Foundations and implementation of a tool bench for static analysis of Java bytecode programs

Laurent Hubert
CNRS/IRISA - Université de Rennes 1
EPI Lande/Celtique

December 17, 2010
PhD advisors: T. Jensen et D. Pichardie
Introduction

On Static Analysis

• Definition

  An automatic analysis of a program performed without actually executing the program

• Goal: improving software quality
Introduction
On Static Analysis

• Definition

An automatic analysis of a program performed without actually executing the program

• Goal: improving software quality

• Other methods
  • Formal methods and code extraction from models or proofs
  • Test
  • Guidelines & best practices
  • etc.
Importance of Software Quality

• Software is used
  • in mobile phones, cars, aircraft;
  • in to design of bridges, roads, aircraft, railways;
  • to predict the future and improve safety (Météo France, Bison Futé, etc.);
  • in banking systems, accounting, human resources;
  • to handle traffic lights, railroad switches;
  • etc.

• Software is expensive

• Verification and validation
  • Expensive, even more than the development
Improving Software Quality with Static Analysis

• Finding errors
  • Usually not sound nor complete, but useful in practice

• Proving partial correctness
  • Allow proving automatically a program respects some property
    • Termination,
    • NullPointerException freeness
    • Initialization of variables before first access,
    • etc.

• These analyses should be sound
Static Analysis and Computability

False Positives and False Negatives

- A static analyzer checks a difficult property (ex: does the program throw a NullPointerException?)
- It cannot be perfect
  - false positive: reject a correct program
  - false negative: accept an incorrect program
Static Analysis and Computability
False Positives and False Negatives

- A static analyzer checks a difficult property (ex: does the program throw a NullPointerException?)
- It cannot be perfect
  - false positive: reject a correct program
  - false negative: accept an incorrect program
Static Analysis and Computability

False Positives and False Negatives

• A static analyzer checks a difficult property (ex: does the program throw a NullPointerException?)

• It cannot be perfect
  • false positive: reject a correct program
  • false negative: accept an incorrect program
Static Analysis and Computability

False Positives and False Negatives

• A static analyzer checks a difficult property (ex: does the program throw a NullPointerException?)

• It cannot be perfect
  • false positive: reject a correct program
  • false negative: accept an incorrect program
A static analyzer checks a difficult property (ex: does the program throw a NullPointerException?)

It cannot be perfect

- false positive: reject a correct program
- false negative: accept an incorrect program
A static analyzer checks a difficult property (ex: does the program throw a NullPointerException?)

- It cannot be perfect
  - false positive: reject a correct program
  - false negative: accept an incorrect program
Static Analysis and Computability

False Positives and False Negatives

- A static analyzer checks a difficult property (ex: does the program throw a NullPointerException?)
- It cannot be perfect
  - false positive: reject a correct program
  - false negative: accept an incorrect program

Precision

false positive rate
(over correct programs)

false negative rate
(over incorrect programs)

manual testing

sound static analysis

reject all

100%

0

100%

5
Static Analysis and Computability

False Positives and False Negatives

- A static analyzer checks a difficult property (ex: does the program throw a NullPointerException?)

- It cannot be perfect
  - false positive: reject a correct program
  - false negative: accept an incorrect program

Precision

false positive rate
(over correct programs)

false negative rate
(over incorrect programs)

sound static analysis

manual testing
Our Approach
Soundness and Precision

- Soundness: should always report incorrect programs (no false negative)
  - Formal specification on small languages
  - Proof of soundness
- Precision (not too many false positives)
  - Study on small, chosen examples
  - Implementation and experimental evaluation to check precision in practice
Language

• Why Java?
  • Popular language
  • Bugs may be expensive in Java applications
    • Largely spread (mobile and smart phones, browsers, etc.)
    • Embedded computing (Javacard, Safety Critical Java, etc.)

• Why Java bytecode?
  • Source may not be available (libraries, load-time analysis, etc.)
  • Less constructs
  • Less evolutions
Soundness

Operational semantics

Correctness relation

Analysis (constraints)
Soundness

Operational semantics

Correctness relation

Analysis (constraints)

• Subject reduction (or preservation) lemma

If a concrete state is correctly abstracted by an abstract state resulting from the constraint solving, then, after another step of the concrete semantics, the new state is also correctly abstracted by the abstract state.
Soundness

Operational semantics

Concrete state

Correctness relation

Abstract state

Analysis (constraints)

• Subject reduction (or preservation) lemma

If a concrete state is correctly abstracted by an abstract state resulting from the constraint solving, then, after another step of the concrete semantics, the new state is also correctly abstracted by the abstract state.
Soundness

Operational semantics

Concrete state

Correctness relation

Abstract state

Analysis (constraints)

• Subject reduction (or preservation) lemma

If a concrete state is correctly abstracted by an abstract state resulting from the constraint solving, then, after another step of the concrete semantics, the new state is also correctly abstracted by the abstract state.
Subject reduction (or preservation) lemma

If a concrete state is correctly abstracted by an abstract state resulting from the constraint solving, then, after another step of the concrete semantics, the new state is also correctly abstracted by the abstract state.
Subject reduction (or preservation) lemma

