CS 6301.007 Machine Learning in Cyber Security – Understanding the Program Representations

Wei Yang

Department of Computer Science
University of Texas at Dallas

Outline

- Overview
- Static Program Representations
  - Abstract Syntax Tree
  - Control Flow Graph
  - Program Dependence Graph
  - Call Graph
- Static Program Analysis Tools
  - AspectJ
  - TraceMatches
  - Soot
- Summary

Outline

- Overview
- Static Program Representations
  - Abstract Syntax Tree
  - Control Flow Graph
  - Program Dependence Graph
  - Call Graph
- Static Program Analysis Tools
  - AspectJ
  - TraceMatches
  - Soot
- Summary

What is Program Analysis?

- Body of work to discover useful facts about programs

- Broadly classified into three kinds:
  - Dynamic (execution-time)
  - Static (compile-time)
  - Hybrid (combines dynamic and static)

Dynamic Program Analysis

- Infer facts of program by monitoring its runs

- Examples:

Array bound checking *Purify*	Datarace detection *Eraser*
Memory leak detection *Valgrind*	Finding likely invariants *Daikon*

Static Analysis

- Infer facts of the program by inspecting its source (or binary) code

- Examples:

Suspicious error patterns
Lint, FindBugs, Coverity

Memory leak detection
Facebook Infer

Checking API usage rules
Microsoft SLAM

Verifying invariants
ESC/Java

QUIZ: Program Invariants

An invariant at the end of the program is (z == c) for some constant c. What is c?

```
int p(int x) { return x * x; }

void main() {
    int z;
    if  (getc() == ‘a’)
        z = p(6) + 6;
    else
        z = p(-7) – 7;
                                z = ?


}
```

QUIZ: Program Invariants

An invariant at the end of the program is (z == c) for some constant c. What is c?

```
int p(int x) { return x * x; }

void main() {
    int z;
    if  (getc() == ‘a’)
        z = p(6) + 6;
    else
        z = p(-7) – 7;

                        z = 42

     if (z != 42)
         disaster();
}
```

Disaster averted!

Discovering Invariants By Dynamic Analysis

Finite number of executions vs. unbounded number of paths

(z == 42) *might be* an invariant

(z == 30) is *definitely not* an invariant

```
int p(int x) { return x * x; }

void main() {
    int z;
    if  (getc() == 'a')
        z = p(6) + 6;
    else
        z = p(-7) – 7;

                              z = 42

     if (z != 42)
           disaster();
}
```

Discovering Invariants By Dynamic Analysis

is definitely

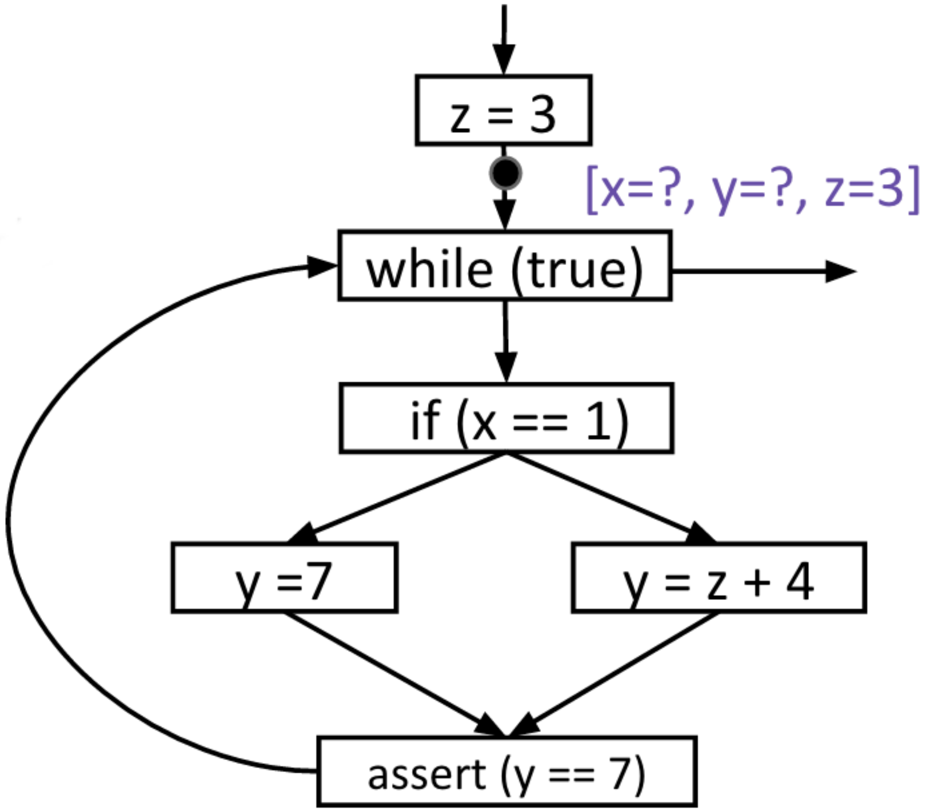
(z == 42) ~~*might be*~~ an invariant

(z == 30) is *definitely not* an invariant

```
int p(int x) { return x * x; }

void main() {
    int z;
    if  (getc() == 'a')
        z = p(6) + 6;
    else
        z = p(-7) – 7;

                                  z = 42

     if (z != 42)
           disaster();
}
```

In a Nutshell

- Control-flow graph
- Abstract vs. concrete states
- Termination
- Completeness
- Soundness

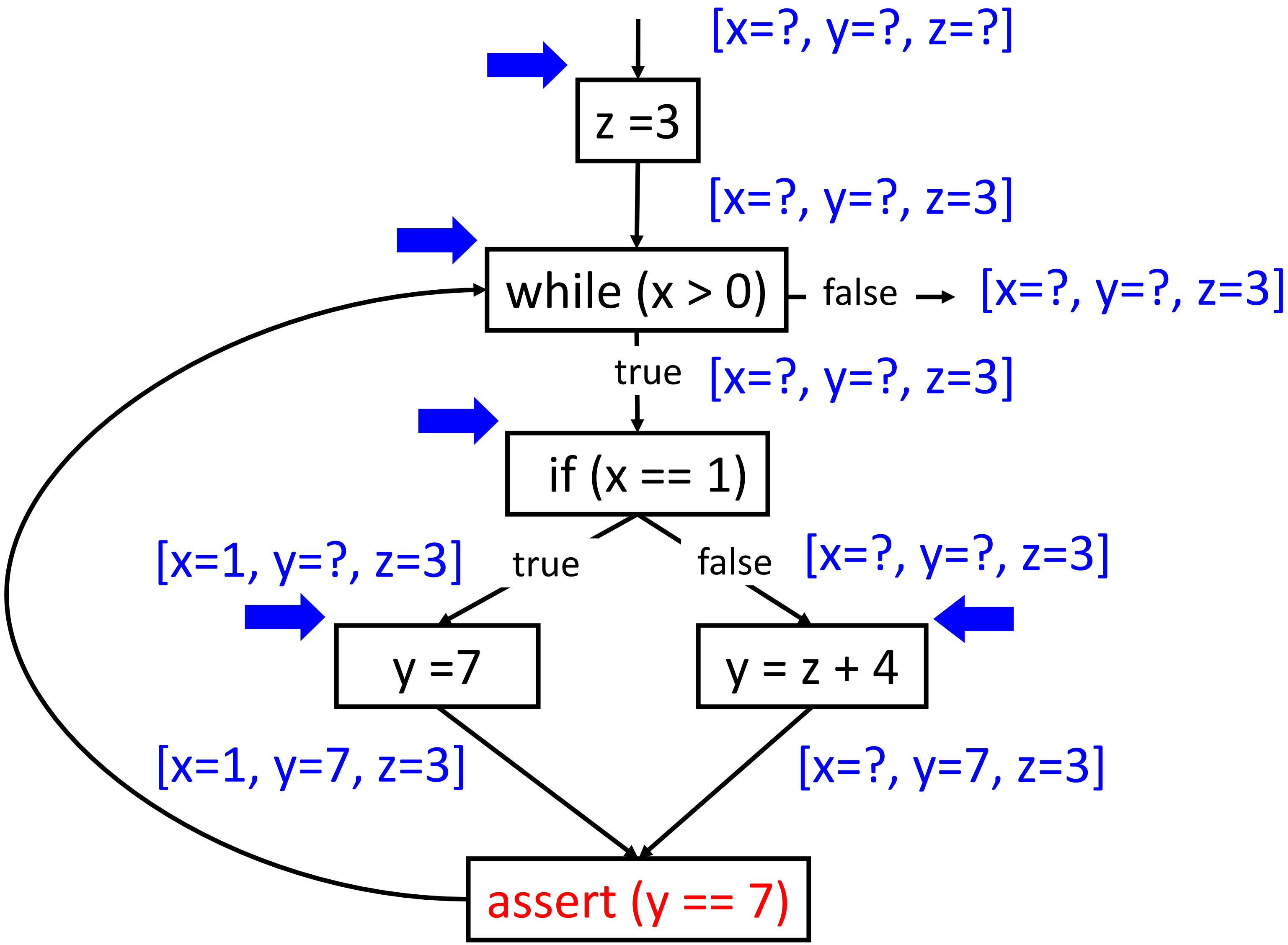


Example Static Analysis Problem

- Find variables that have a constant value at a given program point

```
void main() {
    z = 3;
    while (true) {
        if (x == 1)
            y = 7;
        else
            y = z + 4;
        assert (y == 7);
    }
}
```

Iterative Approximation

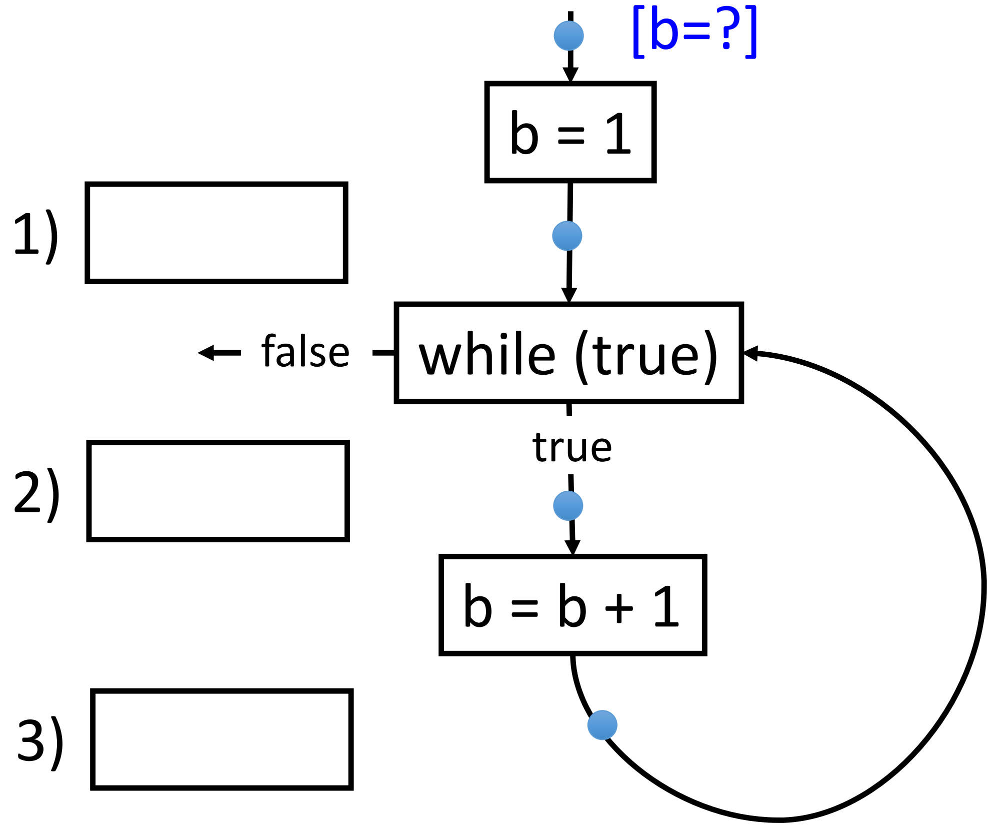


QUIZ: Iterative Approximation

- Fill in the value of variable b that the analysis infers at:

  1) the loop header
  2) entry of loop body
  3) exit of loop body

Enter “?” if a definite value cannot be inferred.

