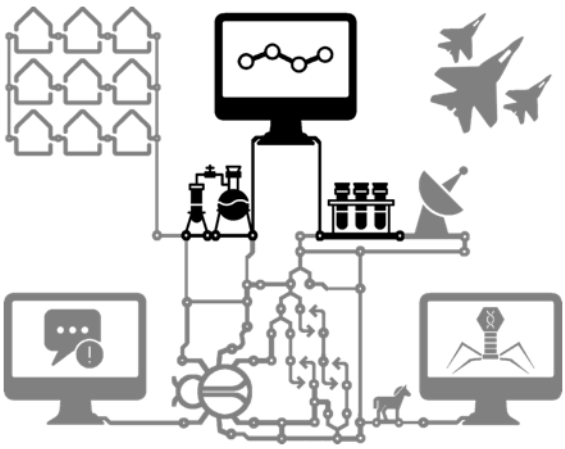
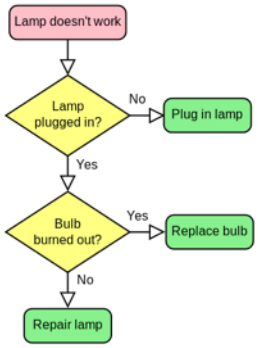
Fundamentals of Program Analysis



Ice Breaker Exercise: EIL5 “Programming”

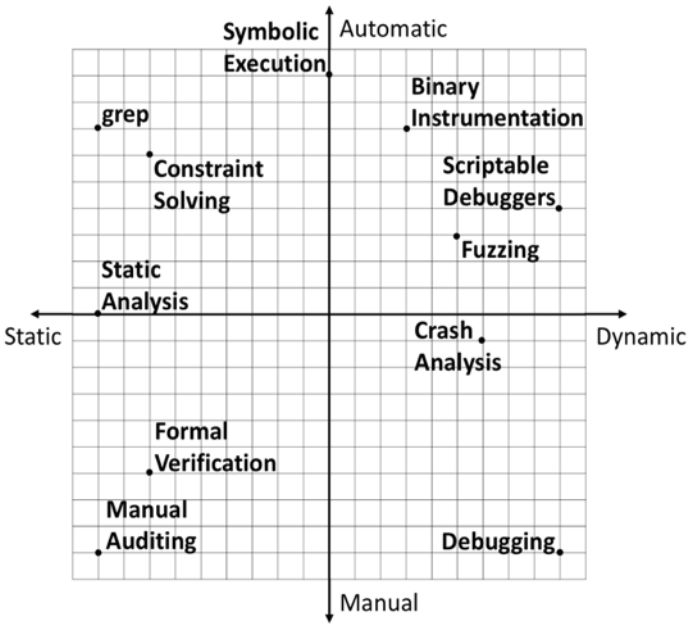
- Explain It Like I’m Five (EIL5): How do computer programs work?
- Can your explanation intuitively address:
  - Complexity of software
  - Programming bugs
  - Security issues



Computers understand and follow very simple instructions. They do not know right from wrong, they only follow instructions exactly as they see them. Programs are made of these simple instructions and cab be thought of like flowcharts. Flowcharts take some *data* (YES/NO) to make decisions. If/Then relationships (Did you eat breakfast today? -> YES/NO) let us *control* decisions based on the answers. We can even loop (Did you eat breakfast today -> No? -> Go back to the start.). We can make lots of flowcharts and combine them to make really complicated programs. Even though the idea of flowcharts is very simple, a big flow chart can be very confusing to understand right? What if you make a mistake in the flowchart? How do you find the mistake? Could someone think of bad answers that cause your flowchart to give a wrong answer? What if I gave some inputs that cause you to go in a loop forever in your flowchart and never give an answer (example: I say I never eat breakfast)?

How do we analyze programs?

- Dynamic Analysis
  - Run the program and see what happens
  - How do we execute all interesting program paths?
  - What do we look for?
- Static Analysis
  - Look at what's inside the program
  - How do we know which program paths are possible?
  - What do we look for?



Source: Contemporary Automatic Program Analysis,
Julian Cohen, Blackhat 2014

Static Program Analysis with Atlas

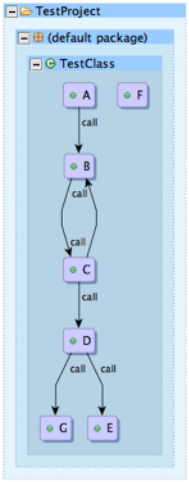
```
1  public class TestClass {
2
3      public void A() {
4          B();
5      }
6
7      public void B() {
8          C();
9      }
10
11     public void C() {
12         B();
13         D();
14     }
15
16     public void D() {
17         G();
18         E();
19     }
20
21     public void E() {
22
23     }
24
25     public void F() {
26
27     }
28
29     public void G(){
30
31     }
32
33 }
```

Program Declarations, Control Flow, and Data Flow

atlas professional

Queryable Graph Database

2-way Source Correspondence



Atlas Query Language

- eXtensible Common Software Graph (XCSG) schema
  - Heterogeneous, attributed, directed graph data structure as an abstraction to represent the essential aspects of the program's syntax and semantics (structure, control flow, and data flow), which are required to reason about software.
  - Expressive query language for users to write composable analyzers
  - Results computed in the form of subgraphs defined by the query, which can be visualized or used as input in to other queries

A Thought Experiment - Given the following program what graph(s) could we produce?

```
public class MyClass {
   public static void A() {
      B();
   }
   public static void B() {
      C();
   }
   public static void C() {
      B();
      D();
   }
   public static void D() {
      G();
      E();
   }
   public static void E() {}
   public static void F() {}
   public static void G() {}
}
```

Control Flow (summary)

A *calls* B
B *calls* C
C *calls* B
C *calls* D
D *calls* G
D *calls* E

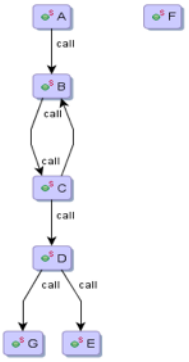
Structure

MyProject *contains* mypackage
mypackage *contains* MyClass
MyClass *contains* methods: A, B, C, D, E, F, G

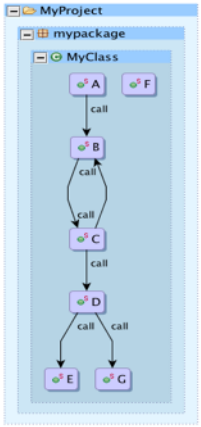
Data Flow?

No data in this program.

Call



Call and Structure

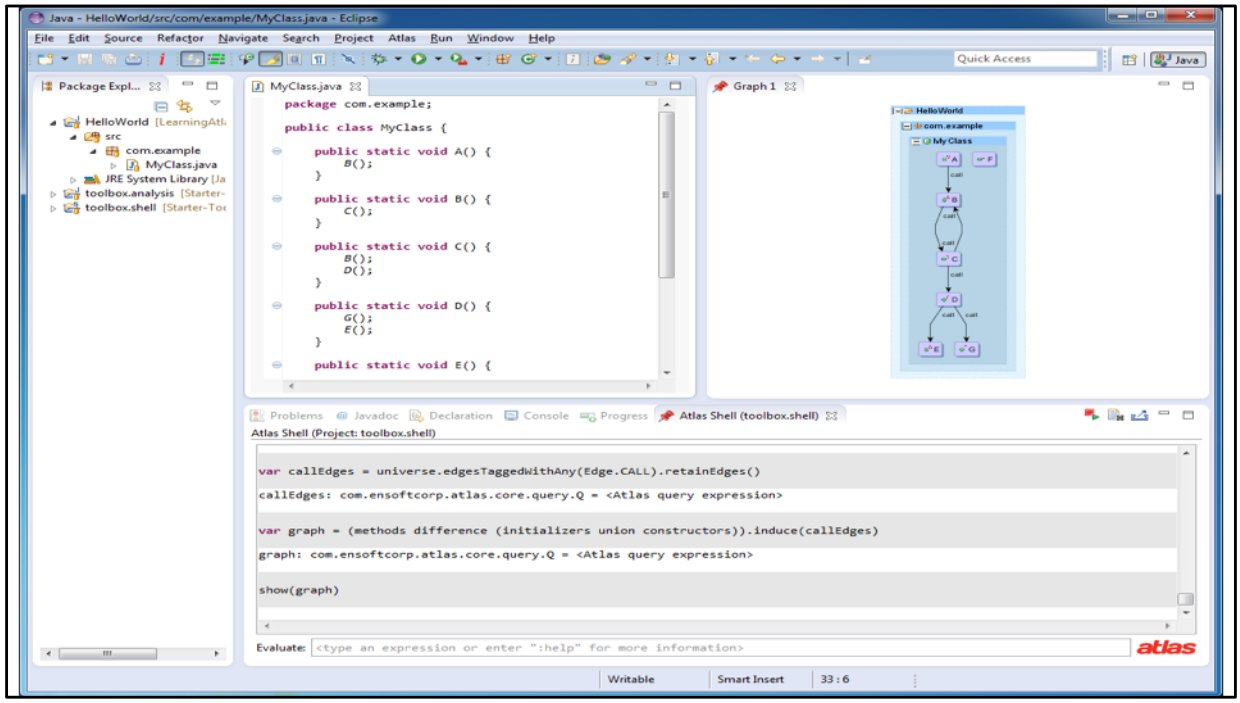


Basic Queries

- Map the Workspace project "*MyProject*"
- Execute the following queries on the Atlas Shell