If a concrete state is correctly abstracted by an abstract state resulting from the constraint solving, then, after another step of the concrete semantics, the new state is also correctly abstracted by the abstract state.
Initialization

• Initialization is often a critical phase
  • security checks may be performed
  • defense mechanisms may be installed
  • operations may be logged
  • fields are initialized
  • etc.
Initialization

- Initialization is often a critical phase
  - Invariants are established

Only fully initialized **objects** should be manipulated by the program

Only fully initialized **classes** should be observable by the program

- Recommendation of the CERT (OBJ04-J), Oracle's guidelines for secure coding & Joshua Bloch in *Effective Java*
Outline

• Initialization of Classes
• Initialization of Objects
• Initialization of Fields
• Implementations
class A extends Object {
    static A EMPTY = new A(" ");
    static HashMap ALL = new HashMap();

    public A(String n) {
        ALL.add(n, this);
    }
}
class A extends Object {
    static A EMPTY;
    static HashMap ALL;

    public A(String n){
        ALL.add(n,this);
    }

    static <clinit>{
        EMPTY = new A("" );
        ALL = new HashMap();
    }
}
The JVM Spec Says...

2.17.4 Initialization

Initialization of a class consists of executing its static initializers (§3.11) and the initializers for static fields (§2.9.2) declared in the class. Initialization of an interface consists of executing the initializers for fields declared in the interface (§2.13.3.1).

Before a class or interface is initialized, its direct superclass must be initialized, but interfaces implemented by the class need not be initialized. Similarly, the superinterfaces of an interface need not be initialized before the interface is initialized.

A class or interface type T will be initialized immediately before one of the following occurs:

- T is a class and an instance of T is created.
- T is a class and a static method of T is invoked.
- A nonconstant static field of T is used or assigned. A constant field is one that is (explicitly or implicitly) both final and static, and that is initialized with the value of a compile-time constant expression. A reference to such a field must be resolved at compile time to a copy of the compile-time constant value, so uses of such a field never cause initialization.

Invocation of certain methods in library classes (§3.12) also causes class or interface initialization. See the Java 2 platform’s class library specifications (for example, class Class and package java.lang.reflect) for details.

The intent here is that a type have a set of initializers that put it in a consistent state and that this state be the first state that is observed by other classes. The static initializers and class variable initializers are executed in textual order and may not refer to class variables declared in the class whose declarations appear textually after use, even though those class variables are in scope. This restriction is designed to detect, at compile time, most circular or otherwise malformed initializations.

Before a class or interface is initialized its superclass is initialized, if it has not previously been initialized.

2.17.5 Detailed Initialization Procedure

Initialization of a class or interface requires careful synchronization, since some other thread may be trying to initialize the same class or interface at the same time. There is also the possibility that initialization of a class or interface may be requested recursively as part of the initialization of that class or interface; for example, a variable initialized in class A might invoke a method of an unrelated class B, which might in turn invoke a method of class A. The implementation of the Java virtual machine is responsible for taking care of synchronization and recursive initialization by using the following procedure. It assumes that the Class object has already been verified and prepared and that the Class object contains state that can indicate one of four situations:

- This Class object is verified and prepared but not initialized.
- This Class object is being initialized by some particular thread T.
- This Class object is fully initialized and ready for use.
- This Class object is in an erroneous state, perhaps because the verification step failed or because initialization was attempted and failed.

The procedure for initializing a class or interface is then as follows:

1. Synchronize on the Class object that represents the class or interface to be initialized. This involves waiting until the current thread can obtain the lock for that object (§8.13).
2. If initialization by some other thread is in progress for the class or interface, then wait on this Class object (which temporarily releases the lock). When the current thread awakens from the wait, repeat this step.
3. If initialization is in progress for the class or interface by the current thread, then this must be a recursive request for initialization. Release the lock on the Class object and complete normally.
4. If the class or interface has already been initialized, then no further action is required. Release the lock on the Class object and complete normally.
5. If the Class object is in an erroneous state, then initialization is not possible. Release the lock on the Class object and throw a NoClassDefFoundError.
6. Otherwise, record the fact that initialization of the Class object is now in progress by the current thread and release the lock on the Class object.
7. Next, if the Class object represents a class rather than an interface, and the direct superclass of this class has not yet been initialized, then recursively perform this entire procedure for the uninitialized superclass. If the initialization of the direct superclass completes abruptly because of a thrown exception, then lock this Class object, label it erroneous, notify all waiting threads, release the lock, and complete abruptly, throwing the same exception that resulted from initializing the superclass.
8. Next, execute either the class variable initializers and static initializers of the class or the field initializers of the interface, in textual order, as though they were a single block, except that final static variables and fields of interfaces whose values are compile-time constants are initialized first.
9. If the execution of the initializers completes normally, then lock this Class object, label it fully initialized, notify all waiting threads, release the lock, and complete this procedure normally.
10. Otherwise, the initializers must have completed abruptly by throwing some exception E. If the Class object is not E and one of its subclasses, then create a new instance of the class ExceptionInInitializerError, with E as the argument, and use this object in place of E in the following step. But if a new instance of ExceptionInInitializerError cannot be created because an OutOfMemoryError occurs, then instead use an OutOfMemoryError object in place of E in the following step.
11. Lock the Class object, label it erroneous, notify all waiting threads, release the lock, and complete this procedure abruptly with reason E or its replacement as determined in the previous step.