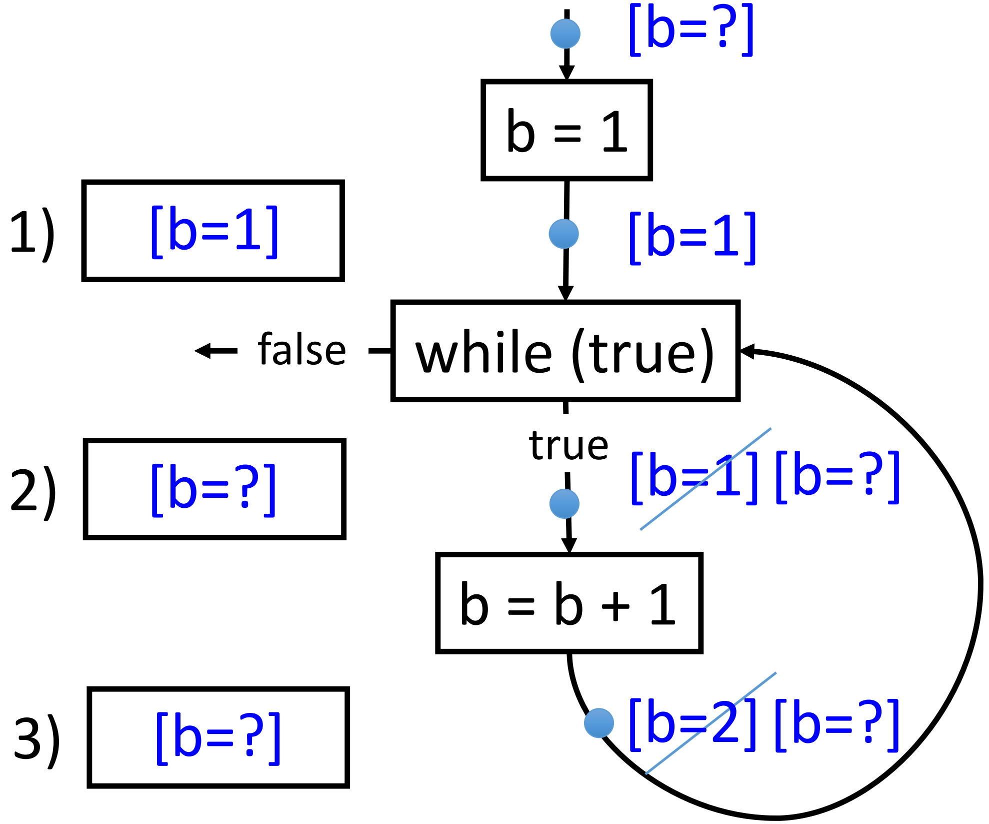


QUIZ: Iterative Approximation

- Fill in the value of variable b that the analysis infers at:

  1) the loop header
  2) entry of loop body
  3) exit of loop body

Enter “?” if a definite value cannot be inferred.



QUIZ: Dynamic vs. Static Analysis

Match each box with its corresponding feature.

	Dynamic	Static
Cost		
Effectiveness		

A. Unsound (may miss errors)

B. Proportional to program's execution time

C. Proportional to program's size

D. Incomplete (may report spurious errors)

QUIZ: Dynamic vs. Static Analysis

Match each box with its corresponding feature.

	Dynamic	Static
Cost	B. Proportional to program's execution time	C. Proportional to program's size
Effectiveness	A. Unsound (may miss errors)	D. Incomplete (may report spurious errors)

Undecidability of Program Properties

- Can program analysis be sound and complete?
  - Not if we want it to terminate!

- Questions like “is a program point reachable on some input?” are undecidable

- https://en.wikipedia.org/wiki/Undecidable_problem

- Undecidability => program analysis cannot ensure termination + soundness + completeness

Outline

- Overview
- Static Program Representations
  - Abstract Syntax Tree
  - Control Flow Graph
  - Program Dependence Graph
  - Call Graph
- Static Program Analysis Tools
  - AspectJ
  - TraceMatches
  - Soot
- Summary

Outline

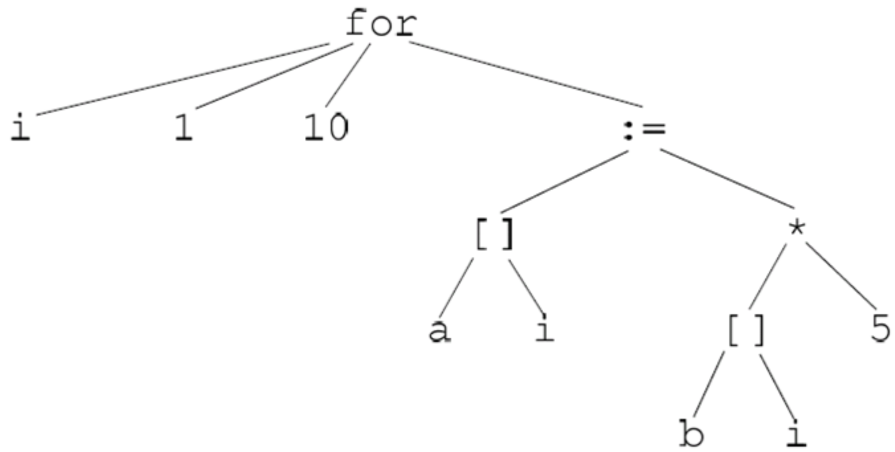
- Overview
- Static Program Representations
  - Abstract Syntax Tree
  - Control Flow Graph
  - Program Dependence Graph
  - Call Graph
- Static Program Analysis Tools
  - AspectJ
  - TraceMatches
  - Soot
- Summary

Abstract syntax tree

- AST
  - An abstract syntax tree (AST) is a finite, labeled, directed tree, where the internal nodes are labeled by operators, and the leaf nodes represent the operands of the operators.

```
for i: = 1 to 10  do
        a[i]: = b[i] * 5;
end
```

AST:



Abstract syntax tree

- ASTs are widely used in compilers (e.g., gcc) when parsing source code.

- ASTs are abstract
  - They don’t contain all information in the program
    - E.g., spacing, comments, brackets, parentheses
  - AST has many similar forms
    - e.g., for, while, repeat...until
  - ASTs are not good for binary code

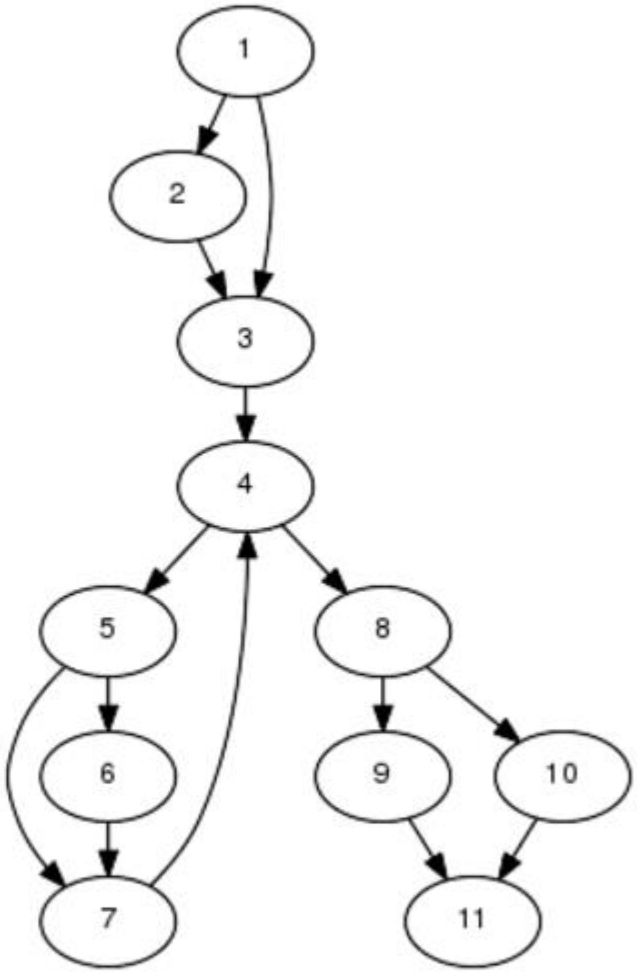
- AST only reflect the syntax of the program under analysis

Outline

- Overview
- Static Program Representations
  - Abstract Syntax Tree
  - Control Flow Graph
  - Program Dependence Graph
  - Call Graph
- Static Program Analysis Tools
  - AspectJ
  - TraceMatches
  - Soot
- Summary

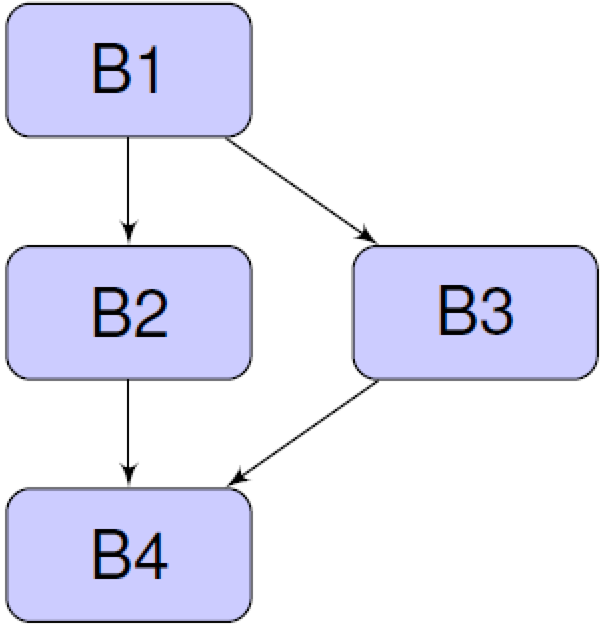
Control Flow Graph

- A directed graph where
  - Each node represents a statement
  - Edges represent control flow



Control Flow Graph

- CFG consists of
  - A maximal sequence of consecutive instructions such that inside the basic block an execution can only proceed from one instruction to the next
  - Edges represent potential flow of control between BBs