  (We will discuss what they mean later)

```
var containsEdges = universe.edgesTaggedWithAny(XCSG.Contains)
var app = containsEdges.forward(universe.project("MyProject"))
var appMethods = app.nodesTaggedWithAny(XCSG.Method)
var initializers = app.methods("<init>").union(app.methods("<clinit>"))
var constructors = app.nodesTaggedWithAny(XCSG.Constructor)
var callEdges = universe.edgesTaggedWithAny(XCSG.Call)
var q = (appMethods.difference(initializers.union(constructors))).induce(callEdges)
show(q)
```


Java - HelloWorld/src/com/example/MyClass.java - Eclipse
File Edit Source Refactor Navigate Search Project Atlas Run Window Help
Quick Access
Java
Package Expl...
HelloWorld [LearningAtl
src
com.example
MyClass.java
JRE System Library [Ja
toolbox.analysis [Starter-
toolbox.shell [Starter-To
MyClass.java
package com.example;
public class MyClass {
public static void A() {
B();
}
public static void B() {
C();
}
public static void C() {
B();
D();
}
public static void D() {
G();
E();
}
public static void E() {
Graph 1
HelloWorld
com.example
MyClass
A
F
B
C
D
E
G
call
Problems
Javadoc
Declaration
Console
Progress
Atlas Shell (toolbox.shell)
Atlas Shell (Project: toolbox.shell)
var callEdges = universe.edgesTaggedWithAny(Edge.CALL).retainEdges()
callEdges: com.ensoftcorp.atlas.core.query.Q = <Atlas query expression>
var graph = (methods difference (initializers union constructors)).induce(callEdges)
graph: com.ensoftcorp.atlas.core.query.Q = <Atlas query expression>
show(graph)
Evaluate:
<type an expression or enter ":help" for more information>
atlas
Writable
Smart Insert
33 : 6

Basic Queries

var A = app.methods("A")
var B = app.methods("B")
var C = app.methods("C")
var D = app.methods("D")
var E = app.methods("E")
var F = app.methods("F")
var G = app.methods("G")

…alternatively…

var A = selected
var B = selected

Note: In the following examples you will need to pass the result to the "show" method on the Atlas Shell to view the results.

Example: show(q.forward(D))

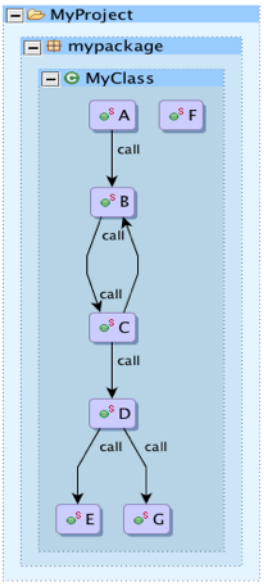
Declare a few variables to represent different methods in our example graph. Note that you can also use the "selected" variable after clicking on the Atlas graph or corresponding source file element.

Forward Traversals

q.forward(origin)
Selects the graph reachable from the given nodes using the forward transitive traversal. Includes the origin in the resulting graph query.
q.forward(D) outputs the graph D→E and D→G.
q.forward(C) outputs the graph C→B→C, C→D→E and C→D→G.

q.forwardStep(origin)
Selects the graph reachable from the given nodes along forward paths of length one. Includes the origin in the resulting graph query.
q.forwardStep(D) outputs the graph D→E and D→G.
q.forwardStep(C) outputs the graph C→B and C→D.
q.forwardStep(F) outputs the graph only F.

q.successors(origin)
Selects the immediate successors reachable from the given nodes Does not include the origin unless it succeeds itself. The result does not includes edges.
q.successors(C) outputs: {D, B}
q.successors(F) outputs: Empty graph

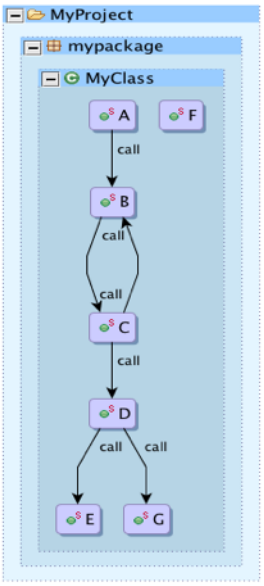


Reverse Traversals

q.reverse(origin)
Selects the graph reachable from the given nodes using the reverse transitive traversal. Includes the origin in the resulting graph query.
q.reverse(D) outputs the graph D←C←B←A and D←C←B←C.
q.reverse(C) outputs the graph C←B←A and C←B←C.

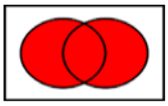
q.reverseStep(origin)
Selects the graph reachable from the given nodes along reverse paths of length one. Includes the origin in the resulting graph query.
q.reverseStep(D) outputs the graph D←C.
q.reverseStep(C) outputs the graph C←B.

q.predecessors(origin)
Selects the immediate predecessors reachable from the given nodes. Does not include the origin unless it precedes itself. The result does not include edge.
q.predecessors(C) outputs: {B}
q.predecessors(F) output: Empty graph.

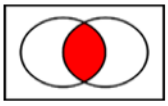


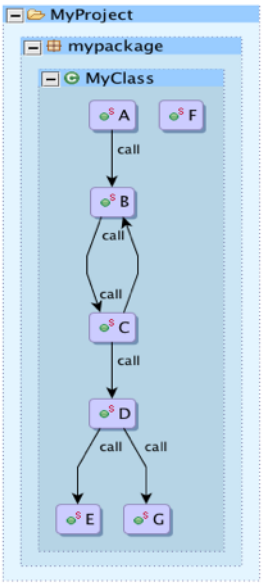
Set Operations (1)

q.union(q2…)
Yields the union of nodes and edges of *this* graph and the *other* graphs.
B.union(C) outputs a graph with nodes B and C.
A.union(B, C) outputs a graph with nodes A, B, and C.
q.union(C) outputs the entire graph.



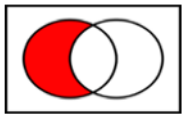
q.intersection(q2…)
Yields the intersection of nodes and edges of *this* graph and the *other* graphs.
A.intersection(B) outputs an empty graph.
q.intersection(C) outputs a graph with only the node C.



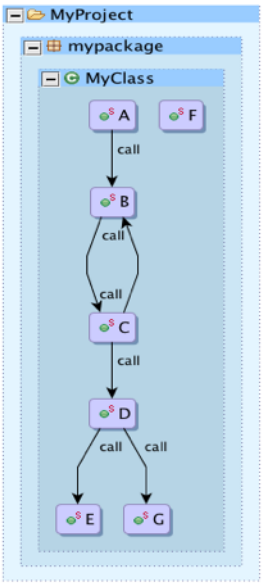


Set Operations (2)

q.difference(q2…)
Selects *q*, excluding nodes and edges in q2. Removing an edge necessarily removes the nodes it connects. Removing a node remove the connecting edge as well.
B.difference(C) outputs a graph with only the node B.
B.difference(A, B) outputs an empty graph.
q.difference(C) outputs the shown graph without the node C and any edges entering or leaving node C.



q.differenceEdges(q2…)
Selects q, excluding the edges from q2.
q.differenceEdges(q) outputs only the nodes A,B,C,D,E,F,G.
q.differenceEdges(q.forwardStep(B)) outputs the graph A→B, C→B, C→D, D→E, D→G, and F (the edge B→C is removed from the original graph).