In some early implementations of the Java virtual machine, an exception during class initialization was ignored rather than allowing it to cause an ExceptionInInitializerError as described here.
A class or interface type T will be initialized immediately before one of the following occurs:

- T is a class and an instance of T is created.
- T is a class and a static method of T is invoked.
- A nonconstant static field of T is used or assigned. A constant field is one that is (explicitly or implicitly) both final and static, and that is initialized with the value of a compile-time constant expression. A reference to such a field must be resolved at compile time to a copy of the compile-time constant value, so uses of such a field never cause initialization.
The JVM Spec Says...

class initializers are lazily invoked,

A class or interface type T will be *initialized* immediately before one of the following occurs:

- T is a class and an instance of T is created.
- T is a class and a static method of T is invoked.
- A nonconstant static field of T is used or assigned. A constant field is one that is (explicitly or implicitly) both *final* and *static*, and that is initialized with the value of a compile-time constant expression. A reference to such a field must be resolved at compile time to a copy of the compile-time constant value, so uses of such a field never cause initialization.
The JVM Spec Says...

class initializers are lazily invoked, ...

...triggered by getstatic, putstatic, invokestatic and new instructions
The JVM Spec Says... 

class initializers are lazily invoked, ...triggered by getstatic, putstatic, invokestatic and new instructions
class initializers are lazily invoked,
...triggered by getstatic, putstatic, invokestatic and new instructions
...and called only once
class initializers are lazily invoked, ...triggered by getstatic, putstatic, invokestatic and new instructions. This makes the control flow hard to predict ... and called only once triggered by getstatic, putstatic, invokestatic and new instructions.
Control Flow

Example

class A extends Object {
    static A EMPTY;
    static HashMap ALL;

    public A(String n) {
        ALL.add(n, this);
    }

    static <clinit> {
        EMPTY = new A("");
        ALL = new HashMap();
    }
}
class A extends Object {
    static A EMPTY;
    static HashMap ALL;

    public A(String n) {
        ALL.add(n, this);
    }
}

static <clinit> {
    EMPTY = new A(" ");
    ALL = new HashMap();
}
Control Flow

Example

class A extends Object {
    static A EMPTY;
    static HashMap ALL;

    public A(String n) {
        ALL.add(n, this);
    }

    static <clinit> {
        EMPTY = new A(" ");
        ALL = new HashMap();
    }
}
Control Flow

Example

class A extends Object {
    static A EMPTY;
    static HashMap ALL;

    public A(String n) {
        ALL.add(n, this);
    }

    static <clinit> {
        EMPTY = new A("" );
        ALL = new HashMap();
    }
}
class A extends Object {
    static A EMPTY;
    static HashMap ALL;

    public A(String n) {
        ALL.add(n, this);
    }

    static <clinit> {
        EMPTY = new A("");
        ALL = new HashMap();
    }
}
Control Flow
Example

class A extends Object {
    static A EMPTY;
    static HashMap ALL;

    public A(String n){
        ALL.add(n, this);
    }

    static <clinit>{
        ALL = new HashMap();
    }
}
class A extends Object {
    static HashMap ALL;
    static A EMPTY;

    public A(String n) {
        ALL.add(n, this);
    }

    static <clinit> {
        ALL = new HashMap();
        EMPTY = new A(" ");
    }
}
Control Flow

Example

class A extends Object {
    static HashMap ALL;
    static A EMPTY;

    public A(String n) {
        ALL.add(n, this);
    }

    static <clinit> {
        ALL = new HashMap();
        EMPTY = new A(" ");
    }
}
class A extends Object {
    static HashMap ALL;
    static A EMPTY;

    public A(String n) {
        ALL.add(n, this);
    }

    static <clinit> {
        ALL = new HashMap();
        EMPTY = new A("" );
    }
}
class A extends Object {
    static HashMap ALL;
    static A EMPTY;

    public A(String n) {
        ALL.add(n, this);
    }

    static <clinit> {
        ALL = new HashMap();
        EMPTY = new A(" ");
    }
}
Control Flow

Example

class A extends Object {
    static HashMap ALL;
    static A EMPTY;

    public A(String n){
        ALL.add(n,this);
    }

    static <clinit>{
        ALL = new HashMap();
        EMPTY = new A(" ");
    }
}
Control Flow

Example

class A extends Object {
    static HashMap ALL;
    static A EMPTY;

    public A(String n) {
        ALL.add(n, this);
    }

    static <clinit> {
        ALL = new HashMap();
        EMPTY = new A(" ");
    }
}
class A extends Object {
    static HashMap ALL;
    static A EMPTY;

    public A(String n) {
        ALL.add(n, this);
    }

    static <clinit> {
        ALL = new HashMap();
        EMPTY = new A(" ");
    }
}
Control Flow