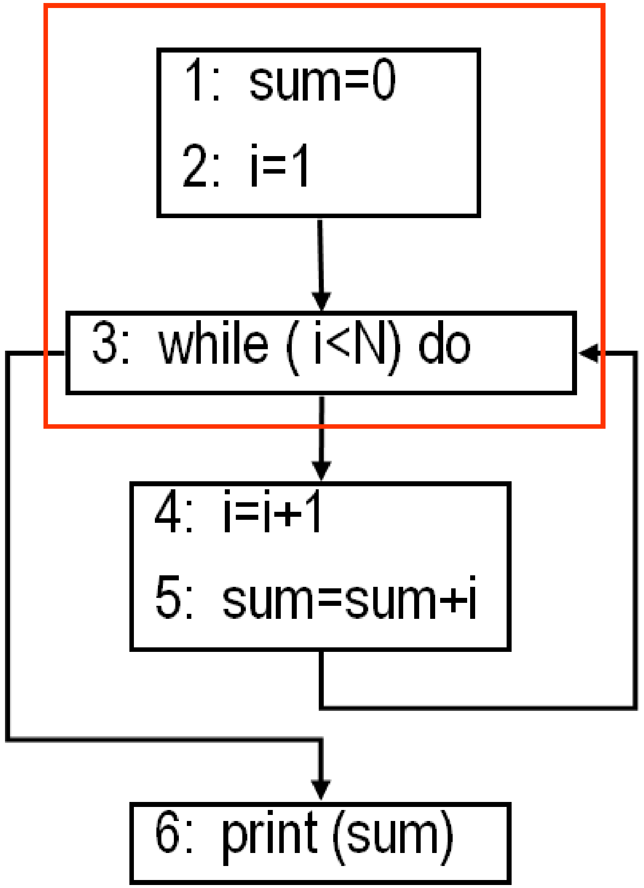
CFG = <V, E, Entry, Exit>

- V = Vertices, nodes (BBs)
- E = Edges; potential flow of control
  - $E \in V \times V$
- Entry; Exit $\in$ V;
  - unique entry and exit

An Example of CFG

- BB- A maximal sequence of consecutive instructions such that inside the basic block an execution can only proceed from one instruction to the next.

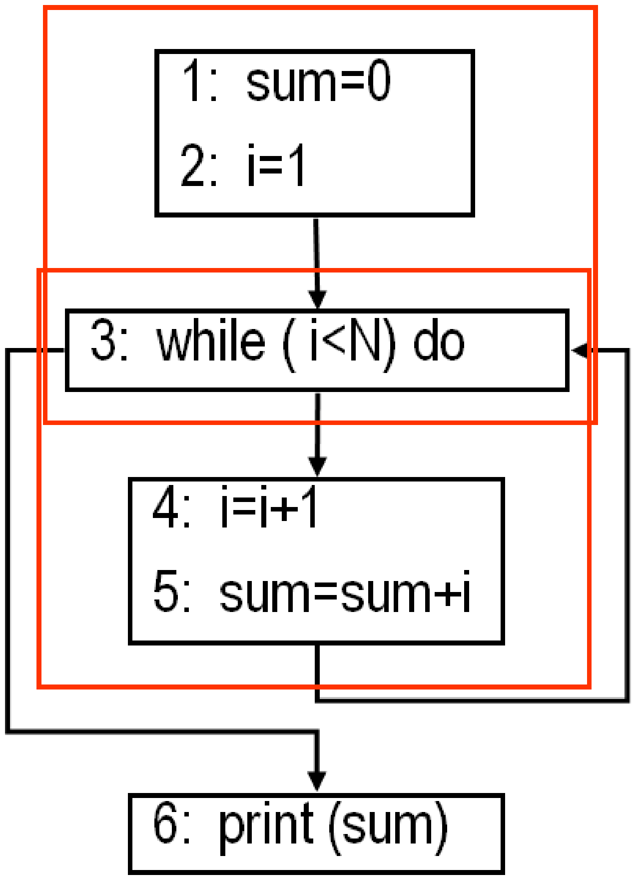
```
1: sum=0
2: i=1
3: while (i<N) do
4: i=i+1
5: sum=sum+i
     endwhile
6: print(sum)
```



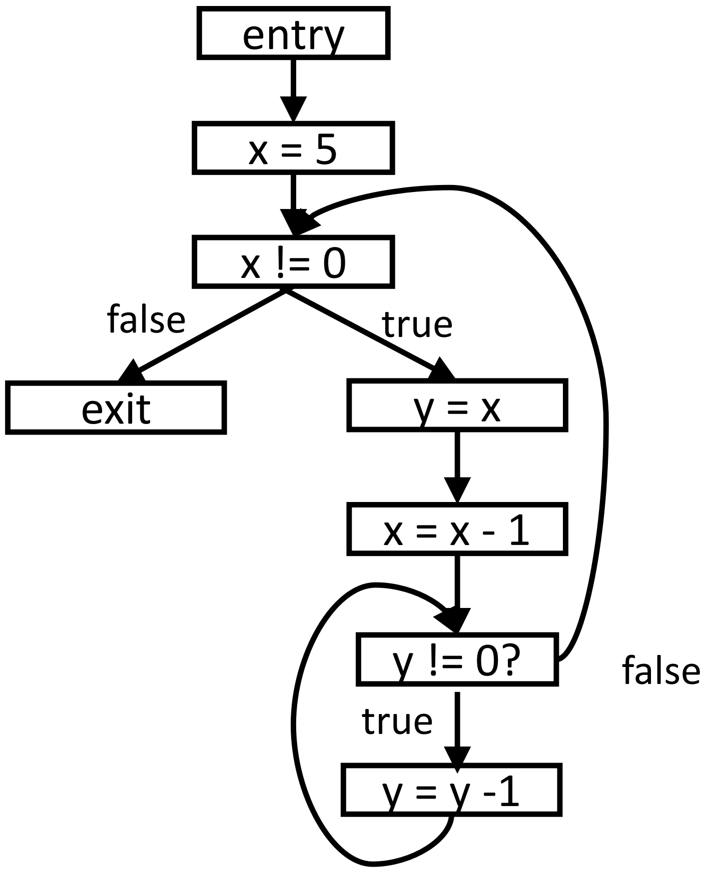
An Example of CFG

- BB- A maximal sequence of consecutive instructions such that inside the basic block an execution can only proceed from one instruction to the next.

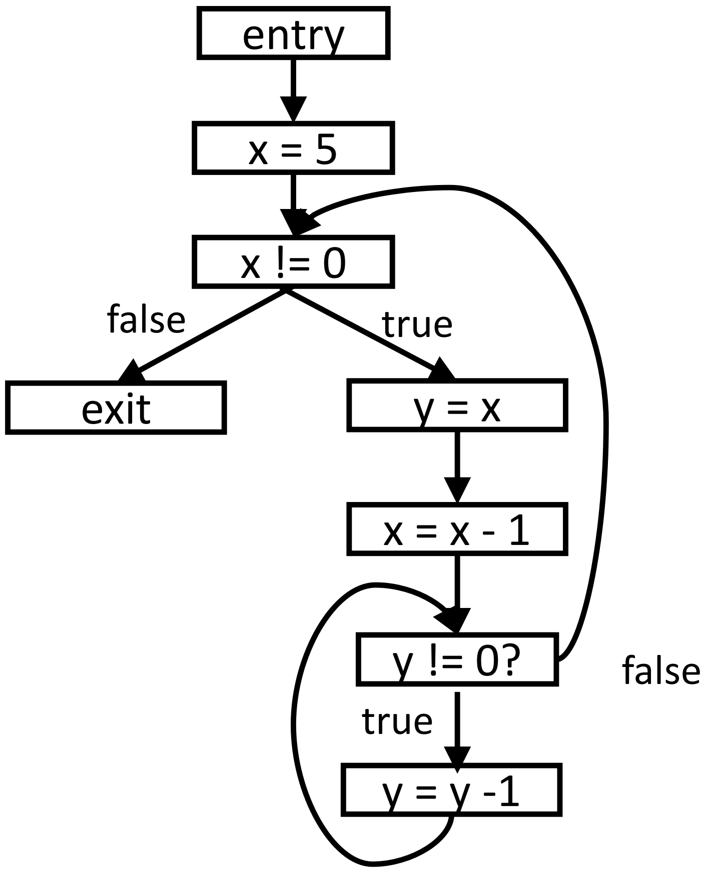
1: sum=0
2: i=1
3: while (i<N) do
4: i=i+1
5: sum=sum+i
    endwhile
6: print(sum)



QUIZ: Control-Flow Graphs



QUIZ: Control-Flow Graphs



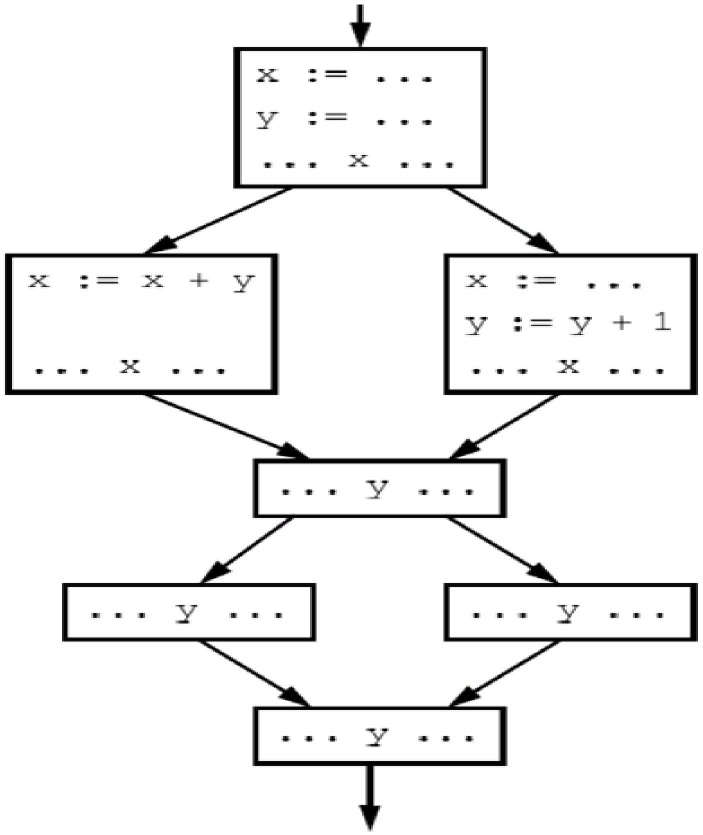

```
x = 5;
while (x != 0) {
  y = x;
  x = x - 1;
  while (y != 0) {
    y = y - 1
  }
}
```

Outline

- Overview
- Static Program Representations
  - Abstract Syntax Tree
  - Control Flow Graph
  - Program Dependence Graph
  - Call Graph
- Static Program Analysis Tools
  - AspectJ
  - TraceMatches
  - Soot
- Summary