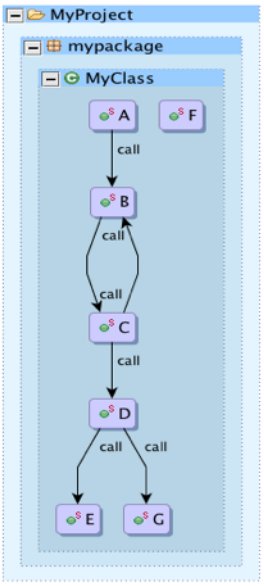


Between Traversals

q.between(fromX, toY)
Selects the subgraph containing all paths starting from a set X to a set Y.
q.between(C, A) outputs Empty graph.
q.between(C, E) outputs the graph C→D→E, C→B→C.

q.betweenStep(fromX, toY)
Selects the subgraph containing all paths of length one starting from a set X to a set Y.
q.betweenStep(C, D) outputs the graph C→D.
q.betweenStep(D, C) outputs Empty graph.
q.betweenStep(C, E) outputs Empty graph.

Note: A possible implementation of betweenStep could be:
q.forwardStep(fromX).intersection(q.reverseStep(toY))



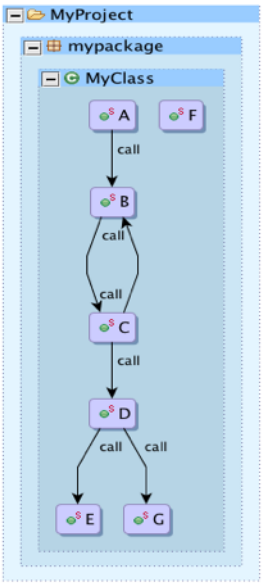
Graph Operations (1)

q.leaves()
Selects the nodes from the given graph with no successors.
q.leaves() outputs {E, F, G}.

q.roots()
Selects the nodes from the given graph with no predecessors.
q.roots() outputs {A, F}.

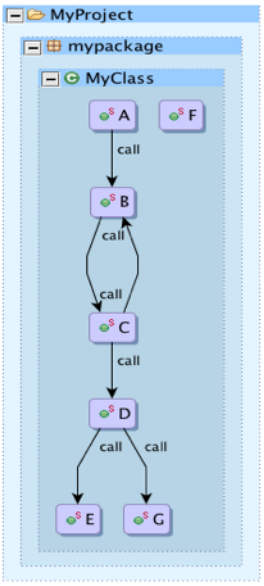
q.retainNodes()
Selects all nodes from the graph, ignoring edges.
q.retainNodes() outputs {A,B,C,D,E,F,G}.

q.retainEdges()
Retain only edges and nodes connected to edges.
q.retainEdges() outputs the shown graph without F



Graph Operations (2)

q2.induce(q)
Adds edges from the given graph query *q2* to *q*.
var q2 = B.union(C)
q2.induce(q) outputs the graph B→C→B.



Graph Elements

- In Atlas a *Q* (query) object can be thought of as a recipe to a *constraint satisfaction problem* (CSP). Building and chaining together *Q*'s costs you almost nothing, but when you ask to see what is in the *Q* (by showing or evaluating the *Q*) Atlas must evaluate the query and execute the graph traversals.
- The evaluated result is a *Graph*. A *Graph* is a set of *GraphElement* objects. In Atlas both a *Node* and an *Edge* are *GraphElement* objects.

```
Graph graph = q.eval();
AtlasSet<Node> graphNodes = graph.nodes();
AtlasSet<Edge> graphEdges = graph.edges();
```

GraphElement Attributes

- A *GraphElement* (Node/Edge) can have attributes
- An attribute is a key that corresponds to a value in the *GraphElement* attribute map.
  - An attribute that is common to almost all nodes and edges is *XCSG.name*.

```
for(Node graphNode : graphNodes){
  String name = (String) graphNode.attr().get(XCSG.name);
}
```

  - Another common attribute is the source correspondence that stores the file and character offset of the source code corresponding to the node or edge. Double clicking on a node or edge takes us to the corresponding source code!

Selecting GraphElements on Attributes

- Attributes can be used to select *GraphElement*s (nodes/edges) out of a graph.
  - For example from the graph we can select all method nodes with the attribute key XCSG.name that have the value "main".

    Q mainMethods = q.selectNode(XCSG.name, "main");

  - We could also select all array's with 3 dimensions.

    Q 3DimArrays = q.selectNode(XCSG.arrayDimension, 3);

Tags: A Special Kind of Attribute

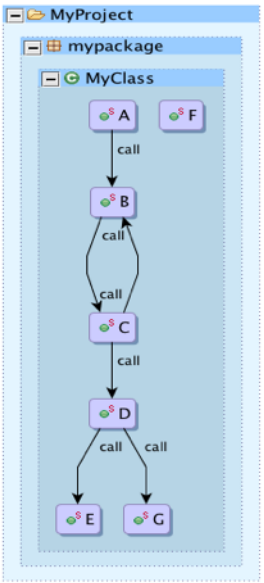
- A Tag is an attribute whose value is TRUE (T)
- The presence of a tag denotes that a *Node* or *Edge* is a member of a set.
  - For example, all method nodes are tagged with *XCSG.Method*.
- Atlas provides several default tags such as *XCSG.Method* that should be used to make code cleaner (and safer from possible schema changes in the future!).

Selecting GraphElements by Tags (1)

q.nodesTaggedWithAny(...)
Selects the nodes tagged with at least one of the given tags.
q.nodesTaggedWithAny(XCSG.Method) outputs: {A,B,C,D,E,F,G}.
q.nodesTaggedWithAny(XCSG.Class) outputs: empty graph.
q.nodesTaggedWithAny(XCSG.Method, XCSG.Class) outputs: {A,B,C,D,E,F,G}.

q.nodesTaggedWithAll(...)
Selects the nodes tagged with all of the given tags.
q.nodesTaggedWithAll(XCSG.Method) outputs: {A,B,C,D,E,F,G}.
q.nodesTaggedWithAll(XCSG.Class) outputs: empty graph.
q.nodesTaggedWithAll(XCSG.Method, XCSG.Class) outputs: empty graph.

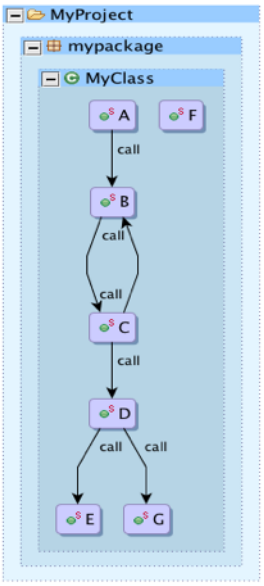
NOTE: The output contains only the nodes.



Selecting GraphElements by Tags (2)

q.edgesTaggedWithAny(...)
Selects edges tagged with at least one of tags. Includes all nodes.
q.edgesTaggedWithAny(XCSG.Call) outputs: the shown graph.

q.edgesTaggedWithAll(...)
Selects edges tagged with all of the given tags. Includes all nodes.
q.edgesTaggedWithAll(XCSG.Call) outputs: the shown q.



Chaining Queries (1)

- We can chain queries to form more complex queries
- Q objects may contain multiple nodes and edges (so an origin can include multiple starting points).
- A second look at the queries we started the example with:

1. First create a subgraph (called containsEdges) of the universe that only contains nodes and edges connected by a *contains* relationship. The Atlas map is heterogeneous, meaning there are many edge and node types. Here we are specifying that we want edges that represent a *contains* relationship from a parent node to a child node.

var containsEdges = universe.edgesTaggedWithAny(XCSG.Contains)

Chaining Queries (2)

2. We then define yet another subgraph (called *app*) which contains nodes and edges declared under the MyProject project.

var app = containsEdges.forward(universe.project(”MyProject"))

3. From the app subgraph we select all nodes that are methods.

var appMethods = app.nodesTaggedWithAny(XCSG.Method)

4. From the subgraph *app* we select all method nodes named "<init>" (instance initializer methods) or "<clinit>" (static initializer methods). We are using a query method called methods(String methodName) that selects methods that have a name that matches the given string. We will explore more query methods later.

var initializers = app.methods("<init>”).union(app.methods("<clinit>"))

Chaining Queries (3)

5. From the app subgraph we select all nodes that are constructors.

var constructors = app.nodesTaggedWithAny(XCSG.Constructor)

6. From the universe create a subgraph (called callEdges) that only contains nodes and edges connected by a call relationship.

var callEdges = universe.edgesTaggedWithAny(XCSG.Call)

