBytes(byte b[],
          int off, int len) throws IOException
```

Java

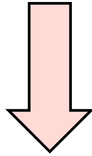


https://luckytoilet.wordpress.com/2010/05/21/how-system-out-println-really-works

What Happens When Print in Java?

MyHello.java:

```
public static void main(String[] args) {
      System.out.println("Hello World");
}
```

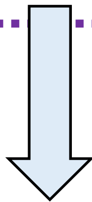


java.io.FileOutputStream.java:

```
private native void writeBytes(byte b[],
           int off, int len) throws IOException
```

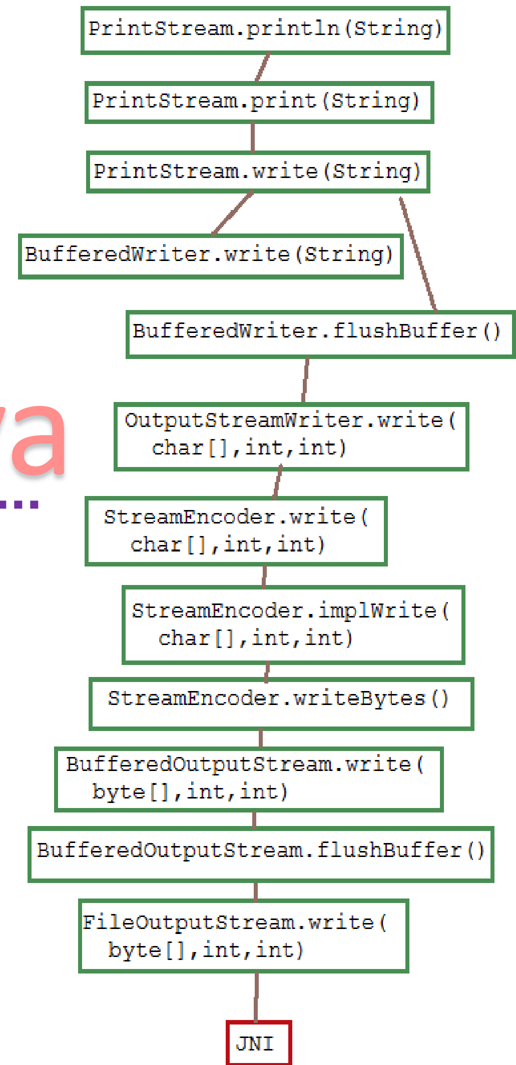
Java

C



FileOutputStream_md.c

```
JNIEXPORT void JNICALL
Java_java_io_FileOutputStream_writeBytes(JNIEnv *env,
     jobject this, jbyteArray bytes, jint off, jint len) {
     writeBytes(env, this, bytes, off, len, fos_fd);
}
```

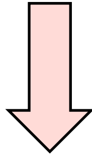


https://luckytoilet.wordpress.com/2010/05/21/how-system-out-println-really-works

What Happens When Print in Java?

MyHello.java:

```
public static void main(String[] args) {
     System.out.println("Hello World");
}
```



java.io.FileOutputStream.java:

```
private native void writeBytes(byte b[],
          int off, int len) throws IOException
```

Java

C

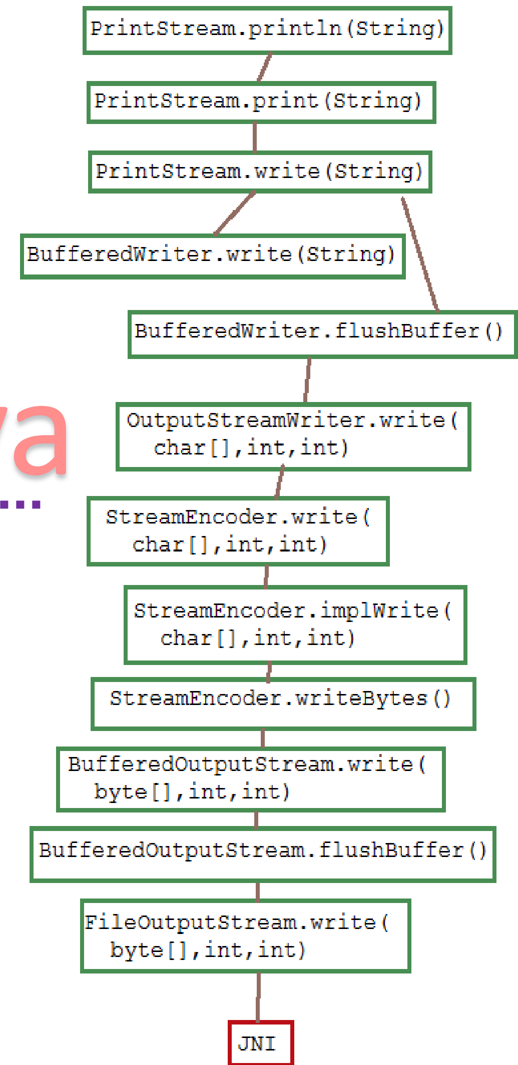
FileOutputStream_md.c

```
JNIEXPORT void JNICALL
Java_java_io_FileOutputStream_writeBytes(JNIEnv *env,
     jobject this, jbyteArray bytes, jint off, jint len) {
     writeBytes(env, this, bytes, off, len, fos_fd);
}
```

The further code is platform dependent

Linux: os:write (c++)

Windows: WriteFile (c)



https://luckytoilet.wordpress.com/2010/05/21/how-system-out-println-really-works

Java Native Interface (JNI)

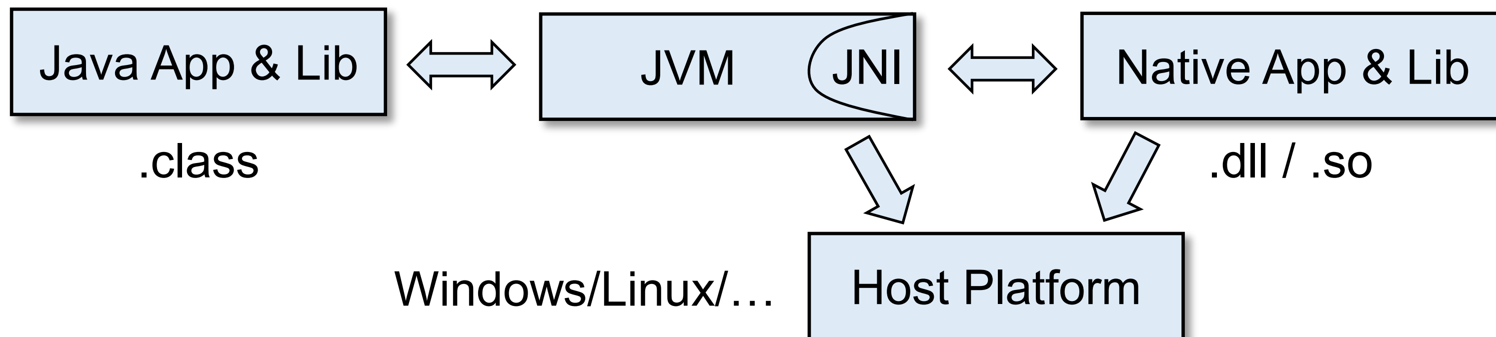
- What is JNI?

- Why we need JNI?

Java Native Interface (JNI)

- What is JNI?

  A function module of JVM which allows interoperation between Java and Native code (C/C++)

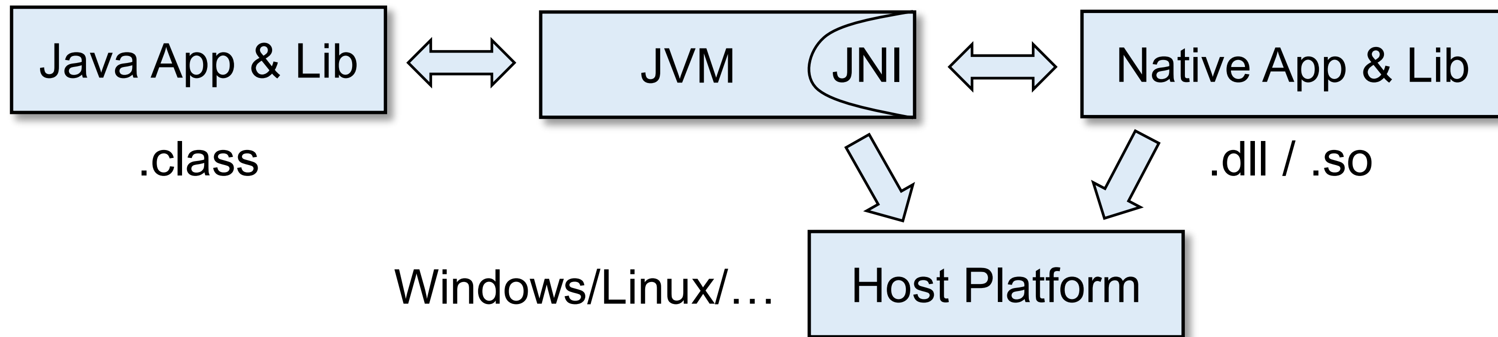


- Why we need JNI?

Java Native Interface (JNI)

- What is JNI?

  A function module of JVM which allows interoperation between Java and Native code (C/C++)



- Why we need JNI?
  - Need platform dependent features (interoperate with OS)
  - Reuse existing libraries (mostly written in C/C++)

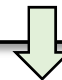
Why Native Code is Hard to Analyze?