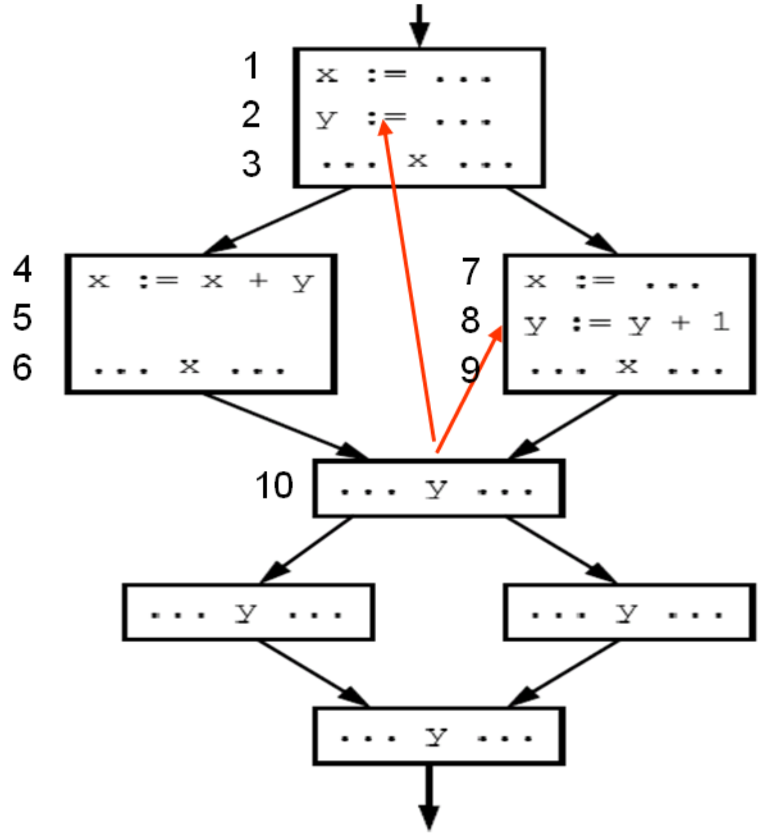
Program Dependence Graph: Data Dependency

- S data depends on T if there exists a control flow path from T to S and a variable is defined at T and then used at S.



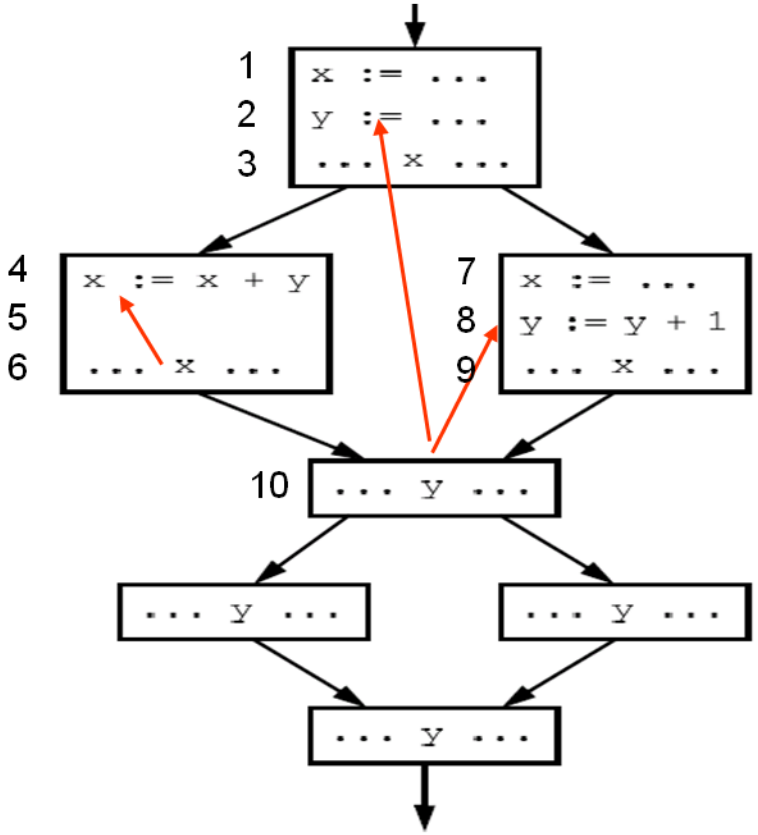
Program Dependence Graph: Data Dependency

- S data depends on T if there exists a control flow path from T to S and a variable is defined at T and then used at S.



Program Dependence Graph: Data Dependency

- S data depends on T if there exists a control flow path from T to S and a variable is defined at T and then used at S.



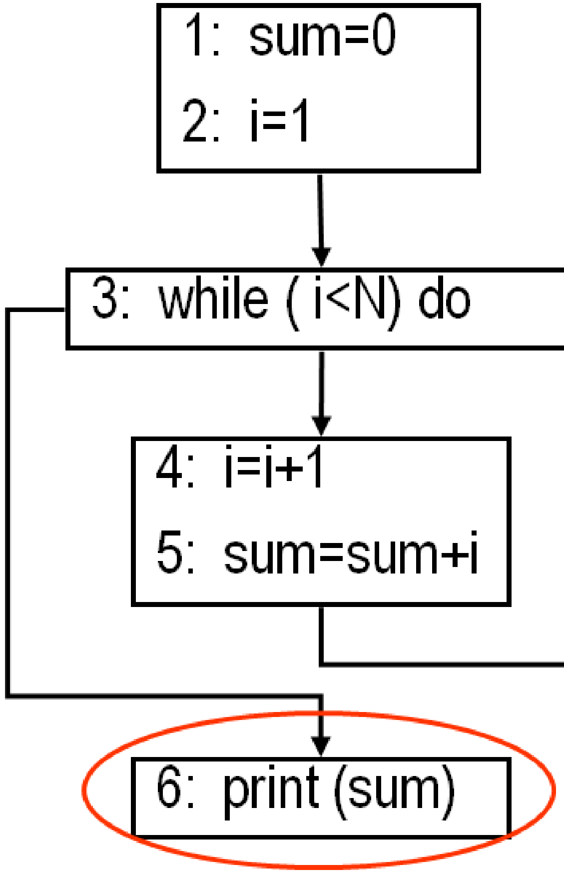
Program Dependence Graph: Dominator

- Dominator
  - A block M dominates a block N if every path from the entry that reaches block N has to pass through block M.
    - By definition, every node dominates itself.
    - The entry block dominates all blocks

- Immediate Dominator
  - A block M immediately dominates block N if M dominates N, and there is no intervening block P such that M dominates P and P dominates N.
    - In other words, M is the last dominator on all paths from entry to N.
    - Not all blocks have immediate dominators (e.g. entry block).

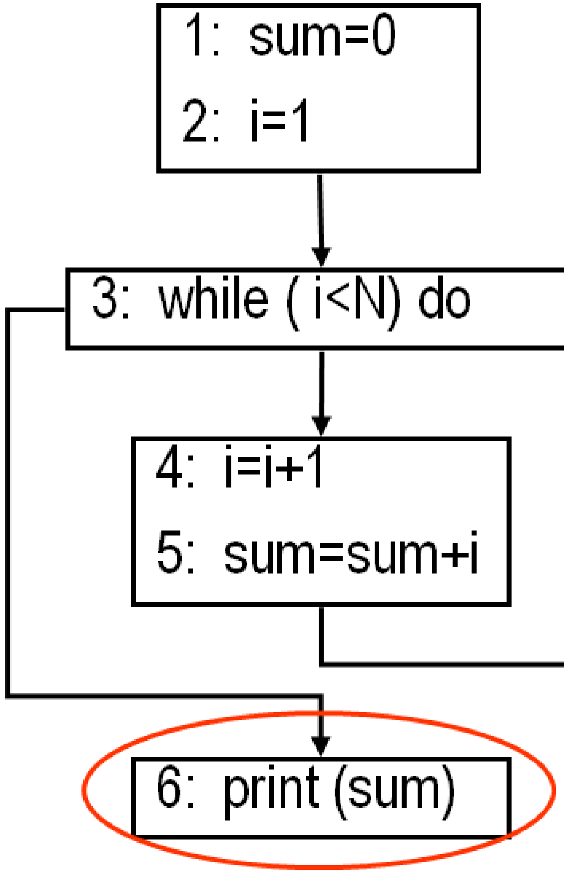
Dominator and I-Dominator Examples

1: sum=0
2: i=1
3: while (i<N) do
4: i=i+1
5: sum=sum+i
    endwhile
6: print(sum)



Dominator and I-Dominator Examples

1: sum=0
2: i=1
3: while (i<N) do
4: i=i+1
5: sum=sum+i
endwhile
6: print(sum)



DOM(6)={1,2,3,6}, IDOM(6)=3

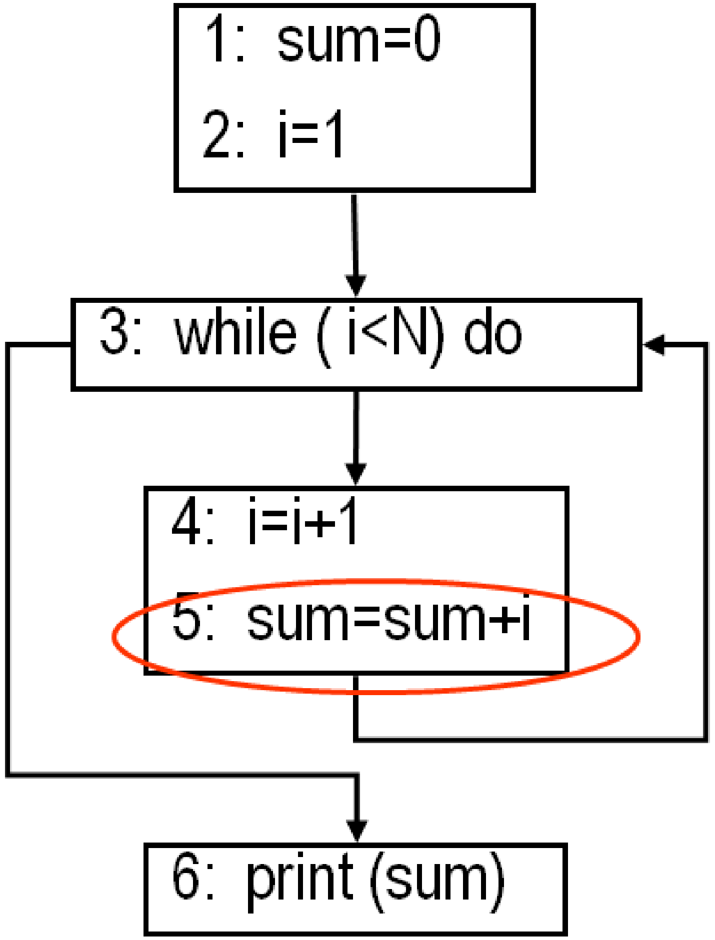
Program Dependence Graph: Post Dominator

- Post-Dominator
  - In the reverse direction, block M post-dominates block N if every path from N to the exit has to pass through block M.
    - The exit block post-dominates all blocks.

- Immediate Post-Dominator
  - It is said that a block M immediately post-dominates block N if M post-dominates N, and there is no intervening block P such that M post-dominates P and P post-dominates N. In other words, M is the last post-dominator on all paths from entry to N.