7. Define graph to be the methods in the app ignoring initializers and constructors with call edges added in where they exist.

var q = (appMethods difference (initializers.union(constructors))).induce(callEdges)

8. Evaluate and display the graph query.

show(q)

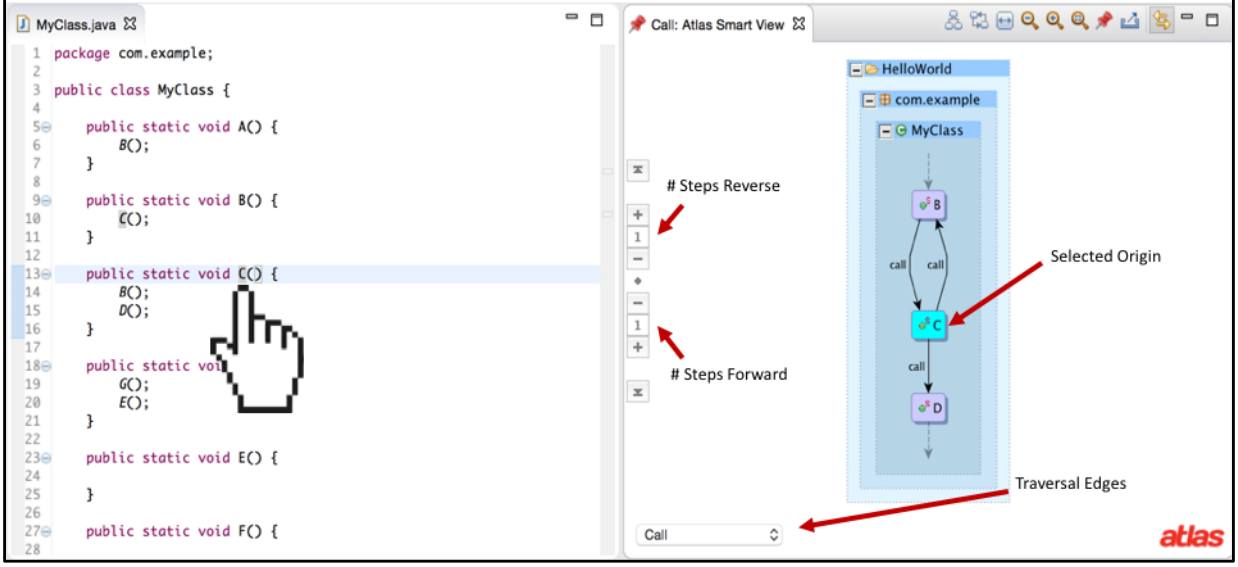
Atlas Schema

- To become proficient in wielding Atlas, you should have:
  - Firm understanding of Extensible Common Software Graph (XCSG) schema
  - Firm understanding of the language you are analyzing (Java source, Jimple, C)
- Examples:
  - How do we detect an inner class with XCSG?
  - No tag for inner class, inner class is defined by a contains relationship.
    - *containsEdges = universe.edgesTaggedWithAny(XCSG.Contains)*
    - *topLevelClasses = containsEdges.successors(universe.nodesTaggedWithAny(XCSG.Package))*
    - *innerClasses = containsEdges.forward(topLevelClasses).difference(topLevelClasses)*
  - What about Java vs. Jimple (Java Bytecode)?
    - No concept of inner classes in bytecode

Atlas Schema Resources

- http://ensoftatlas.com/wiki/Extensible_Common_Software_Graph
- Eclipse → Show Views → Other… → Atlas → Element Detail View
- Atlas Shell (test out queries on the fly!)
- Atlas Smart Views (interactive graphs)

Atlas “Smart Views”



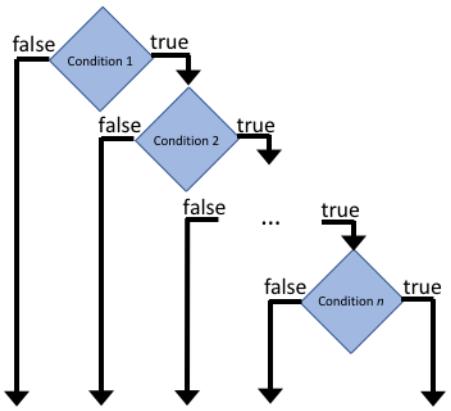
Practice Problems

- Is a java.util.Map a java.util.Collection?
- Let methods B and F be sensitive methods. Return the subset of the sensitive methods that are called by an application.
- Find all calls within the app to custom collection type constructors. Define a custom collection type to be a class that inherits from java.util.Collection and is defined within the app (your project).

Exercise: Counting Program Paths

- How many paths are there for *n* nested branches?

```
if(condition_1){
   if(condition_2){
      if(condition_3){
         ...
         if(condition_n){
            // conditions 1 through n
            // must all be true to reach here
         }
      }
   }
}
```



Each condition controls whether or not the next condition executes. If any *n* condition are false, then execution jumps to the block after condition 1, which gives us *n paths*. There is a single path when all conditions are true that leads to the execution of the code guarded by the n^{th} condition. That gives us n+1 paths for *n* nested branches.

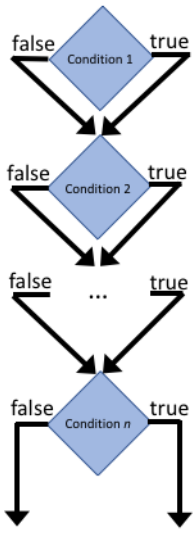
For n=2, there are 3 paths. C1=FALSE/C2=FALSE, C1=TRUE/C2=FALSE, C1=TRUE/C2=TRUE

What if we add a constraint that condition 1 equals condition 2? Then some of the paths are infeasible. Either condition 1 and condition 2 are true in which case the two false paths are not followed or the conditions are false in which case the true path is not followed. The number of feasible paths is less than or equal to the total number of paths in the program. In the worst case we have to consider that all paths are feasible.

Exercise: Counting Program Paths

- How many paths are there for *n* non-nested branches?

```
if(condition_1){
   // code block 1
}
if(condition_2){
   // code block 2
}
if(condition_3){
   // code block 3
}
...
if(condition_n){
   // code block n
}
```



For non-nested branches, each branch is independent of the other. Condition 1 does not influence whether or not condition 2 is executed.

For n=2, there are 4 paths. C1=FALSE/C2=FALSE, C1=TRUE/C2=FALSE, C1=FALSE/C2=TRUE, C2=FALSE/C2=FALSE

Each branch offers two possibilities. For n=3 there are 2*2*2 paths. For *n* non-nested branches there are 2^n paths.

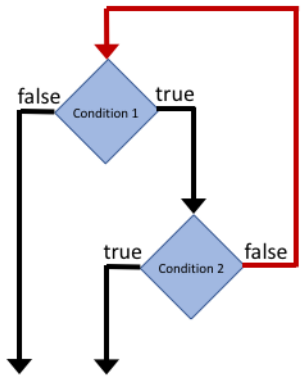
In the worst case, the number of paths in software is exponential!

This is sometimes called the path explosion problem. If we were to count all paths in the Linux kernel there are more paths than there are stars in the galaxy. With the constant growth of software, the computational demands to analyze programs continues to grow.

Exercise: Considering Loops

- Programs may have loops
  - How many paths does this program have?
  - Can we say if this program halts?

```
while(condition_1){
   if(condition_2){
      break;
   }
}
```