Example

class A extends Object {
    static HashMap ALL;
    static A EMPTY;

    public A(String n) {
        ALL.add(n, this);
    }

    static <clinit> {
        ALL = new HashMap();
        EMPTY = new A(" ");
    }
}
Control Flow

Example

class A extends Object {
    static HashMap ALL;
    static A EMPTY;

    public A(String n) {
        ALL.add(n, this);
    }

    static <clinit> {
        ALL = new HashMap();
        EMPTY = new A(""");
    }
}
A First Analysis

• Purpose of the analysis
  • Prove static fields are **written** before being **used**
  • Refine the CFG

• Instrumented semantics
  • $h$: set of classes for which initialization has **started**

• Flow sensitive abstraction: ($May$, $Must$, $Wf$)
  • $May$: over-approximates $h$
  • $Must$: under-approximates $h$
  • $Wf$: under-approximate the set of written fields

\[
Must \subseteq h \subseteq May
\]
A 3-valued Initialisation State Analysis

class A {
  static int F;
  static <clinit>{{
    F = 2;}}

  i = A.F;
}
A 3-valued Initialisation State Analysis

class A {
    static int F;
    static <clinit>{
        F = 2;
    }

    i = A.F;
}
A 3-valued Initialisation State Analysis

class A {
    static int F;
    static <clinit>{{
        F = 2;
    }}

    MayInit A;
    i = A.F;
}
A 3-valued Initialisation State Analysis

• 3 states
  
  • Initialization not started (α)
  
  • Under initialization (β)
  
  • Initialization finished (γ)

```java
class A {
    static int F;
    static <clinit> {
        F = 2;
    }

    MayInit A;
    i = A.F;
}
```
A 3-valued Initialisation State Analysis

- 3 states
  - Initialization not started ($\alpha$)
  - Under initialization ($\beta$)
  - Initialization finished ($\gamma$)

```java
class A {
    static int F;
    static <clinit>{
        F = 2;
    }
    MayInit A;
    i = A.F;
}
```

MayInit A; calls A.<clinit>
A 3-valued Initialisation State Analysis

- 3 states
  - Initialization not started ($\alpha$)
  - Under initialization ($\beta$)
  - Initialization finished ($\gamma$)

```java
class A {
    static int F;
    static <clinit> {
        F = 2;
    }
    MayInit A;
    i = A.F;
}
```

MayInit A; calls A.<clinit>

The initializer has already been fully run
A 3-valued Initialisation State Analysis

- 3 states
  - Initialization not started (\(\alpha\))
  - Under initialization (\(\beta\))
  - Initialization finished (\(\gamma\))

```java
class A {
    static int F;
    static <clinit> {
        F = 2;
    }
}
MayInit A;
i = A.F;
```

In both cases, after `MayInit A;` the initializer has been fully run.
Comparison

Comparison of the 2 analyses

2nd analysis (A \mapsto \{\alpha, \gamma\}, Wf=\{\})

• If A.<clinit> initializes F, F will be initialized
• 1st analysis (May={A}, Must={}, Wf=\{\})
  • No information (code may be reachable from A.<clinit>)

```java
class A {
    static int F;
    static <clinit>{
        F = 2;
    }
}
```

```java
i = A.F;
```
Comparison of the 2 analyses

- 2\textsuperscript{nd} analysis ($A \mapsto \{\alpha, \gamma\}, Wf=\{\})$
  - If $A.\texttt{<clinit>}$ initializes $F$, $F$ will be initialized
- 1\textsuperscript{st} analysis ($May=\{A\}, Must=\{\}, Wf=\{\}$)
  - No information (code may be reachable from $A.\texttt{<clinit>}$)
Initialization of Classes

• Identified
  • Challenge of class initialization
  • Examples that need to be analyzed

• Proposed a first analysis [BYTECODE'09]
  • Formally specified and proved sound
  • In practice: not precise enough

• Proposed a second analysis
  • Based on a 3-valued initialization state
  • Formally specified (but not yet proved sound nor implemented)
Outline

- Initialization of Classes
- Initialization of Objects
- Initialization of Fields
- Implementations
Object Initialization in Java

Timeline of an instance of class Bar

An instance of Bar
Object Initialization in Java

Timeline of an instance of class Bar

An instance of Bar

allocation of an instance of Bar
Object Initialization in Java

Timeline of an instance of class Bar

Allocation of an instance of Bar

Call to the constructor of Bar

An instance of Bar

Object

Foo

Bar

uninitialized
Object Initialization in Java

Timeline of an instance of class Bar

allocation of an instance of Bar

call to the constructor of Bar

call to the super constructor (Foo)

An instance of Bar

Object

Foo

Bar

uninitialized
Object Initialization in Java

Timeline of an instance of class Bar

allocation of an instance of Bar
call to the constructor of Bar
call to the super constructor (Foo)
call to the super constructor (Object)

An instance of Bar

Object Initialization in Java

Object

Foo

Bar

uninitialized
Object Initialization in Java

Timeline of an instance of class `Bar`
- allocation of an instance of `Bar`
- call to the constructor of `Bar`
- call to the super constructor (`Foo`)
- call to the super constructor (`Object`)
- initialization performed by `Object`
Object Initialization in Java

