Outline

- Introduction
- How does Soot work: The Soot Representation
  - From Bytecode to 3-address code (Jimple)
  - 3-address code analyses and optimizations
  - From 3-address code to Bytecode: feasible
- Understanding Jimple
- Analyzing Jimple
Java .class files

- Contain fields, methods and attributes
- Fields: instance variables or class variables
- Methods: contain Java bytecode

```java
// java source
int cc (int x, int y) {
    int z;
    z = x*y;
    return z; }
```

```java
// bytecode
Method int cc (int, int) {
    0 iload 1
    1 iload 2
    2 imul
    3 istore 3
    4 iload 3
    5 ireturn }
```

Intermediate Representations

- Bytecode vs. 3-address code

**Bytecode:**
1. Each instruction has implicit effect on stack
2. No types for local variables
3. > 200 kinds of insts

**Typed 3-address code:**
1. Each stmt acts explicitly on named variables
2. Types for each local variable
3. Only 15 kinds of stmt

Do analysis on JIMPLE 3-address code IR.
Intermediate Representations

- Source vs. 3-address code

Source
1. Irregular structure (somewhat)
2. Complex statements and expressions

3-address code:
1. More regular structure
2. 15 kinds of stmts, simple expressions and statements

Analysis is simpler and more effective on JIMPLE 3-address code than source!

Overview of Soot

Class files

JIMPLIFY

ANALYSIS/OPTIMIZATION

Optimized jimple

Grimp IR

Baf IR

Class files
Bytecode to Jimple

- **Step 1: Naïve jimple**
  - Introducing named variables:
    - iload 0 \rightarrow $s0 = u0$
  - Still not easy to analyze
- **Step 2: Typed jimple**
  - Assigning types to local variables
  - Code is verbose and still not easy to analyze
- **Step 3: Cleanup**
  - Elimination of unused variables, other simplifications
  - Clean JIMPLE, ready to optimize

Advantages of Jimple and Soot

- **JIMPLE**
  - Typed local variables
  - Simple expressions (1 operator / stmt)
- **SOOT**
  - Uses and defs are easily available
  - Soot provides data-flow analysis framework
  - Hierarchy information available
  - Can construct call graph
Analysis and Optimizations on Jimple

- Virutal call resolution
- Method Inlining
- Side effect analysis
- Partial redundancy elimination
- Object inlining
- Synchronization removal & stack allocation
- Decompiling Java bytecode
- Array bounds check elimination

Jimple to Bytecode

- Need to be efficient: not trivial
- There are two ways:
  - Going through the Baf IR (low-level stack based)
  - Going through the Grimp IR (high-level closer to source)
- Bottom line from empirical evaluation:
  - It can be done efficiently!
    - Without optimization: programs run in comparable time
    - With optimizations: programs run faster
Outline

- Introduction
- How does Soot work: The Soot Representation
  - From Bytecode to 3-address code (Jimple)
  - 3-address code analyses and optimizations
  - From 3-address code to Bytecode: feasible
- Understanding Jimple
- Analyzing Jimple

Understanding Jimple

- Run soot: java soot.Main –jimple MyClass

```java
public class A extends java.lang.Object {
    public void <init>() {
        A r0;
        r0 := @this: A;
        specialinvoke r0.<java.lang.Object: void <init>()>();
        return; }

    public void m() {
        A a = new A();
        a.m();
    }
}
```

...
public class A {
    main(String[] args) {
        A a = new A();
        a.m();
    }
    public void m() {
    }
}

...
Different Kinds of Calls

- Constructor Call:
  \[ A \ a = \text{new} \ A(); \]
  \[ s1 = \text{new} \ A; \]
  specialinvoke \( s1. <A: \text{void} \ <\text{init}>()>()(); \]

- Virtual Call:
  a.m();
  virtualinvoke r2.\( <A: \text{void} \ m()>()(); \)

- Static Call:
  sm();
  staticinvoke \( <A: \text{void} \ sm()>()(); \)

- Interface Call:
  x.m();
  interfaceinvoke r0.\( <\text{pack2.X: void} \ m()>()(); \)

Analyzing Jimple

- E.g., We need to know information about calls
- Basic Soot Classes: Scene, SootClass, SootMethod, Body, JimpleBody, Local, Unit
- Analysis steps:
  - Step 1: Load JIMPLEs into memory:
    - Iterate over each class in the program
    - Iterate over each method and load the jimple body
  - Step 2: Analyze the jimples in memory
  - Step 3: (later) Instrument jimple and write back to bytecode
Analyzing Jimple

- E.g., Filter out call statements

```java
for (Iterator uIt = m.getActiveBody().getUnits().iterator();
    uIt.hasNext();) { // iterate over all statements
    Stmt s = (Stmt) uIt.next(); // get next statement
    if (s.containsInvokeExpr()) { // is it a call?
        ... } // if call, then do something
```

Analyzing Jimples, cont.

- E.g., What kind of call?

```java
InvokeExpr call = (InvokeExpr) s.getInvokeExpr(); // get a soot object that represents the call expression

if (call instanceof StaticInvokeExpr ||
    call instanceof SpecialInvokeExpr) {
    ... }
if (call instanceof VirtualInvokeExpr ||
    call instanceof InterfaceInvokeExpr) {
    } // Do something else
```
Other Kinds of Stmts?

if (s instanceof IfStmt) {...} // is it an If Stmt?

if (s instanceof GotoStmt) {...} // is it a Goto Stmt?

Other kinds of stmt, do something different with them.

We are building the Loader driver.