```java
public class JNIExample {
    static { System.loadLibrary("GuessMeLib"); }
    private native Object guessMe();
    public static void main(String[] args) {
        JNIExample je = new JNIExample();
        Object obj = je.guessMe();
        … …
    }
}
```

Java

C

```c
# include <jni.h>
JNIEXPORT jobject JNICALL Java_JNIExample_guessMe(JNIEnv *env, jobject obj) {
    jclass clz = (*env)->FindClass(env, "JNIExample");
    jmethodID constructor = (*env)->GetMethodID(env, clz, "<init>", "()V");
    return (*env)->NewObject(env, clz, constructor);
}
```



GuessMeLib.so

Why Native Code is Hard to Analyze?

Load native library

```
public class JNIExample {
    static { System.loadLibrary("GuessMeLib"); }
    private native Object guessMe();
    public static void main(String[] args) {
        JNIExample je = new JNIExample();
        Object obj = je.guessMe();
        … …
    }
}
```

Java

C

```
# include <jni.h>
JNIEXPORT jobject JNICALL Java_JNIExample_guessMe(JNIEnv *env, jobject obj) {
    jclass clz = (*env)->FindClass(env, "JNIExample");
    jmethodID constructor = (*env)->GetMethodID(env, clz, "<init>", "()V");
    return (*env)->NewObject(env, clz, constructor);
}
```

GuessMeLib.so

Why Native Code is Hard to Analyze?

Load native library

```
public class JNIExample {
    static { System.loadLibrary("GuessMeLib"); }
    private native Object guessMe();
    public static void main(String[] args) {
        JNIExample je = new JNIExample();
        Object obj = je.guessMe();
        … …
    }
}
```

declare native method

Java

C

```
# include <jni.h>
JNIEXPORT jobject JNICALL Java_JNIExample_guessMe(JNIEnv *env, jobject obj) {
    jclass clz = (*env)->FindClass(env, "JNIExample");
    jmethodID constructor = (*env)->GetMethodID(env, clz, "<init>", "()V");
    return (*env)->NewObject(env, clz, constructor);
}
```

GuessMeLib.so

Why Native Code is Hard to Analyze?