Timeline of an instance of class Bar

- allocation of an instance of Bar
- call to the constructor of Bar
- call to the super constructor (Foo)
- call to the super constructor (Object)
- initialization performed by Object
- initialization performed by Foo

An instance of Bar

- uninitialized
- being initialized
- initialized
Object Initialization in Java

Timeline of an instance of class Bar

- allocation of an instance of Bar
- call to the constructor of Bar
- call to the super constructor (Foo)
- call to the super constructor (Object)
- initialization performed by Object
- initialization performed by Foo
- initialization performed by Bar

An instance of Bar
Object Initialization in Java

Timeline of an instance of class Bar

- allocation of an instance of Bar
- call to the constructor of Bar
- call to the super constructor (Foo)
- call to the super constructor (Object)
- initialization performed by Object
- initialization performed by Foo
- initialization performed by Bar
- fully initialized
Object Initialization in Java

Timeline of an instance of class **Bar**

- Allocation of an instance of **Bar**
- Call to the constructor of **Bar**
- Call to the super constructor (**Foo**)
- Call to the super constructor (**Object**)
- Initialization performed by **Object**
- Initialization performed by **Foo**
- Initialization performed by **Bar**
- Fully initialized

Enforced by the BCV

Not enforced by the BCV

An instance of **Bar**
Example

The Class Loader

• A security sensitive class
  • Maps permissions to Class objects
  • Defines namespaces with the VM
  • Checks (with the security manager) that the code has the permission to access or define classes
  • Run with all permissions

• Invariant established by the constructor

  The caller has the permission CreateClassLoader

• Its methods rely on this invariant
public abstract class ClassLoader {
    protected ClassLoader() {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {sm.checkCreateClassLoader();}
    }
    protected final native void resolveClass(Class c);
}
Example

The Finalizer attack

```java
public abstract class ClassLoader {
    protected ClassLoader() {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {sm.checkCreateClassLoader();}
    }
    protected final native void resolveClass(Class c);
}
```
Example
The Finalizer attack

```java
public abstract class ClassLoader {
    protected ClassLoader() {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {sm.checkCreateClassLoader();}
    }

    protected final native void resolveClass(Class c);
}

class Attacker extends ClassLoader{
    Attacker(){super();} // this call will fail

    public static void main(String args[]){
        try{Attacker o = new Attacker();}
        ...
    }
}
```
Example
The Finalizer attack

```java
public abstract class ClassLoader {
    protected ClassLoader() {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {sm.checkCreateClassLoader();}
    }
    protected final native void resolveClass(Class c);
}

class Attacker extends ClassLoader {
    Attacker(){super();} // this call will fail

    public static void main(String args[]){
        try{Attacker o = new Attacker();}
        ...
    }

    protected void finalize(){
        this.resolveClass(...); // Attacker exploit!
    }
}
```
public abstract class ClassLoader {
    private volatile boolean initialized;

    protected ClassLoader() {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {sm.checkCreateClassLoader();}
        this.initialized = true;
    }

    protected final void resolveClass(Class c){
        if (!initialized) {
            throw new SecurityException("ClassLoader object not initialized");
        }
        ...
    }
}
Oracle's patch
Program your own monitor

```java
public abstract class ClassLoader {
    private volatile boolean initialized;

    protected ClassLoader() { 
        SecurityManager sm = System.getSecurityManager(); 
        if (sm != null) { sm.checkCreateClassLoader(); }
        this.initialized = true;
    }

    protected final void resolveClass(Class c) {
        if (!initialized) {
            throw new SecurityException(
                "ClassLoader object not initialized" );
        }...
    }
}
```

Lots of coding
Not automated
Failure to adhere leads to vulnerability
Solutions

• Our solution: a type system
  • Failure to adhere leads to a type error
  • No additional code
  • Only few type annotations are needed
  • Formally defined
  • Proved sound
Fähndrich and Leino's raw Types

• M. Fähndrich and R. Leino. Declaring and checking non-null types in an object-oriented language. In OOPSLA'03.

• Introduce raw types
  • $T_{\text{raw}}(S)$ denotes the partially initialized objects of type $T$ or subclass thereof, where $S$ is the lowest class frame initialized

• Proposed for a nullness type system
  • Fields declared in subclasses of $S$ may yield null even if declared non-null
The Type System

- Based on raw types but
- Applied to solve a security issue
- Formaly specified and proved sound
- Formalization as a type and effect system
- Type of local variables may evolve (flow sensitive)

\[
P \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ quad
The Type System

• Based on raw types but
  • Applied to solve a security issue
  • Formally specified and proved sound
• Formalization as a type and effect system
  • Type of local variables may evolve (flow sensitive)

\[
\frac{P}{\Gamma \vdash \text{ins} : L \rightarrow L'}
\]

• Typing rule for method calls

\[
c' = c.\text{super} \quad L(y) \sqsubseteq c'.\text{init.argtype} \\
L' = L[\text{this} \mapsto \text{Raw}(c')] \\
c.\text{init} \vdash \text{super}(y) : L \rightarrow L'
\]
Soundness