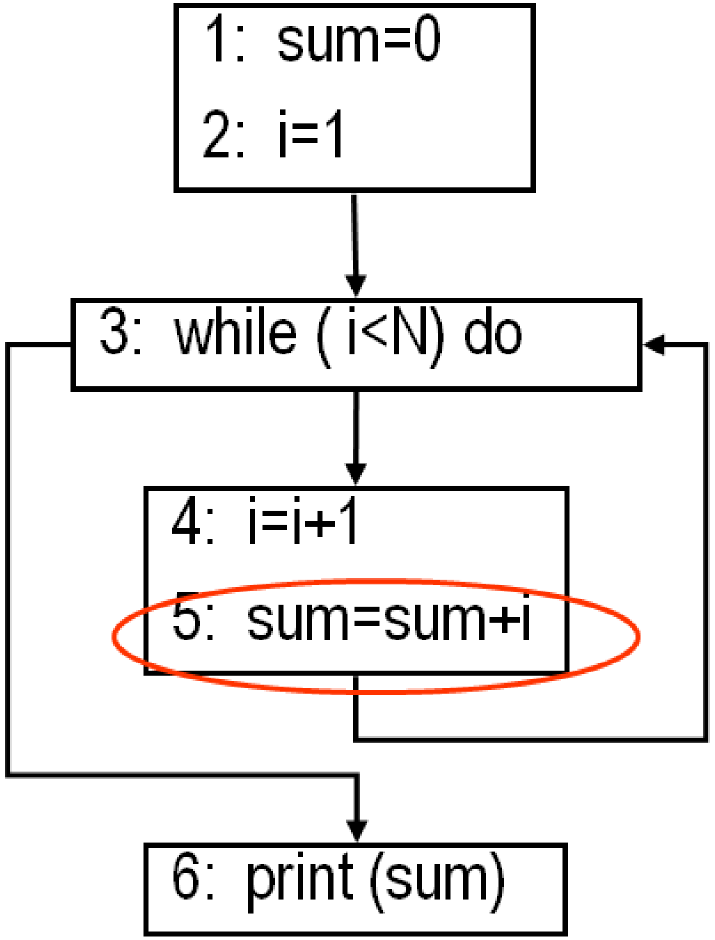
Dominator and I-Dominator Examples

1: sum=0
2: i=1
3: while (i<N) do
4: i=i+1
5: sum=sum+i
    endwhile
6: print(sum)



Dominator and I-Dominator Examples

1: sum=0
2: i=1
3: while (i<N) do
4: i=i+1
5: sum=sum+i
    endwhile
6: print(sum)

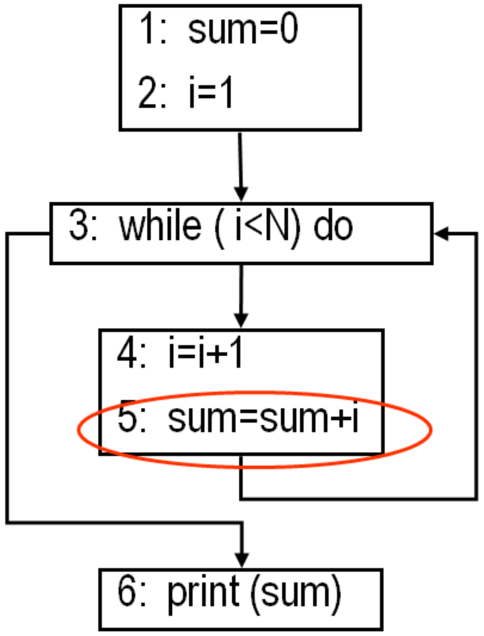


PDOM(5)={3,5,6} IPDOM(5)=3

Program Dependence Graph: Control Dependence

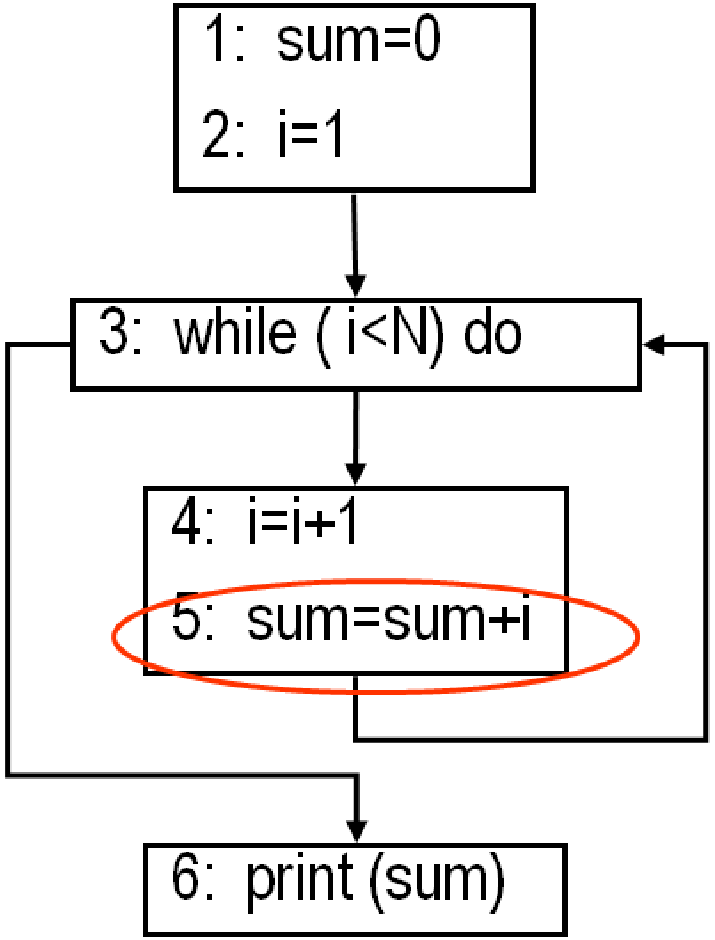
- A node (basic block) Y is control-dependent on another X iff X directly determines whether Y executes
  - there exists a path from X to Y s.t. every node in the path other than X and Y, is post-dominated by Y
  - X is not strictly post-dominated by Y (if $X \neq Y, Y$ does not postdominate X).

```
1: sum=0
2: i=1
3: while (i<N) do
4: i=i+1
5: sum=sum+i
   endwhile
6: print(sum)
```



Control Dependence Examples

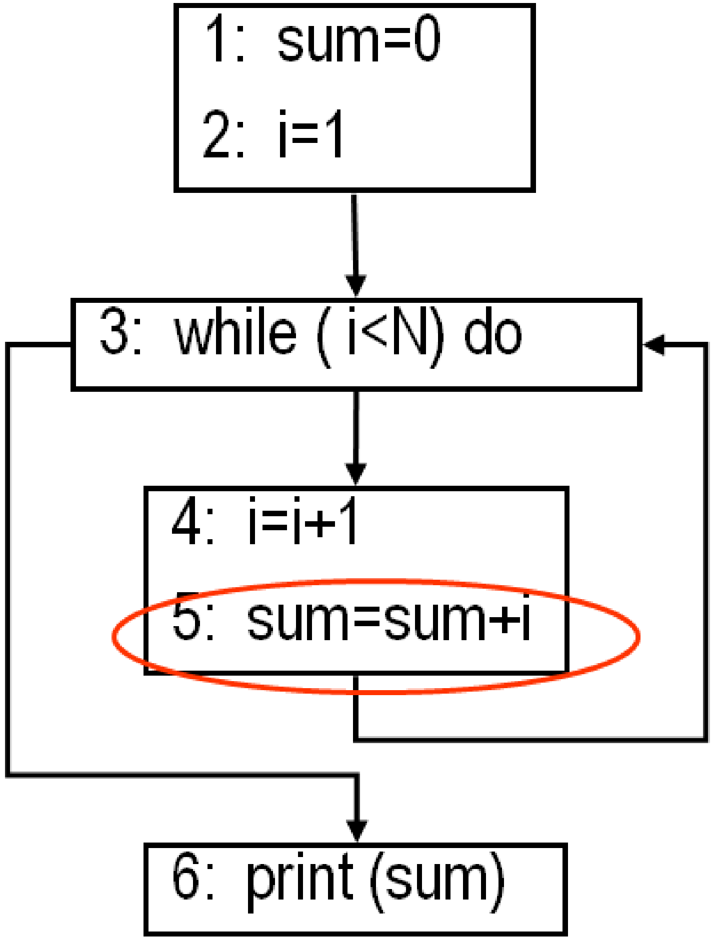
1: sum=0
2: i=1
3: while (i<N) do
4: i=i+1
5: sum=sum+i
endwhile
6: print(sum)



Control Dependence Examples

1: sum=0
2: i=1
3: while (i<N) do
4: i=i+1
5: sum=sum+i
endwhile
6: print(sum)

CD(5)=3



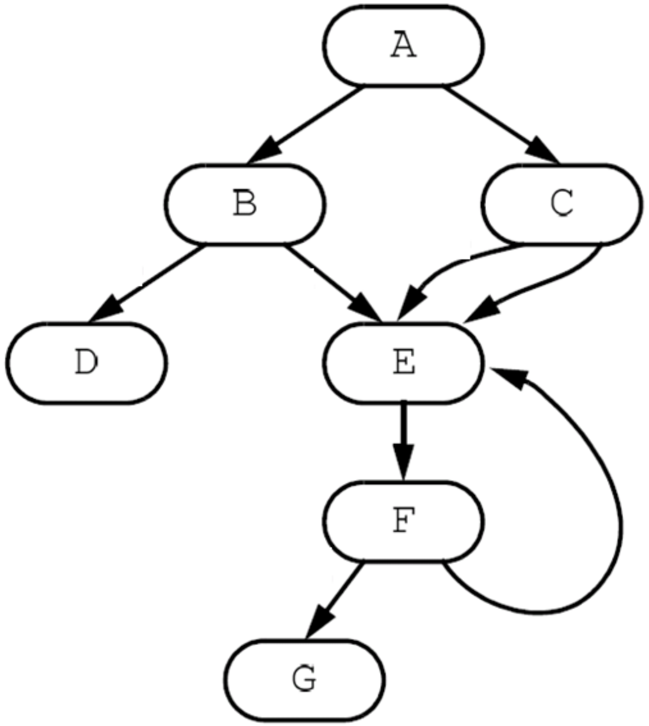
Outline

- Overview
- Static Program Representations
  - Abstract Syntax Tree
  - Control Flow Graph
  - Program Dependence Graph
  - Call Graph
- Static Program Analysis Tools
  - AspectJ
  - TraceMatches
  - Soot
- Summary

Call Graph

- Call graph
  - Nodes are procedures
  - Edges are calls

- Hard cases for building call graph
  - Calls through function pointers (or Java reflection)