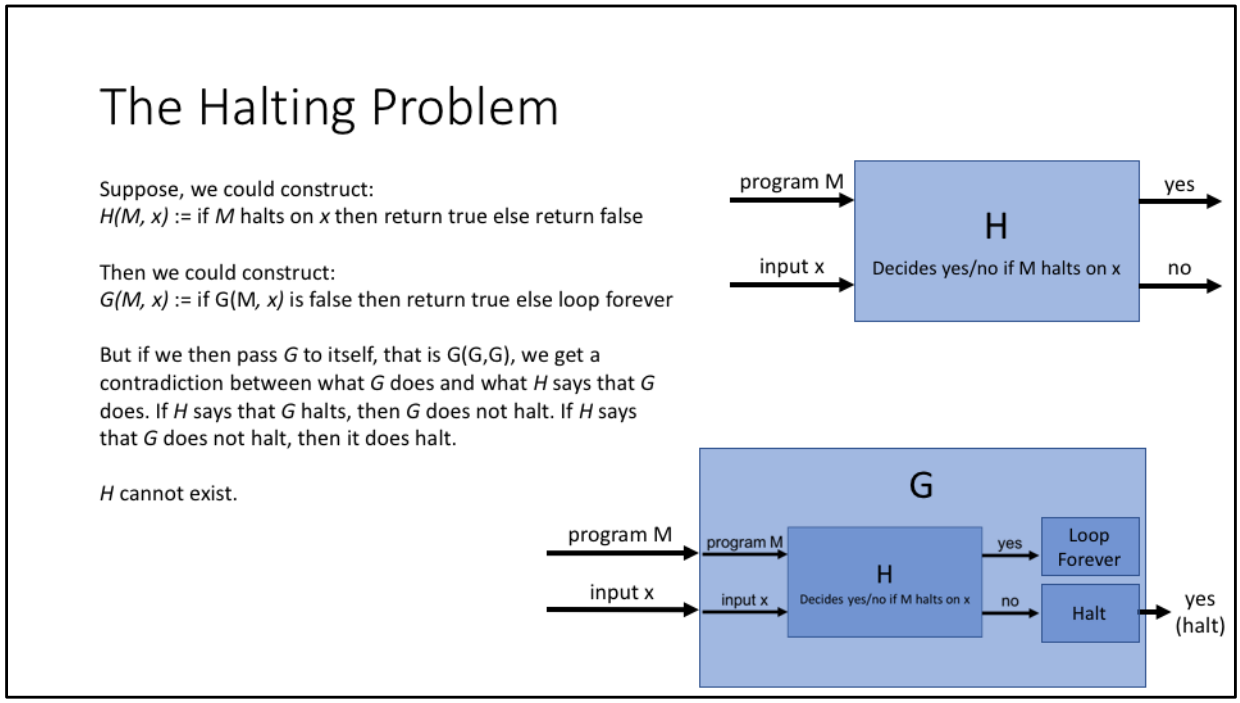


The presence of a back edge indicates there is a loop in the program. In this code condition 1 is a loop header.

If we are going to count paths here we have to consider whether or not we want to treat the path C1 → C2 → C1 as being different than the path C1 → C2 → C1 → C2 → C1. We could count this as one path or an infinite number of paths (looping forever).

Without loops our programs would be very limited. Imagine a program with *n* instructions, where *n* is some finite number. By the pigeon hole principle, a program with *n* instructions must complete in *n* or less steps (running *n*+1 steps means we must have revisited some instruction). Since CPUs run incredibly fast in modern processors, this would imply that most programs would terminate very quickly unless the size of the program was enormous. Loops help to reduce the size of programs by repeating common tasks. In fact sometimes we want to do something *forever* or until the program is terminated by the user such as listen for web connections on a webserver until the webserver is shutdown, which we cannot do without loops. It is common knowledge among experienced developers that the majority of CPU time spent inside a program is spent inside a program's loops.

For this program, if condition 1 is true and condition 2 is false this program loops forever. We can say that this program halts (does not loop forever) if condition 1 is false or if condition 2 is true, but can we answer this question for any arbitrary program? That is, could we write a program that answers yes/no whether or not another program will halt on some input?



Could we write a program that answers yes/no whether or not another program will halt on some input? Surprisingly, no we cannot. While we can say that some programs terminate and some do not, we cannot answer this question for all programs. This is computer science's first and most fundamental theorem. There cannot exist an algorithm that decides whether any given program ever terminates. The halting theorem was proven in both Alonzo Church's and Alan Turing's papers in 1936.

The proof goes like this. Suppose we could construct a program *H* that takes another program *M* and some input *x* that *M* will run on and decides true or false whether or not *M* halts on *x.* If *H* existed, then we could could simply construct a program *G* that runs *H* with *M* and *x* and loops forever if *H* returns yes and halts otherwise. If we feed G to itself, then there is a contradiction between what *G* does and what *H* says that *G* does. If *H* says that *G* halts, then *G* does not halt. If *H* says that *G* does not halt, then it does halt. So *H* cannot exist.

It turns out that the halting problem is undecidable. In fact many questions about programs are undecidable. For example a *points-to* analysis, an analysis that maps variables to the memory the variable is pointing to, has been shown to reduce to the

halting problem. Even the slightly easier, *alias* analysis (answers whether aliases reference the same location in memory) has been reduced to the halting problem. In fact Rice's theorem states that all non-trivial, semantic properties of programs are undecidable. As a result there are fundamental limits on what a program analysis is capable of answering.

Def-Use (DU) Chain

The *backward dataflow* is constructed by applying DU chains.

```
1. x = 2;
2. y = 3;
3. z = 7;
4. a = x + y;
5. b = x + z;
6. a = 2 * x;
7. c = y + x + z;
8. t = a + b;
9. print(t);
```

Statement 8 *defines* **t** and *uses* **a** and **b**

Equivalently, *write-set*(8) = {**t**} and *read-set*(8) = {**a**, **b**}

A *DU chain* consists of a *variable definition*, and *all the uses* of that variable reachable from the definition.

Statement 4 and 6 provide definitions of the variable **a**.

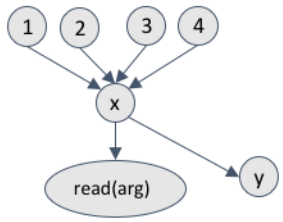
The definition 6 reaches the use of **a** at statement 8

The definition 4 is *killed* by the definition 6, thus it *cannot* reach the use at 8.

How can we have multiple definitions reaching the same use?

Code Transformation (before – flow insensitive): Static Single Assignment Form

```
1. x = 1;
2. x = 2;
3. if(condition)
4.    x = 3;
5. read(x);
6. x = 4;
7. y = x;
```



Resulting graph when statement ordering is not considered.

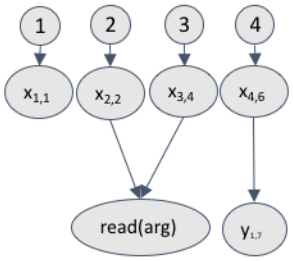
Now let's consider the ordering of

Code Transformation (after – flow sensitive): Static Single Assignment Form

1. x = 1;
2. x = 2;
3. if(condition)
4. x = 3;
5. read(x);
6. x = 4;
7. y = x;

1. $x_{1,1}$ = 1;
2. $x_{2,2}$ = 2;
3. if(condition)
4. $x_{3,4}$ = 3;
5. read($x_{2,2,3,4}$);
6. $x_{4,6}$ = 4;
7. $y_{1,7}$ = $x_{4,6}$;

Note: <Def#,Line#>



Exercise: Counting Uses and Definitions

- Can the number of live definitions for a given use grow exponentially with the number of paths?
  - No. Consider *n* non-nested branches. A re-definition kills the previous definition, so only *2* live definitions can survive for all 2^n paths. The most definitions that could survive for n branches is *n+1* definitions in the case of each path on *n* nested branches.
- Can the number of uses for a given definition grow exponentially with the number of paths?
  - Yes. Consider *n* non-nested branches. A use of a definition made before the first branch can be added on all 2^n paths.

Points-to Analysis / Alias Analysis

- How can we statically answer whether or not a variable *a* and variable *b* point to the same value in memory?

```
if(a == b){ crash(); }
```

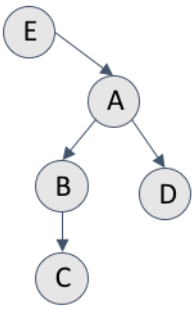
Anderson's Algorithm

- Flow-Insensitive, Context-Insensitive algorithm
- Pointer Assignments as set constraints.
- $v_1, v_2, v_3, \ldots v_n$ denote pointers pointing to objects $O_1, O_2, O_3 \ldots O_n$. $P(v_i)$ denotes points-to-set of v_i.
- Rules
  - Initialize $P(v_i) = \{O_i\}\ \forall\ 1 \leq i \leq n$
  - If $v_i = v_j$; then $P(v_i) \supseteq P(vj)$
  - If $v_i = v_j$ and $v_j = v_k$; then $P(v_i) \supseteq P(v_i) \cup P(v_j) \cup P(v_k)$

Anderson's Algorithm: Complexity

- $O(n^3)$
- Explanation :
  - No. of assignments to consider is $O(n^2)$
  - Rule 2 requires iterating over all assignments for each pointer : O(n)
  - Hence complexity is $O(n^2*n) = O(n^3)$

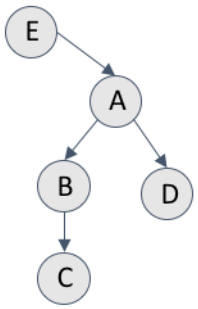
Complexity (Visual Explanation)



For each assignment we must check subset constraints.