```
public class JNIExample {
    static { System.loadLibrary("GuessMeLib"); }
    private native Object guessMe();
    public static void main(String[] args
        JNIExample je = new JNIExample();
        Object obj = je.guessMe();
        … …
    }
}
```

Load native library

declare native method

JNI functions allow to create objects, access fields, invoke methods, etc. in native code

230 JNI functions

```
FindClass
GetMethodID
NewObject
CallVoidMethod
… …
```

Java

C

```
# include <jni.h>
JNIEXPORT jobject JNICALL Java_JNIExample_guessMe(JNIEnv *env, jobject obj) {
    jclass clz = (*env)->FindClass(env, "JNIExample");
    jmethodID constructor = (*env)->GetMethodID(env, clz, "<init>", "()V");
    return (*env)->NewObject(env, clz, constructor);
}
```

GuessMeLib.so

Why Native Code is Hard to Analyze?

Load native library

```
public class JNIExample {
    static { System.loadLibrary("GuessMeLib"); }
    private native Object guessMe();
    public static void main(String[] args
        JNIExample je = new JNIExample();
        Object obj = je.guessMe();
        … …
    }
}
```

declare native method

How Java static analyzer analzyes this call?

JNI functions allow to create objects, access fields, invoke methods, etc. in native code

230 JNI functions

```
FindClass
GetMethodID
NewObject
CallVoidMethod
… …
```

Java

C

```
# include <jni.h>
JNIEXPORT jobject JNICALL Java_JNIExample_guessMe(JNIEnv *env, jobject obj) {
    jclass clz = (*env)->FindClass(env, "JNIExample");
    jmethodID constructor = (*env)->GetMethodID(env, clz, "<init>", "()V");
    return (*env)->NewObject(env, clz, constructor);
}
```

GuessMeLib.so

How to Handle Native Code?

- Current practice

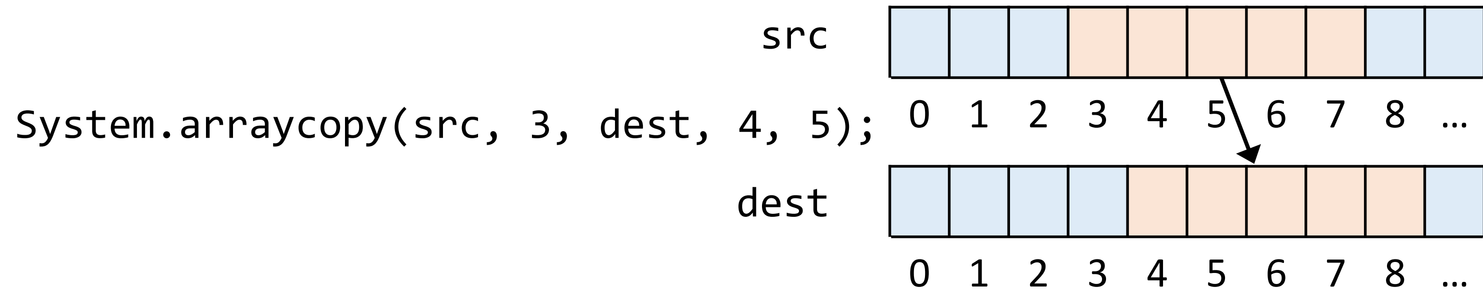
  Manually models the critical native code

Example

Native Code Modeling (Example)