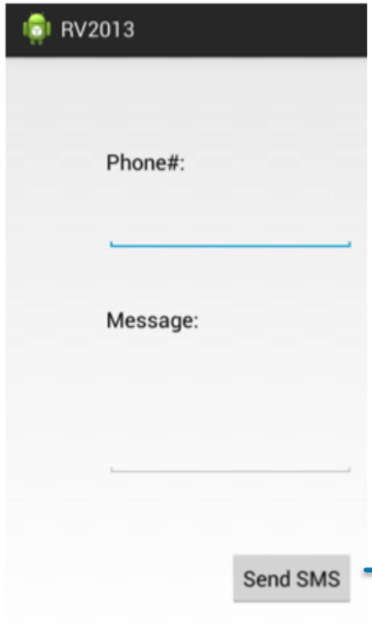
Outline

- Overview
- Static Program Representations
  - Abstract Syntax Tree
  - Control Flow Graph
  - Program Dependence Graph
  - Call Graph
- Static Program Analysis Tools
  - AspectJ
  - TraceMatches
  - Soot
- Summary

Outline

- Overview
- Static Program Representations
  - Abstract Syntax Tree
  - Control Flow Graph
  - Program Dependence Graph
  - Call Graph
- Static Program Analysis Tools
  - AspectJ
  - TraceMatches
  - Soot
- Summary

Running Example: SMS Messenger




```java
public class RV2013 extends Activity {

  private EditText phoneNr, message;
  private SmsManager smsManager = SmsManager.getDefault();

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_rv2013);
    Log.i("INFO", "in onCreate");
   }

  public void sendSms(View v){
    Log.i("INFO", "in sendSms");

    phoneNr = (EditText)findViewById(R.id.phoneNr);
    message = (EditText)findViewById(R.id.message);

    smsManager.sendTextMessage(phoneNr.getText().toString(), null,
       message.getText().toString(), null, null);

    }
}
```

Policy 1: No Premium SMS Messages

- Policy 1: Do not send messages to 0900 numbers

- Idea:
  - Intercept all calls to SmsManager.sendTextMessage()
  - If phone number starts with 0900, raise an alert
  - Otherwise, proceed as normal

- Can be done using all the tools
  - Most straightforward pick: AspectJ

Policy 2: A Closer Look

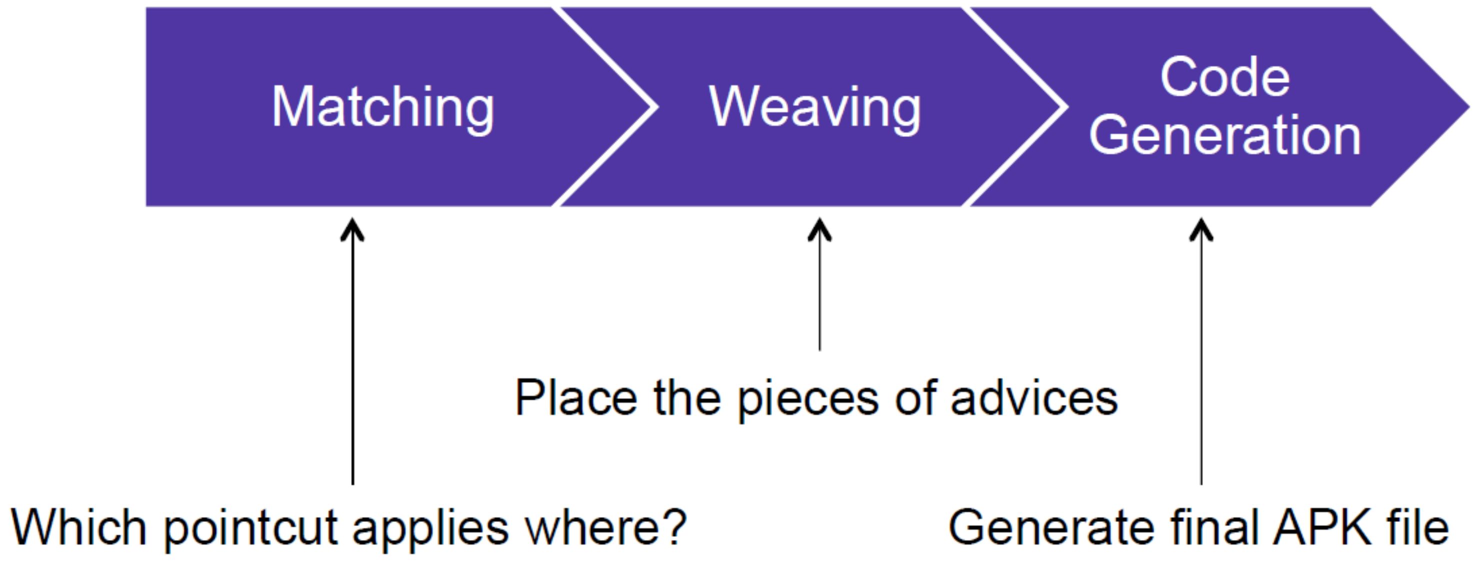
- Policy 2: Do not send more than three messages to same number
- Idea:
  - Intercept all calls to SmsManager.sendTextMessage()
  - On every call, increment a counter by 1

  - If the counter below or equal to 3, proceed normally
  - If the counter exceeds 3, raise an alert and block

- Can be done using all the tools
  - Most straightforward pick: Tracematches

AspectJ

Three phases for generating the instrumented application:



AspectJ - Motivation

```
public void withdraw(long accId, int amount) {
  if (hasPermission(accId)) {
    bankLogger.info(accId + " withdrow amount " + amount);

    Transaction tx = null;
    try {
      Transaction tx = session.beginTransaction();

      Account account = (Account)session.get(Account.class, accId);
      double balance = account.getBalance();
      balance = balance - amount;
      account.setBalance(balance);
      session.save(balance);

      tx.commit();
    } catch(RuntimeException ex) {
      if (tx!=null) tx.rollback;
    }

  } else {
    throw new SecurityException("Access Denied");
  }
}
```

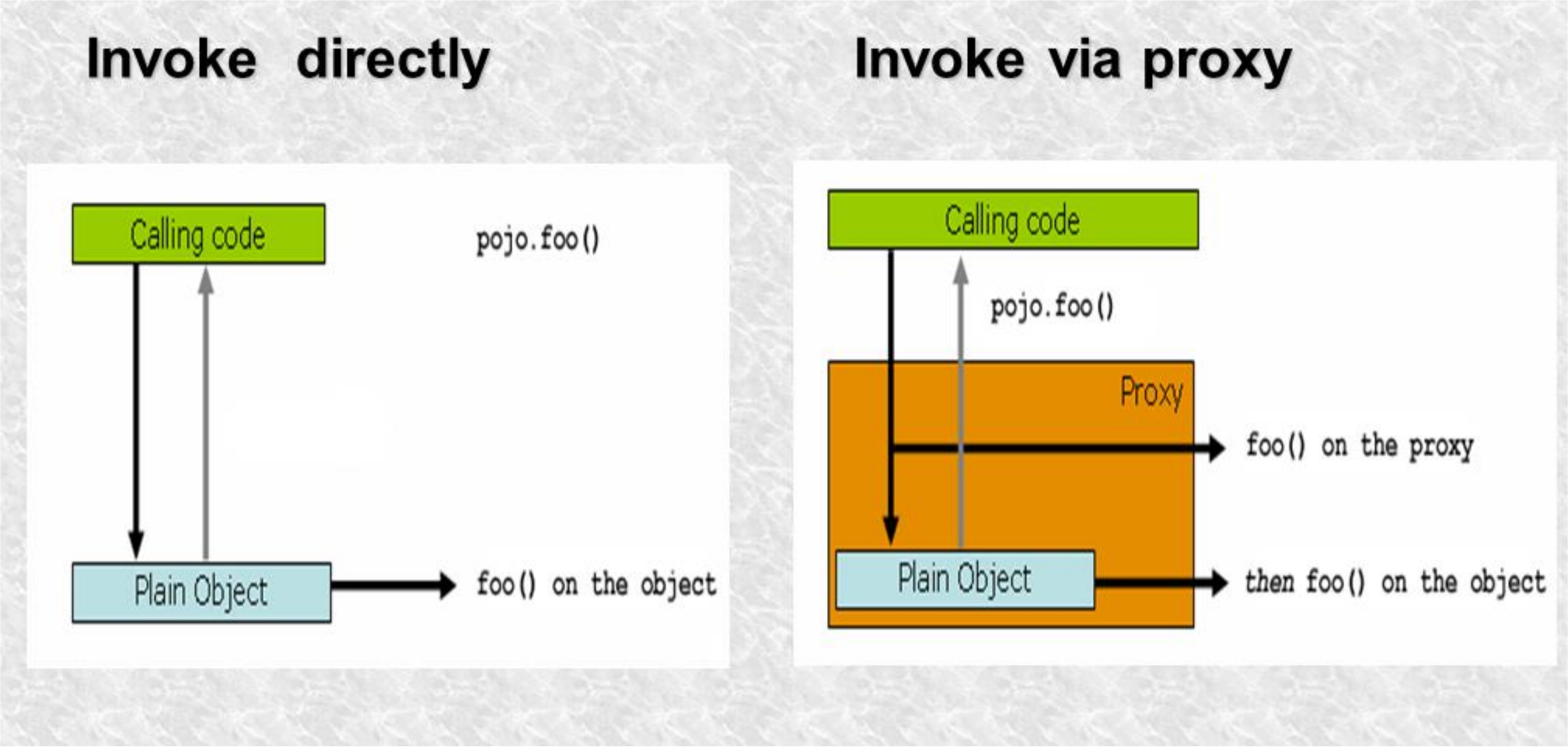
Business

Permission

Logger

Transaction

AspectJ - Methodology



Instrumentation with AspectJ

```
public void sendSms(View v) {
     phoneNr = (EditText)findViewById(R.id.phoneNr);
     message = (EditText)findViewById(R.id.message);

     smsManager.sendTextMessage(phoneNr.getText().toString(), null, message.getText().toStri
            null, null);

   }
}
```

"before" advice

"around" advice

"after" advice

Pointcut

AspectJ: A Simple Example (1)

```
import android.telephony.SmsManager;
import android.app.PendingIntent;
import android.util.Log;

public aspect SendSMS_3sms {
    pointcut sendSms() : call (void SmsManager.sendTextMessage
            (String, String, String, PendingIntent, PendingIntent));




}
```

Pointcut

AspectJ: A Simple Example (1)

```
import android.telephony.SmsManager;
import android.app.PendingIntent;
import android.util.Log;

public aspect SendSMS_3sms {
    pointcut sendSms() : call (* SmsManager.sendTextMessage(..));

    after(): sendSms() {
        Log.i("Aspect", "SMS message sent.");
    }
}
```

Pointcut

“after” advice

AspectJ: A Simple Example (2)

```
public void sendSms(View v) {
     phoneNr = (EditText)findViewById(R.id.phoneNr);
     message = (EditText)findViewById(R.id.message);



     smsManager.sendTextMessage(phoneNr.getText().toString(), null, message.getText().toString(),
            null, null);

     Log.i("Aspect", "SMS message sent.");


   }
}
```

AspectJ: Parameterized Pointcuts

```
import android.telephony.SmsManager;
import android.app.PendingIntent;
import android.util.Log;

public aspect SendSMS_3sms {
    pointcut sendSms(String no) : call (* SmsManager.sendTextMessage(..))
            && args(no, ..);

    after(String no): sendSms(no) {
            Log.i("Aspect", "SMS message sent to no. " + no);
    }
}
```

Pointcut

“after” advice

Recap on Policy 1: No Premium SMS msgs.