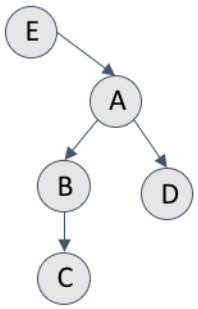
- C=B; so P(C) ⊇ P(B)
- B=A; so P(B) ⊇ P(A)
- D=A; so P(D) ⊇ P(A)

Complexity (Visual Explanation)



If the points-to set of A changes, then the points-to sets of B,C, and D can change. In the worst case we must traverse the entire graph, which is $O(n^2)$ complexity.

Complexity (Visual Explanation)



Since assignments can be processed in any order, we must consider the worst case where *every* assignment causes us to re-traverse the entire graph.

Consider A=E;

In the worst case there can be n re-traversals of the graph. So, $n*n^2 = O(n^3)$.

Basic Anderson's Algorithm in Atlas

Local Points-to Analysis Example in ~30 lines of code

1. Assign a unique ID to each "new" allocation
2. For each assignment, propagate the IDs on the right hand side of the assignment to the left hand side of the assignment
3. Repeat until no IDs can be propagated (fixed point)

Key Atlas Features:

- Tags (quickly locate allocation sites)
- Attributes (storage of points-to set information)
- Data Flow Edges (assignments)
- Static single assignment form (captures local statement ordering)

Basic Anderson's Algorithm in Atlas

1. Assign a unique ID to each "new" allocation
   - Let a unique ID be a unique integer for each allocation site
   - Store IDs in an Atlas Node attribute containing a set of integers
   - New allocations are tagged with XCSG.Instantiation

```
int id = 0;
LinkedList<Node> worklist = new LinkedList<Node>();

AtlasSet<Node> instantiations =
universe.nodesTaggedWithAny(XCSG.Instantiation).eval().nodes();

for(Node instantiation : instantiations){
   Set<Integer> pointsToIds = getPointsToSet(instantiation);
   pointsToIds.add(id++);
   worklist.add(instantiation);
}
```

```
// helper method to create/access points-to sets
Set<Integer> getPointsToSet(Node node){
   String P2ID = "points-to-set";
   if(ge.hasAttr(P2ID)){
      return (HashSet<Integer>) node.getAttr(P2ID);
   } else {
      Set<Integer> pointsToIds = new HashSet<Integer>();
      node.putAttr(P2ID, pointsToIds);
      return pointsToIds;
   }
}
```

Basic Anderson's Algorithm in Atlas

2. For each assignment, propagate the IDs on the right hand side of the assignment to the left hand side of the assignment
   - Atlas represents assignments as data flow edges (from -> to)
   - LHS points-to set must be a superset of the RHS points-to set

```
void propagatePointsTo(Node from){
   Q dataFlowEdges universe.edgesTaggedWithAny(XCSG.LocalDataFlow);
   AtlasSet<Node> toNodes = dataFlowEdges.successors(Common.toQ(from)).eval().nodes();
   for(Node to : toNodes){
      Set<Integer> fromPointsToSet = getPointsToSet(from);
      Set<Integer> toPointsToSet = getPointsToSet(to);
      if(toPointsToSet.addAll(fromPointsToSet)){
         worklist.add(to);
      }
   }
}
```

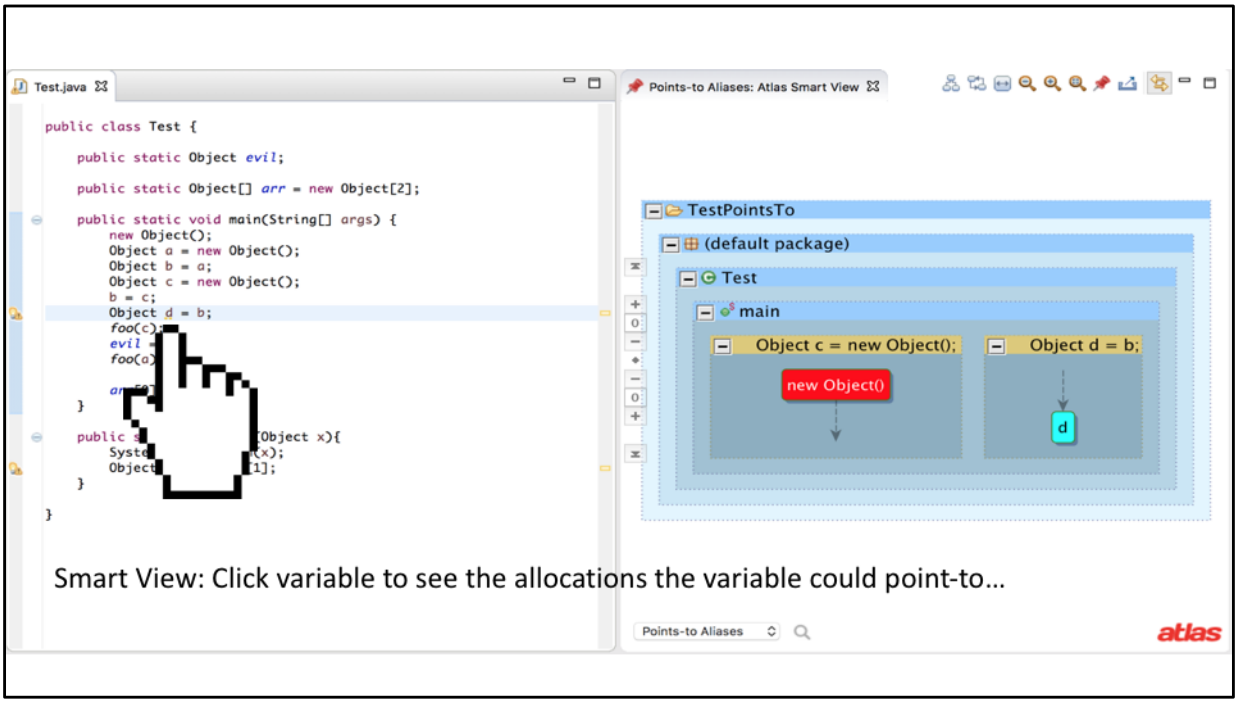
Basic Anderson's Algorithm in Atlas

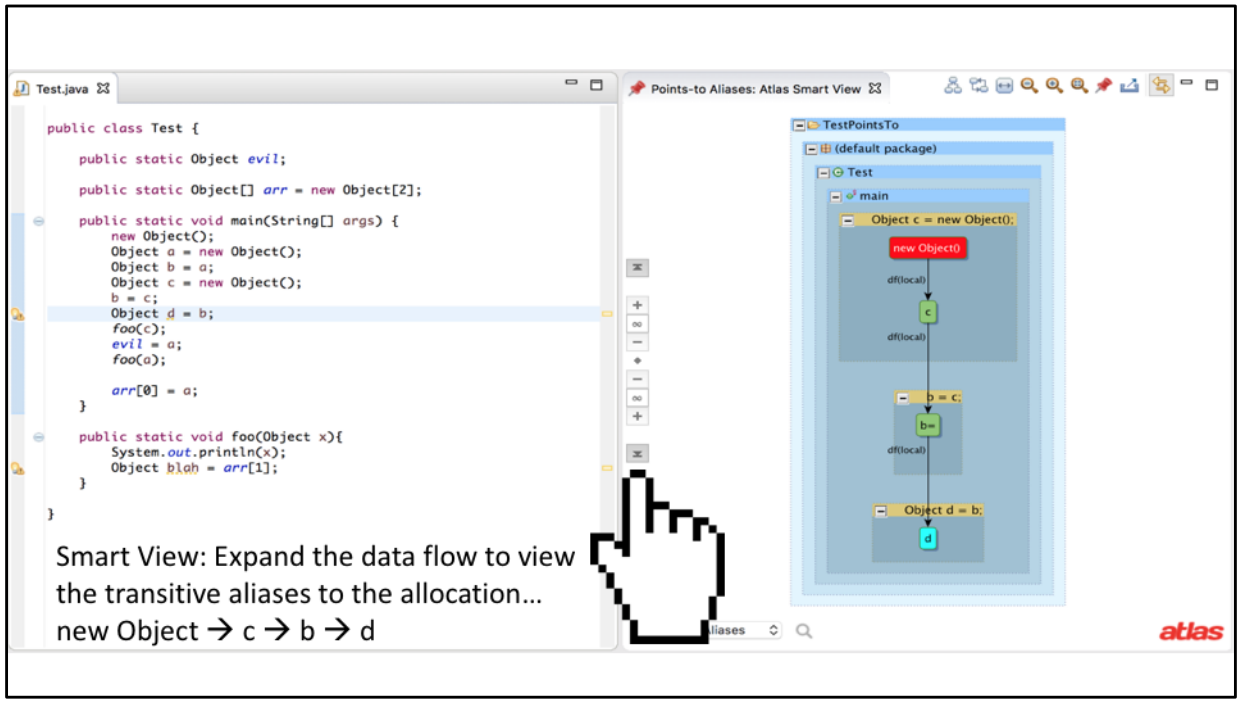
3. Repeat until no IDs can be propagated (fixed point)