• Concrete semantics
  • Domains. E.g.: \( \emptyset = \text{Class} \times \text{Class}_\bot \times (\text{Field} \to \mathbb{V}) \)
  • Operational semantics.
    E.g.: 
    \[
    \frac{m.\text{instrs}[i] = x \leftarrow e \quad x \neq \text{this} \quad e \downarrow_{\rho,\sigma} v}{\langle m, i, \rho, \sigma, cs \rangle \Rightarrow \langle m, i+1, \rho[x \leftarrow v], \sigma, cs \rangle}
    \]

• Abstract semantics
  • Abstract domains
  • Analysis

• Correctness relation. E.g.: 
  \[
  \frac{\sigma(l) = [c, c, o]}{\sigma \vdash l : \text{Init}}
  \]
Soundness

• Concrete semantics
  • Domains. E.g.: $\emptyset = \text{Class} \times \text{Class}_\bot \times (\text{Field} \to \forall)$
  • Operational semantics.
    E.g.:
    \[
    \frac{m.\text{instrs}[i] = x \leftarrow e}{\langle m, i, \rho, \sigma, cs \rangle \Rightarrow \langle m, i+1, \rho[x \leftarrow v], \sigma, cs \rangle}
    \]
    \[\begin{align*}
x \neq \text{this} & \quad e \Downarrow_{\rho, \sigma} v
    \end{align*}\]

• Abstract semantics
• Abstract domains
• Analysis

• Correctness relation. E.g.:
\[
\frac{\sigma(l) = [c, c, o]}{\sigma \vdash l : \text{Init}}
\]
Experimental results

- Implementation includes dynamic features (*casts*)
- We studied 3 packages of Oracle's JRE: `java.lang`, `java.security` and `javax.security`
- We verified 377 classes (131 KLoc)
  - + Few annotations needed (53)
  - + Few runtime checks (4 cast operators)
  - - synthetic methods (3)
  - - limitation on arrays (1)
    - we forbid storing partially initialized objects in arrays
Experimental results

- Implementation includes dynamic features (*casts*)
- We studied 3 packages of Oracle's JRE: `java.lang`, `java.security`, and `javax.security`
- We verified 377 classes (131 KLOC)
  - + Few annotations needed (53)
  - + Few runtime checks (4 cast operators)
  - - Synthetic methods (3)
  - - Limitation on arrays (1)

  99% of classes verified
  1 annotation every 2.3 kloc on average

Works in practice
Initialization of Objects

Conclusion

• Only fully initialized objects should be accessed
  • Not enforced by the ByteCode Verifier (BCV)
  • Has led to exploits (privilege escalation using ClassLoader)

• Solution by Oracle: a coding pattern
  • Failure to adhere silently leads to vulnerability

• Our solution [ESORICS’10]
  • A type system, formally defined and proved sound
  • A machine checked soundness proof (D. PICHARDIE)
  • A prototype implementation of the checker
Outline

• Initialization of Classes
• Initialization of Objects
• Initialization of Fields
• Implementations
Field initialization

Problem 1: all fields are null by default
Solution 1: annotations are valid after construction
Field initialization

Problem 1: all fields are null by default
Solution 1: annotations are valid after construction

class C extends A{
    @NotNull O f;

    C(){
        this.f = new O();
    }

    static @NotNull O m(@NotNull C x){
        return x.f;
    }
}
Field initialization

Problem 1: all fields are null by default
Solution 1: annotations are valid after construction

class C extends A{
    @NotNull O f;

    C(){
        this.f = new O();
    }

    static @NotNull O m(@NotNull C x){
        return x.f;
    }
}
Field initialization

Problem 1: all fields are null by default
Solution 1: annotations are valid after construction

class C extends A{
    @NotNull O f;

    C(){
        this.f = new O();
    }

    static @NotNull O m(@NotNull C x){
        return x.f;
    }
}
Field initialization

Problem 1: all fields are null by default
Solution 1: annotations are valid after construction

class C extends A{
    @NotNull O f;

    C(){
        this.f = new O();
    }

    static @NotNull O m(@NotNull C x){
        return x.f;
    }
}
class C extends A{
    @NotNull O f;

    C(){
        this.f = new O();
    }

    static @NotNull O m(@NotNull C x){
        return x.f;
    }
}
Field initialization

Problem 1: all fields are null by default
Solution 1: annotations are valid after construction

class C extends A{
    @NotNull O f;

    C(){
        this.f = new O();
    }
}

static @NotNull O m(@NotNull C x){
    return x.f;
}
Field initialization

Problem 1: all fields are null by default
Solution 1: annotations are valid after construction

class C extends A{
    @NotNull O f;

    C(){
        this.f = new O();
    }
}

static @NotNull O m(@NotNull C x){
    return x.f;
}

allocation of C
call to C.<init>
call to A.<init>
initialization of A
initialization of C
(and f)
Field initialization

Problem 2: partially initialized objects may be accessed

class C extends A{
    @NotNull O f;