- Policy 1: Do not send messages to 0900 numbers

- Idea:
  - Intercept all calls to SmsManager.sendTextMessage()
  - If phone number starts with 0900, raise an alert
  - Otherwise, proceed as normal

- We need to replace the original code
  - “around” advice: instead-of, with the ability to “proceed” to original code

Policy 1: No Premium SMS Messages

```
import android.telephony.SmsManager;
import android.app.PendingIntent;
import android.util.Log;

public aspect SendSMS_PremiumAspect {
    pointcut sendSms(String no) : call (void SmsManager.sendTextMessage(..)) && args(no, ..);

    void around(String no): sendSms(no) {
        if (no.startsWith("0900"))
            Log.e("Aspect", "Premium SMS message blocked.");
        else
            proceed(no);
    }
}
```

Recap on Policy 2: Prevent SMS Spam

- Policy 2: Do not send more than three messages to the same number

- Idea:
  - Intercept all calls to SmsManager.sendTextMessage()
  - On every call, increment a counter by 1

  - If the counter below or equal to 3, proceed normally
  - If the counter exceeds 3, raise an alert and block

Policy 2: No SMS Spam

```
import …

public aspect SendSMS_PremiumAspect {
    Map<String, Integer> counter = new HashMap<String, Integer>();

    pointcut sendSms(String no) : call (void SmsManager.sendTextMessage(..)) && args(no, ..);

    void around(String no): sendSms(no) {
        if (counter.containsKey(no)) counter.put(no, counter.get(no) + 1); else counter.put(no, 1);
        if (counter.get(no) > 3)
            Log.e("Aspect", “SMS spam message blocked.");
        else
            proceed(no);
    }
}
```

Limitations of AspectJ

- Use around advice to block policy violations
  - Does not remove dependent code / “backwards slice”
  - Example: Remove all debug outputs, computation of debug values remains
- No global reasoning about the program
  - Premium SMS messages may only be sent to numbers entered by the user
- Monitors for sequences cumbersome to implement
  - Remember the map for the counts per phone number
  - Can we do better?

Outline

- Overview
- Static Program Representations
  - Abstract Syntax Tree
  - Control Flow Graph
  - Program Dependence Graph
  - Call Graph
- Static Program Analysis Tools
  - AspectJ
  - TraceMatches
  - Soot
- Summary

TRACEMATCHES

Adding trace matching with free variables to AspectJ

Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam and Julian Tibble

OOPSLA 2005

http://dl.acm.org/citation.cfm?id=1094839

Recap on Policy 2: Prevent SMS Spam

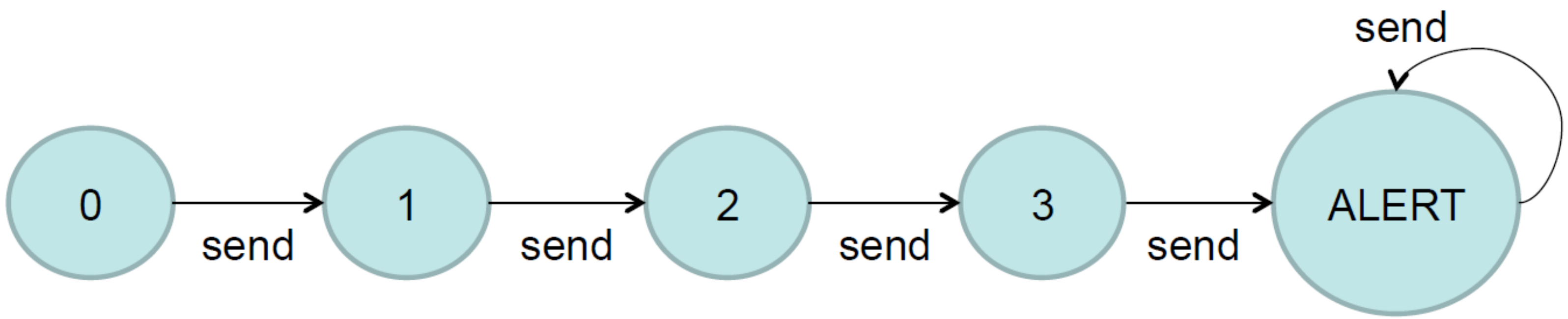
- Policy 2: Do not send more than three messages to same number

- Looks like an automaton
  - “SMS message sent” is an event
  - Use states for counting
  - Normal states (s0, .. s3), alert state s4

- Use one automaton per phone number
  - Always the same structure, we just need a single blueprint

Policy 2: The Automaton

Policy 1: Do not send more than three messages to same number



Finite-state automata can be expressed as regular expressions!

send, send, send, send+

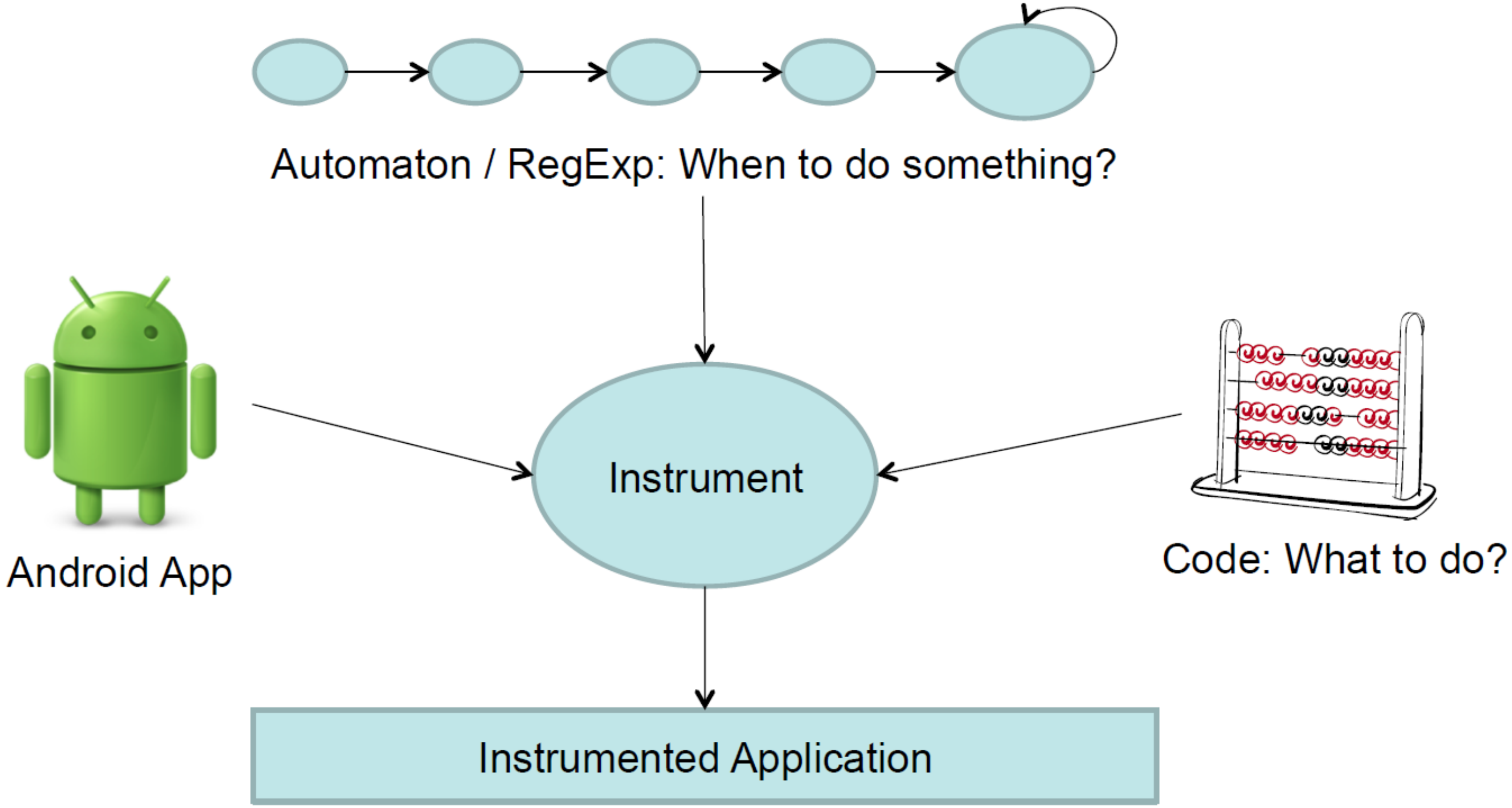
send[3] send+

Policy 2: Declarative State Machine Defs.

- Tracematches handles the automaton for us!
  - Declaratively instrument apps with automaton-based monitors
  - Regular expression defines the monitor
  - If the monitor automaton accepts, user-defined code is run
  - No custom bookkeeping for automaton required!

- Allows for much more concise definition of policy 2

Policy 2: The Big Picture



Tracematches – SMS Spam

```
import android.telephony.SmsManager;
import android.app.PendingIntent;
import android.util.Log;

public aspect SMSSpam {
    tracematch (String no) {
        sym sendSms after:
            call (void SmsManager.sendTextMessage(..)) && args (no,..);
        sendSms[3] sendSms+ {
            Log.e("SPAM", "SMS spam detected to no: " + no);
        }
    }
}
```

No manual bookkeeping required

Tracematches – Limitations

- Tracematches only support finite state machines / regular expressions
- Tracematches cannot share symbol definitions
- No possibility of custom bookkeeping inside the automaton
  - Not possible to enforce more complex privacy policies

Outline

- Overview
- Static Program Representations
  - Abstract Syntax Tree
  - Control Flow Graph
  - Program Dependence Graph
  - Call Graph
- Static Program Analysis Tools
  - AspectJ
  - TraceMatches
  - Soot
- Summary

Soot overview

- Soot: a Java compiler testbed, static analysis and transformation tool
- Tools based on Soot:
  - translation of Java to C
  - instrumentation of Java programs
  - obfuscator for Java
  - software watermarking
- Jimple: Java sIMPLE, a stack-less, three address representation, only 15 instructions

Soot

- Important Resources
  - Options
  - Java Doc
  - http://www.brics.dk/SootGuide/