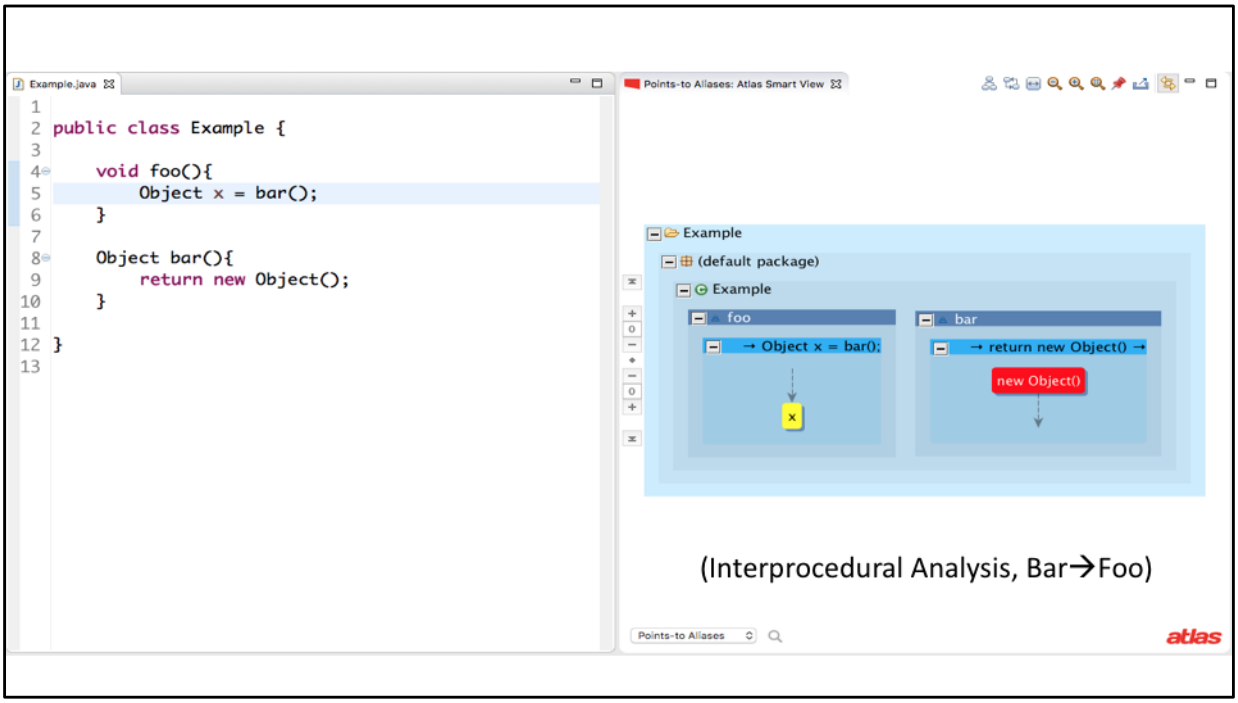
```
// keep propagating allocation ids forward along assignments
// until there is nothing more to propagate
while(!worklist.isEmpty()){
   Node from = worklist.removeFirst();
   propagatePointsTo(from);
}
```

Advanced Anderson's Algorithm in Atlas

- Consider resolving dynamic dispatches on-the-fly for *interprocedural* data flow
  - Variable runtime type must be known to resolve dispatch target
- Consider more efficient set storage (ex: BDDs)
- Consider array component access
- Implementation: https://github.com/EnSoftCorp/points-to-toolbox


Test.java
public class Test {
public static Object evil;
public static Object[] arr = new Object[2];
public static void main(String[] args) {
new Object();
Object a = new Object();
Object b = a;
Object c = new Object();
b = c;
Object d = b;
foo(c);
evil
foo(a)
}
}
Points-to Aliases: Atlas Smart View
TestPointsTo
(default package)
Test
main
Object c = new Object();
Object d = b;
new Object()
d
Points-to Aliases
atlas
Smart View: Click variable to see the allocations the variable could point-to…


Test.java
public class Test {
public static Object evil;
public static Object[] arr = new Object[2];
public static void main(String[] args) {
new Object();
Object a = new Object();
Object b = a;
Object c = new Object();
b = c;
Object d = b;
foo(c);
evil = a;
foo(a);
arr[0] = a;
}
public static void foo(Object x){
System.out.println(x);
Object blah = arr[1];
}
}
Smart View: Expand the data flow to view the transitive aliases to the allocation…
new Object → c → b → d
Points-to Aliases: Atlas Smart View
TestPointsTo
(default package)
Test
main
Object c = new Object();
new Object()
df(local)
c
df(local)
b = c;
b=
df(local)
Object d = b;
d
atlas


Example.java
1
2 public class Example {
3
4 void foo(){
5 Object x = bar();
6 }
7
8 Object bar(){
9 return new Object();
10 }
11
12 }
13
Points-to Aliases: Atlas Smart View
Example
(default package)
Example
foo
Object x = bar();
x
bar
return new Object()
new Object()
(Interprocedural Analysis, Bar→Foo)
Points-to Aliases
atlas

Program Slicing Motivating Example

- If the output is wrong at line 9, do we have to consider every line of the program to find the bug?

Example:

```
1. x = 2;
2. y = 3;
3. z = 7;
4. a = x + y;
5. b = x + z;
6. a = 2 * x;
7. c = y + x + z;
8. t = a + b;
9. print(t); // t is wrong value!
```

Program Slicing

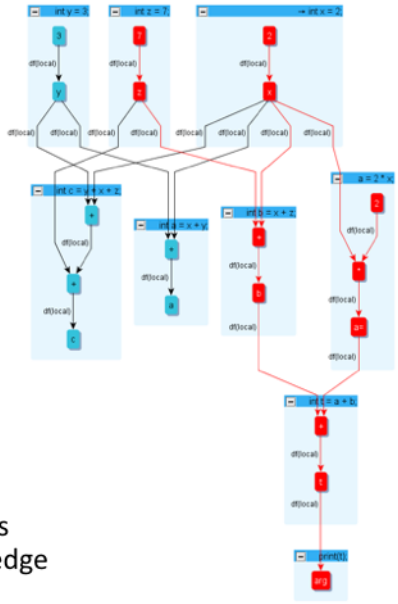
- Data Dependence
- Control Dependence
- Taint Analysis

Data Flow Graph

Example:

1. x = 2;
2. y = 3;
3. z = 7;
4. a = x + y;
5. b = x + z;
6. a = 2 * x;
7. c = y + x + z;
8. t = a + b;
9. print(t); ← detected failure

Relevant lines:
1,3,5,6,8

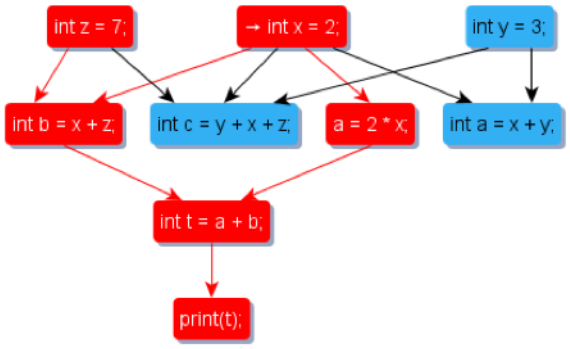


What lines must we consider if the value of *t* printed is incorrect?

- A *Data Flow Graph* (DFG) creates a graph of primitives and variables where each assignment represents an edge from the RHS to the LHS of the assignment.

Data Dependence Graph

- Note that we could summarize data flow on a per statement level.
- This graph is called a *Data Dependence Graph* (DDG)



Data Dependence Slicing

- Reverse Data Dependence Slice
  - What statements influence the assigned value in this statement?
- Forward Data Dependence Slice
  - What statements could the assigned value in this statement influence?