    C(){
        C.m(this);
        this.f = new O();
    }

    static @NotNull O m(@NotNull C x){
        return x.f;
    }
}

allocation of C
- call to C.<init>
- call to A.<init>

initialization of A

initialization of C

initialization of f
Field initialization

Problem 2: partially initialized objects may be accessed

class C extends A{
    @NotNull O f;
    C(){
        C.m(this);
        this.f = new O();
    }
}

static @NotNull O m(@NotNull C x){
    return x.f;
}

An object partially initialized is being accessed

Raw(A): annotations of A (and above) can be trusted
Fähndrich and Leino's Nullness Type System

• Introduce the notion of raw types for objects under construction

• Combine a type system and a data flow analysis
  • to ensure all non-null fields are initialized in their constructor

• Need annotations on method parameters, return values and fields
Fähndrich and Leino's Nullness Type System

• Introduce the notion of raw types for objects under construction

• Combine a type system and a data flow analysis
  • to ensure all non-null fields are initialized in their constructor

• Need annotations on method parameters, return values and fields

• Our contribution
  • Formalization
  • Unified framework (constraint based system)
  • Inference of all annotations
  • Proof of soundness
  • Proof of completeness w.r.t their type system
Abstract Domain

\[ \text{Value}^\# = \{ \text{Raw}(c) \mid c \in \text{Class} \} \cup \{ \text{Raw}, \text{NotNull}, \text{MaybeNull} \} \]

\[ \text{Locals}^\# = \text{Var} \rightarrow \text{Value}^\# \]

\[ \text{Heap}^\# = \text{Field} \rightarrow \text{Value}^\# \]

\[ \text{ThisVal}^\# = \text{Field} \rightarrow \{ \text{Init}, \text{MaybeUnInit} \} \]

Tracks the initialization of fields during object construction.
Example

class C extends A {
    C f,g;
    ...
    C(){
        super();
        this.f = new C();
        return;
    }
    ...
}"
Example

class C extends A {
    C f,g;
    ...
    C(){
        super();
        this.f = new C();
        return;
    }
    ...
    }

Method | pre | [Raw]
------ | ---- | ----
Heap
C.f  | ⊥    |    
C.g  | ⊥    |    

38
class C extends A {
    C f, g;
    ...
    C() {
        super();
        this.f = new C();
        return;
    }
    ...
}
```java
class C extends A {
    C f,g;
    ...
    C()
    {
        super();
        this.f = new C();
        return;
    }
    ...
}
```

<table>
<thead>
<tr>
<th>Method</th>
<th>pre</th>
<th>[Raw]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>C.f</td>
<td>⊥</td>
</tr>
<tr>
<td></td>
<td>C.g</td>
<td>⊥</td>
</tr>
<tr>
<td>This</td>
<td>C.f</td>
<td>MBUnInit</td>
</tr>
<tr>
<td></td>
<td>C.g</td>
<td>MBUnInit</td>
</tr>
<tr>
<td>Locals</td>
<td>this</td>
<td>Raw(A)</td>
</tr>
</tbody>
</table>
class C extends A {
    C f, g;
    ...
    C() {
        super();
        this.f = new C();
        return;
    }
    ...
}
Example

class C extends A {
    C f,g;
    ...
    C(){
        super();
        this.f = new C();
        return;
    }
    ...
}
Constraint Based System

• Complete-program inter-procedural analysis
• General form of constraints

\[
\frac{f(M^#, H^#, T^#, L^#) \subseteq (M^#, H^#, T^#, L^#)}{M^#, H^#, T^#, L^# \vdash (m, i) : ins}
\]
Constraint Based System

- Complete-program inter-procedural analysis
- General form of constraints

\[
f(M^#, H^#, T^#, L^#) \subseteq (M^#, H^#, T^#, L^#) \\
M^#, H^#, T^#, L^# \vdash (m, i) : \text{ins}
\]

- Example of constraint

\[
x = this \land f \in \text{class}(m).\text{fields} \\
T^#(m, i)[f \mapsto \text{Init}] \subseteq T^#(m, i + 1) \\
L^#(m, i) \subseteq L^#(m, i + 1) \\
\llbracket e \rrbracket^# \subseteq H^#(f) \\
M^#, H^#, T^#, L^# \vdash (m, i) : x.f \leftarrow e
\]
Initialization of Fields

Conclusion

• Non-null annotations are useful
  • Annotating a program is not trivial

• We propose a fully automatic analysis [FMOODS'08]
  • Using a unified framework
  • Proved sound
  • Machine checked soundness proof (D. PICHARDIE)

• The analysis is proved complete w.r.t. Fähndrich and Leino's type system
Initialization of Fields

Conclusion

• Non-null annotations are useful
  • Annotating a program is not trivial

• We propose a fully automatic analysis [FMOODS'08]
  • Using a unified framework
  • Proved sound
  • Machine checked soundness proof (D. PICHARDIE)

• The analysis is proved complete w.r.t. Fähndrich and Leino's type system
Outline

• Initialization of Classes
• Initialization of Objects
• Initialization of Fields
• Implementations
Formal specification and proof

Proved on a toy language

Soundness

Prototype

Java bytecode: JVM spec compliant

Precision

Exceptions

Lazy class loading

Object initialization

Interfaces

...
Formal specification and proof → Prototype

Java bytecode: JVM spec compliant
Exceptions
Lazy class loading
Object initialization
Interfaces...
Formal specification and proof

Prototype

a backend: the SAWJA framework

Java bytecode: JVM spec compliant

Exceptions
Lazy class loading
Object initialization
Interfaces
...
Overview

SAWJA

Javalib

Static Analysis oriented (IR, CFG computation, etc.)