```
java.lang.System.arraycopy(src, srcPos, dest, destPos, length)
```

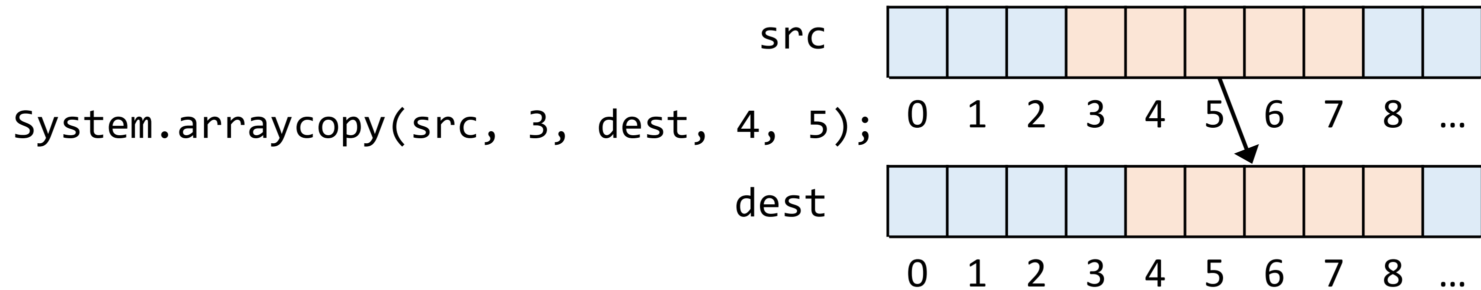
- Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array



Native Code Modeling (Example)

```
java.lang.System.arraycopy(src, srcPos, dest, destPos, length)
```

- Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array

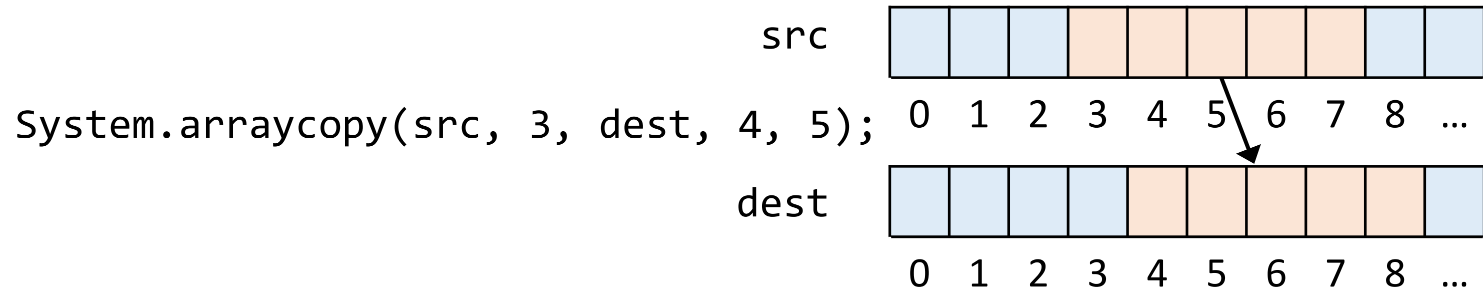


- For efficiency, arraycopy() is implemented in native code

Native Code Modeling (Example)

```
java.lang.System.arraycopy(src, srcPos, dest, destPos, length)
```

- Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array



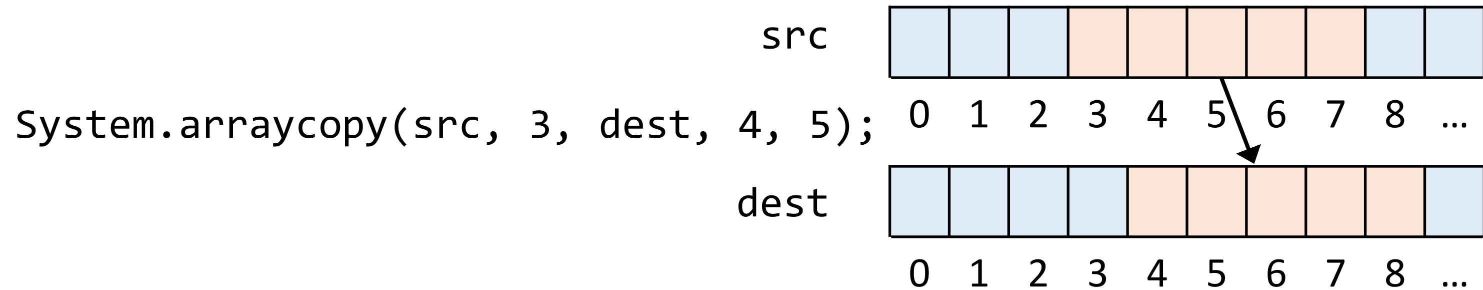
- For efficiency, arraycopy() is implemented in native code
- An alternative modeling