Intermediate Representations

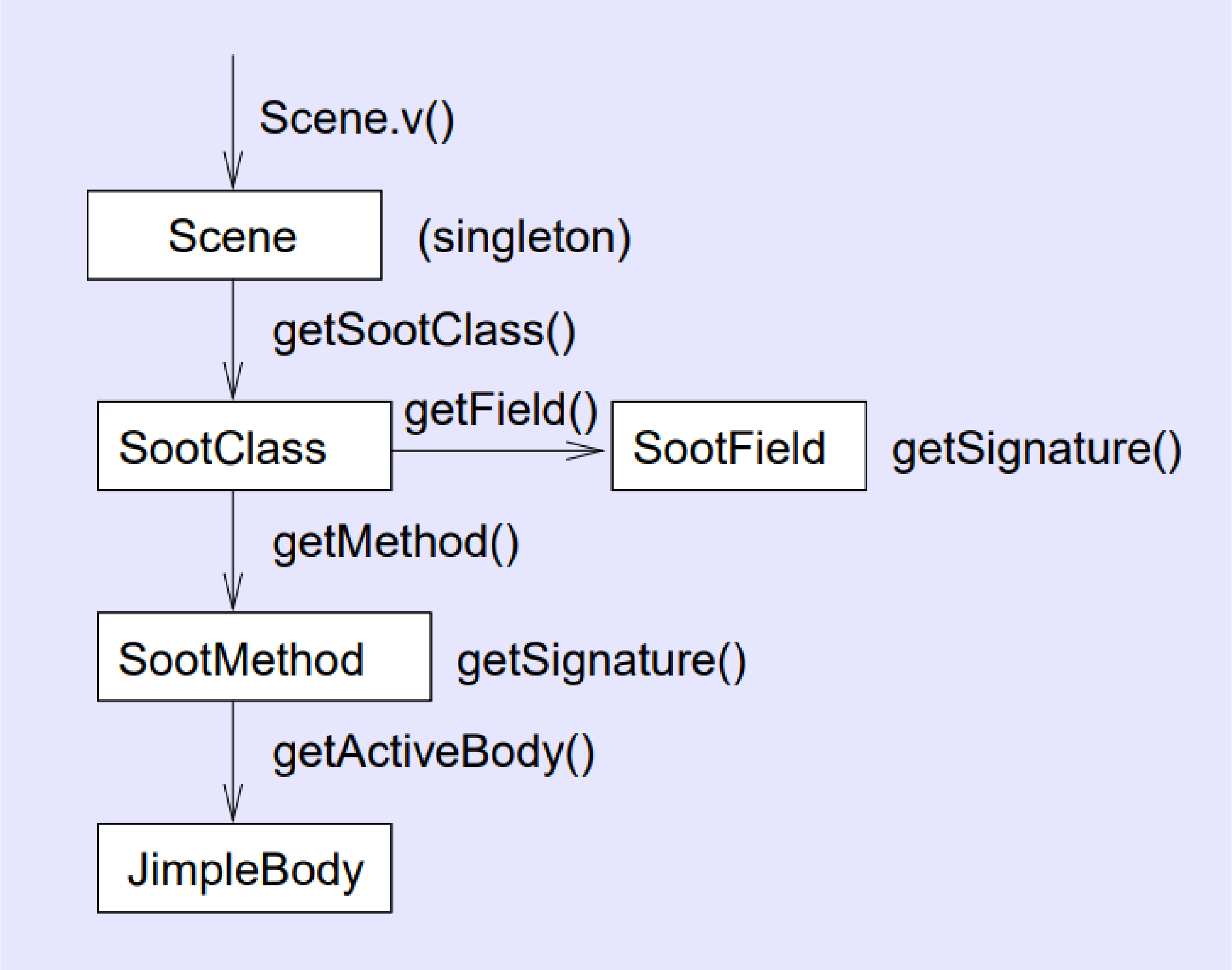
- Baf
  - a compact rep. of Bytecode (stack-based)
- Jimple
  - Java’s simple, typed, 3-addr (stackless) representation
- Shimple
  - Static Single Assignment-form version of the Jimple representation.
    - SSA-form guarantees that each local variable has a single static point of definition which significantly simplifies a number of analyses.
- Grimp
  - similar to Jimple, but with expressions aggregated
    - allows trees of expressions together with a representation of the new operator
    - in this respect Grimp is closer to resembling Java source code than Jimple is and so is easier to read and hence the best intermediate representation for inspecting disassembled code by a human reader.

Soot Data Structure Basics

- Soot builds data structures to represent:
  - a complete environment (Scene)
  - classes (SootClass)
  - Fields and Methods (SootMethod, SootField)
  - bodies of Methods (come in different flavors, corresponding to different IR levels, ie. JimpleBody)

- These data structures are implemented using OO techniques, and designed to be easy to use and generic where possible.

Soot Classes



Soot Classes – Example on Callgraph

```
@Override
protected void internalTransform(String phaseName, Map options) {
        CHATransformer.v().transform();
                SootClass a = Scene.v().getSootClass("testers.A");

        SootMethod src = Scene.v().getMainClass().getMethodByName("doStuff");
        CallGraph cg = Scene.v().getCallGraph();

        Iterator<MethodOrMethodContext> targets = new Targets(cg.edgesOutOf(src));
        while (targets.hasNext()) {
            SootMethod tgt = (SootMethod)targets.next();
            System.out.println(src + " may call " + tgt);
        }
}
```

Step 1: New Body Transformer

```
PackManager.v().getPack("jtp").add(
    new Transform("jtp.myAnalysis", new MyBodyTransformer()));
```

Add own BodyTransformer

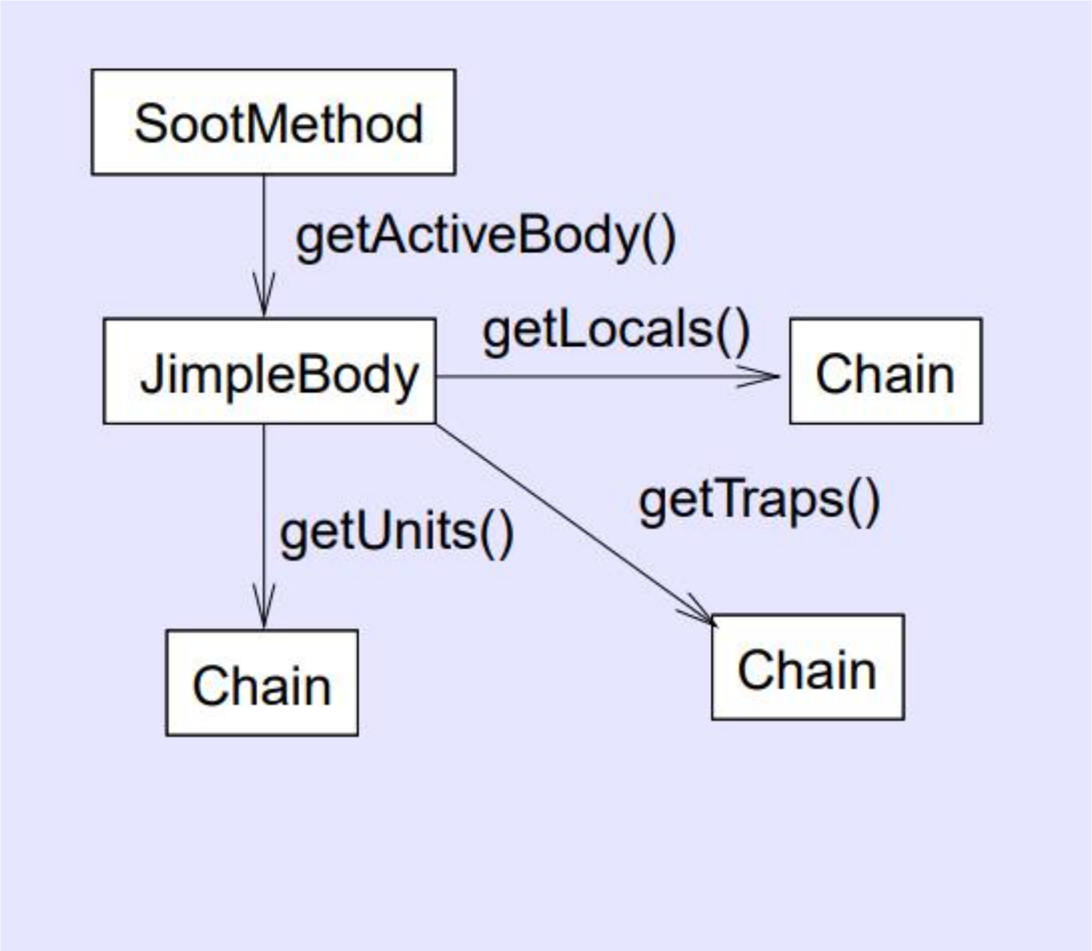
```
soot.Main.main(new String[] { ... })
```

Start Soot

Step 2: Iterating over classes and methods

```
@Override
protected void internalTransform(Body body, String arg0, Map arg1) {

        Iterator<Unit> i = body.getUnits().snapshotIterator();

            while (i.hasNext()) {
                    Unit u = i.next();
                     //do something
            }
        }
      }
  }
}
```

Body-centric View



Adding/Removing Statements

Jimple Statement 1

`insertBefore(`*`newStmt`*`, `*`stmt`*`)`

Jimple Statement 2

`insertAfter(newStmt, stmt)`

~~Jimple Statement 3~~

~~Jimple Statement 4~~

`remove(stmt)`

...

Removing Statements

```
....
while (i.hasNext()) {
  Stmt s = (Stmt)i.next();
  if (s.containsInvokeExpr()) {
    String declaringClass =
            s.getInvokeExpr().getMethod().getDeclaringClass().getName();
    if (declaringClass.equals("android.util.Log"))
       body.getUnits().remove(s);
   }
}
...
```

check for invoke expressions

get the class name

check for a specific class

Full Callgraph Example

```java
public class CallGraphExample
{
    public static void main(String[] args) {
       List<String> argsList = new ArrayList<String>(Arrays.asList(args));
       argsList.addAll(Arrays.asList(new String[]{
               "-w",
               "-main-class",
               "testers.CallGraphs",//main-class
               "testers.CallGraphs",//argument classes
               "testers.A"          //
       }));


       PackManager.v().getPack("wjtp").add(new Transform("wjtp.myTrans", new SceneTransformer() {

        @Override
        protected void internalTransform(String phaseName, Map options) {
               CHATransformer.v().transform();
                       SootClass a = Scene.v().getSootClass("testers.A");

               SootMethod src = Scene.v().getMainClass().getMethodByName("doStuff");
               CallGraph cg = Scene.v().getCallGraph();

               Iterator<MethodOrMethodContext> targets = new Targets(cg.edgesOutOf(src));
               while (targets.hasNext()) {
                   SootMethod tgt = (SootMethod)targets.next();
                   System.out.println(src + " may call " + tgt);
               }
        }

       }));

           args = argsList.toArray(new String[0]);

           soot.Main.main(args);
    }
}
```

```java
public class CallGraphExample
{
    public static void main(String[] args) {
        List<String> argsList = new ArrayList<String>(Arrays.asList(args));
        argsList.addAll(Arrays.asList(new String[]{
                "-w",
                "-main-class",
                "testers.CallGraphs",//main-class
                "testers.CallGraphs",//argument classes
                "testers.A"          //
        }));

        PackManager.v().getPack("wjtp").add(new Transform("wjtp.myTrans", new SceneTransformer() {

         @Override
         protected void internalTransform(String phaseName, Map options) {
                CHATransformer.v().transform();
                        SootClass a = Scene.v().getSootClass("testers.A");

                SootMethod src = Scene.v().getMainClass().getMethodByName("doStuff");
                CallGraph cg = Scene.v().getCallGraph();

                Iterator<MethodOrMethodContext> targets = new Targets(cg.edgesOutOf(src));
                while (targets.hasNext()) {
                    SootMethod tgt = (SootMethod)targets.next();
                    System.out.println(src + " may call " + tgt);
                }
         }

        }));

            args = argsList.toArray(new String[0]);

            soot.Main.main(args);
    }
}
```