.file oriented (parsing, etc.)

.class
Overview

- Eclipse Plugin
- NIT
- SAWJA
- Javalib

Nullability Inference Tool, [FMOODS'08], [PASTE'08]

.class file oriented (parsing, etc.)

Static Analysis oriented (IR, CFG computation, etc.)
Overview

- Eclipse Plugin
  - NIT
  - Javalib
  - SAWJA
  - SecInit

- Secure object initialization, [ESORICS'10]
- Nullability Inference Tool, [FMOODS'08], [PASTE'08]
- Used by others (Frama-C / CEA, D. Gurov RIT Stockholm, Secure cloning, etc.)
- Static Analysis oriented (IR, CFG computation, etc.)

Used by others
- (Frama-C / CEA, D. Gurov RIT Stockholm, Secure cloning, etc.)

NULLABILITY INFERENCE TOOL, [FMOODS'08], [PASTE'08]
SAWJA (Static Analysis Workshop for JAva)

- OCaml library for developing Java bytecode static analyses
- High level intermediate representation
  - Transformation proved sound (by D. Demange and D. Pichardie)
- High level API for efficient browsing of class hierarchy
  - Did you ever look at the method resolution specification?
    - Implements a large part of the JVM Specification (structural constraints, resolution, lookups, control flow, etc.)
- Efficient
- SAWJA+Javalib: 28.2 kloc
**NIT (Nullability Inference Tool)**

- Precision: safe dereferences
  - Intra: intra-procedural analysis
  - MAN: with method annotations
  - FAN: with field annotations
  - NIT: with NIT
- Benchmarks: average over 18 programs (Soot, JDTCore, Jess, ESC/Java, Julia, JavaCC, etc.)

[Bar chart showing precision values for Intra, MAN, FAN, and NIT]
The Eclipse Plugin
NIT
The Secure Initialization Checker in Numbers

• Based on SAWJA+Javalib (28.2 kloc)
• SecInit Prototype: Java compliant but only 1.3 kloc
  • Core analysis: 636 loc
    • Constraints: 288 loc (114 casts)
    • Variance checking, error handling, printing functions
• Efficient (~200 methods / second)
Outline

• Initialization of Classes
• Initialization of Objects
• Initialization of Fields
• Implementations
• Conclusions and Perspectives
Conclusion

Formal analyses

• Improve understanding of initialization in Java

• Formal analysis proposed
  • Nullness analysis with focus on instance fields [FMOODS'08]
    • Proved sound and complete w.r.t. Fähndrich's type system
  • Type system for secure object initialization [ESORICS'10]
    • Offers a sound solution to an actual security issue
  • Class Initialization [BYTECODE'09]
    • Improves the precision of the control flow graph
    • Allows proving static fields are initialized before being read
Conclusion

Implementations

• **SAWJA**: efficient library, 28 kloc, 1.5 man-year, LGPL [FOVEOOS'10] (with V. Monfort, N. Barré, D. Demange, T. Turpin, D. Pichardie, etc.)

• **NIT**: 9.3 kloc, 8 man-month, GPL [PASTE'08]
  • Scalable (analyses Soot (49,810 methods) within 2 min)

• **NIT/Eclipse** (with N. Barré, demonstrated at [JAVAONE'09])
  • Integration in an actual development environment

• **SeclInit** (presented at [FMCO'10])
  • Based on **SAWJA**: 1.3 kloc, 200 methods / second
  • 99% classes proved sound with 1 annotation every 2.3 kloc
Perspectives

- Generics & Arrays (Secure object initialization and nullness)
- Generics as guides for context (object) sensitivity
- Arrays are much more dynamic than objects (aliasing issues)
  - field names are runtime indices
  - the number of fields is dynamic
  - static types do not help

```java
Integer[] ai = new Integer[2];
Object[] ao = (Object[])ai;
ao[0] = new Integer(1);
ao[1] = "";
```
Perspectives

• NIT and the Eclipse plugin: difficulties to understand inferred annotations
  • Keep more information to navigate results backward
  • Virtual calls: how to choose a particular call?
  • Infeasible paths: how to integrate user feedback?

• SAWJA
  • Context sensitive analyses
    • Dependency: does it include the context?
  • Extract some algorithms from proofs
    • IR construction, control flow algorithms (RTA, XTA, etc.)
Foundations and implementation of a tool bench for static analysis of Java bytecode programs

Laurent Hubert
CNRS/IRISA - Université de Rennes 1
EPI Lande/Celtique

December 17, 2010
PhD advisors: T. Jensen et D. Pichardie