```
System.arraycopy(src, sp, dest, dp, ln);
```

Native Code Modeling (Example)

```
java.lang.System.arraycopy(src, srcPos, dest, destPos, length)
```

- Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array



- For efficiency, arraycopy() is implemented in native code
- An alternative modeling

~~System.arraycopy(src, sp, dest, dp, ln)~~;

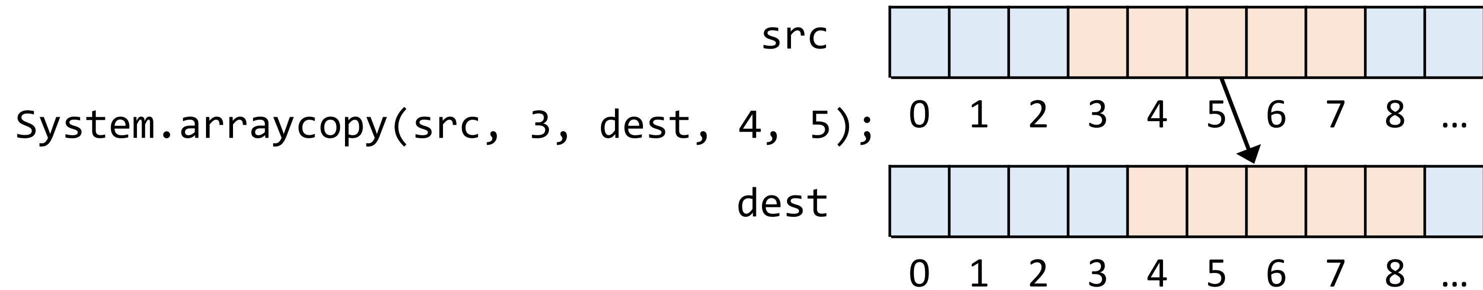
```
for(int i = 0; i < ln; i++)
    dest[dp+i] = src[sp+i];
```

Java

Native Code Modeling (Example)

`java.lang.System.arraycopy(src, srcPos, dest, destPos, length)`

- Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array



- For efficiency, arraycopy() is implemented in native code
- An alternative modeling

```
System.arraycopy(src, sp, dest, dp, ln);
for(int i = 0; i < ln; i++)
    dest[dp+i] = src[sp+i];
```

Java

```
    temp = src.arr;
    dest.arr = temp;
```

pointer analysis's view

How to Handle Native Code? (Cont.)

- Current practice

  Manually models the critical native code

Example

- Recent work

  Identifying Java Calls in Native Code via Binary Scanning (ISSTA 2020)

  George Fourtounis, Leonidas Triantafyllou, Yannis Smaragdakis,, University of Athens

More About Soundiness

Soundiness Home Page

If you wanted to learn about soundiness, you are in the right place. Below is a brief excerpt from our Soundiness manifesto...

Static program analysis is a key component of many software development tools, including compilers, development environments, and verification tools. Practical applications of static analysis have grown in recent years to include tools by companies such as Coverity, Fortify, GrammaTech, IBM, and others. Analyses are often expected to be sound in that their result models all possible executions of the program under analysis. Soundness implies that the analysis computes an over-approximation in order to stay tractable; the analysis result will also model behaviors that do not actually occur in any program execution. The precision of an analysis is the degree to which it avoids such spurious results. Users expect analyses to be sound as a matter of course, and desire analyses to be as precise as possible, while being able to scale to large programs.

Soundness would seem essential for any kind of static program analysis. Soundness is also widely emphasized in the academic literature. Yet, in practice, soundness is commonly eschewed: we are not aware of a single realistic whole-program analysis tool (e.g., tools widely used for bug detection, refactoring assistance, programming automation, etc.) that does not purposely make unsound choices. Similarly, virtually all published whole-program analyses are unsound and omit conservative handling of common language features when applied to real programming languages.

The typical reasons for such choices are engineering compromises: implementers of such tools are well aware of how they could handle complex language features soundly (e.g., by assuming that a complex language feature can exhibit any behavior), but do not do so because this would make the analysis unscalable or imprecise to the point of being useless. Therefore, the dominant practice is one of treating soundness as an engineering choice.

In all, we are faced with a paradox: on the one hand we have the ubiquity of unsoundness in any practical whole-program analysis tool that has a claim to precision and scalability; on the other, we have a research community that, outside a small group of experts, is oblivious to any unsoundness, let alone its preponderance in practice.

The term "soundiness" has been inspired in part by Stephen Colbert's "truthiness."

http://soundiness.org

graphy

Below is a short bibliography of papers related to soundiness produced from this bibtex file. We aim to list papers that either (1) measure the unsoundness of whole-program analyses in some way, (2) give solutions to analyzing "hard" language features, or (3) measure the utility of soundness in a particular context. If you have further suggestions for references, submit an issue or a pull request.

- Li, Y., Tan, T. and Xue, J. Effective Soundness-Guided Reflection Analysis. SAS (2015).
- Smaragdakis, Y. and Kastrinis, G. and Balatsouras, G. and Bravenboer, M. More Sound Static Handling of Java Reflection, 2014.
- Christakis, M., Müller, P. and Wüstholz, V. An Experimental Evaluation of Deliberate Unsoundness in a Static Program Analyzer. VMCAI (2015).
- Christakis, M., Müller, P. and Wüstholz, V. Collaborative Verification and Testing with Explicit Assumptions. FM (2012).
- Li, Y., Tan, T., Sui, Y. and Xue, J. Self-Inferencing Reflection Resolution for Java. ECOOP (2014).
- Feldthaus, A., Schäfer, M., Sridharan, M., Dolby, J. and Tip, F. 2013. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. ICSE (2013), 752–761.
- Richards, G. 2012. Eval Begone ! Semi-Automated Removal of Eval from JavaScript Programs. OOPSLA (2012).
- G. Soares, R. Gheyi, and T. Massoni. Automated behavioral testing of refactoring engines. IEEE TSE (2013).
- Bodden, E., Sewe, A., Sinschek, J., Oueslati, H. and Mezini, M. 2011. Taming Reflection Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. ICSE (2011), 241–250.
- Richards, G., Hammer, C. and Burg, B. 2011. The Eval that men do: A large-scale study of the use of eval in JavaScript applications. ECOOP (2011).
- Bravenboer, M. and Smaragdakis, Y. 2009. Strictly declarative specification of sophisticated points-to analyses. ACM SIGPLAN Notices (Oct. 2009), 243.
- Smaragdakis, Y. and Csallner, C. 2007. Combining Static and Dynamic Reasoning for Bug Detection. Tests and Proofs (2007).
- Lhoták, O. 2007. Comparing call graphs. PASTE (2007).
- Xue, J. and Nguyen, PH. 2005. Completeness Analysis for Incomplete Object-Oriented Programs. CC (2005).
- Livshits, B., Whaley, J. and Lam, M. 2005. Reflection analysis for Java. Programming Languages and Systems. 0326227 (2005).
- Flanagan, Cormac and Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B. and Stata, R. 2002. Extended Static Checking for Java. PLDI (2002).

Contents

1. Soundness and Soundiness

2. Hard Language Feature: Java Reflection

3. Hard Language Feature: Native Code

The X You Need To Understand in This Lecture

- Understand soundness: its motivation and concept
- Understand why Java reflection and native code are hard to analyze

注意注意！
划重点了！

软件分析

南京大学
计算机科学与技术系
程序设计语言与
静态分析研究组
李樾 谭添

NANJING UNIVERSITY 1902