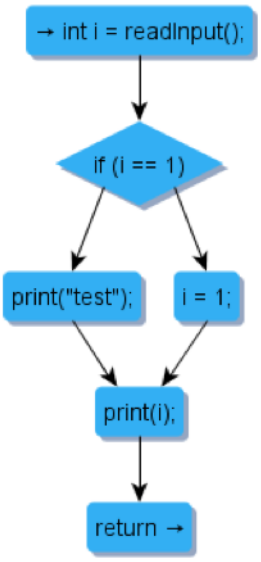
Control Flow Graph

Example:

```
1. i = readInput();
2. if(i == 1)
3.     print("test");
   else
4.     i = 1;
5. print(i);
6. return; // terminate
```

Relevant lines: 1,2,4

detected failure

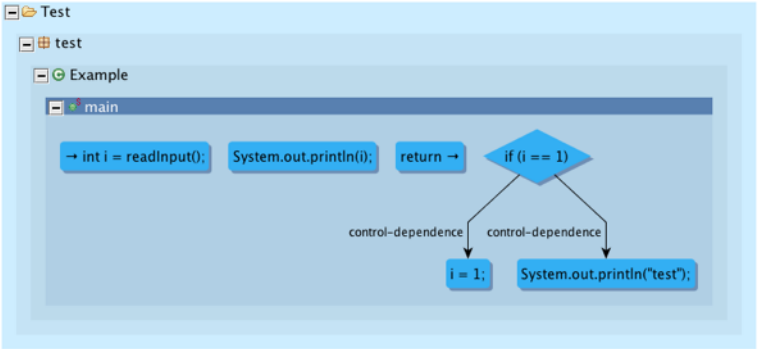


What lines must we consider if the value of *i* printed is incorrect?

- A *Control Flow Graph* (CFG) represents the possible sequential execution orderings of each statement in a program.
- Data flow influences control flow, so this graph is not enough.

Control Dependence Graph

- If a statement X determines whether a statement Y can be executed then statement Y is *control dependent* on X.

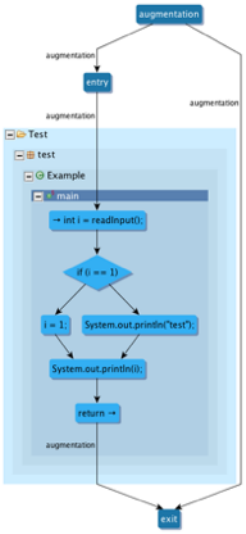


Control Dependence Graph (CDG)

- If a statement X determines whether a statement Y can be executed then statement Y is control dependent on X.
  - Classic algorithm: Ferrante, Ottenstein, and Warren (FOW) algorithm

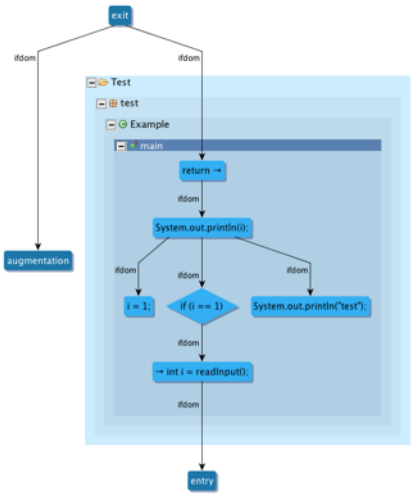
Building a CDG (1)

- First augment the CFG with a single “entry” node and single “exit” node.
- Create an “augmentation” node which has the “entry” and “exit” nodes as children.



Building a CDG (2)

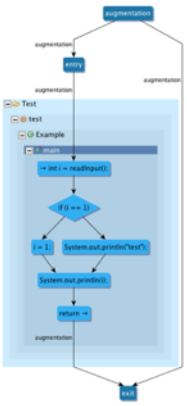
- X *dominates* Y if every path from the entry node to Y must go through X
- A *dominator tree* is a tree where each node's children are those nodes it immediately dominates
- Compute a forward dominance tree (i.e. post-dominance analysis) of the augmented CFG



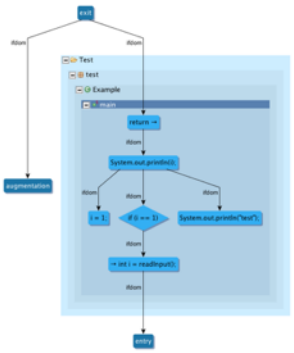
Building a CDG (3)

- The *least common ancestor* (LCA) of two nodes X and Y is the deepest tree node that has both X and Y as descendants
- For each edge (X → Y) in CFG, find nodes in FDT from LCA(X,Y) to Y, which are *control dependent* on X.
  - Exclude LCA(X,Y) if LCA(X,Y) is not X

Augmented CFG (ACFG)



Forward Dominance Tree (FDT)



Building a CDG (4)

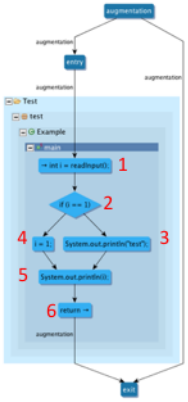
Edge X→Y in ACFG	LCA(X,Y) in FDT	FDT Nodes Between(LCA, Y)
1 → 2	2	~~2~~
2 → 3	5	~~5~~, 3
2 → 4	5	~~5~~, 4
4 → 5	5	~~5~~
3 → 5	5	~~5~~
5 → 6	6	~~6~~

Note: Remove LCA(X,Y) if LCA(X,Y) != X

Example:

1. i = readInput();
2. if(i == 1)
3. print("test");
   else
4. i = 1;
5. print(i);
6. return; // terminate program

Augmented CFG (ACFG)

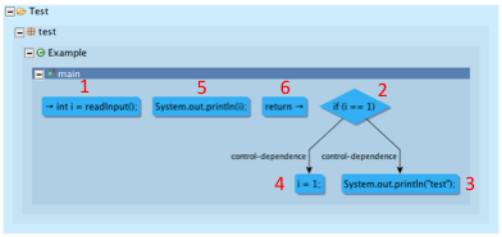


Forward Dominance Tree (FDT)

Control Dependence Graph

Edge X→Y in ACFG	LCA(X,Y) in FDT	FDT Nodes Between(LCA, Y)
1 → 2	2	~~2~~
2 → 3	5	~~5~~, 3
2 → 4	5	~~5~~, 4
4 → 5	5	~~5~~
3 → 5	5	~~5~~
5 → 6	6	~~6~~

FDT Nodes Between(LCA, Y) are *Control Dependent* on X.



Control Dependence Slicing

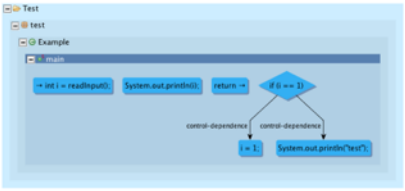
- Reverse Control Dependence Slice
  - What statements does this statement's execution depend on?
- Forward Control Dependence Slice
  - What statements could execute as a result of this statement?

Control Dependence Slicing

Example:

```
1. i = readInput();
2. if(i == 1)
3.   print("test");
   else
4.   i = 1;
5. print(i);
6. return; // terminate program
```

detected failure



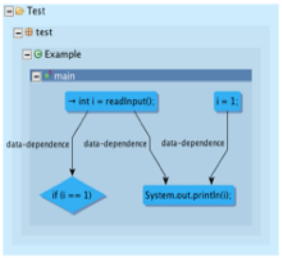
What lines must we consider if the value of *t* printed is incorrect?

Is the CDG sufficient to answer the question?

Program Dependence Graph

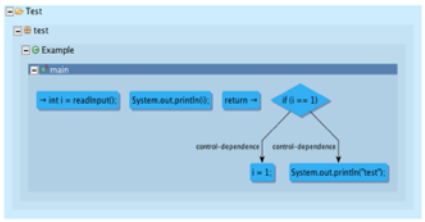
- Both DDG and CDG nodes are statements
- The union of a DDG and the CDG is a PDG

DDG



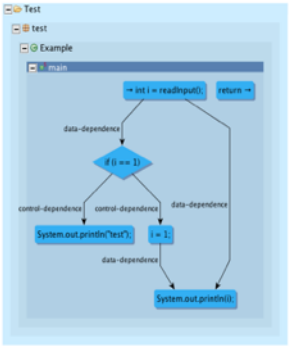
+

CDG



=

PDG



Program Slicing (Impact Analysis)

- Reverse Program Slice

  Answers: What statements does this statement's execution depend on?

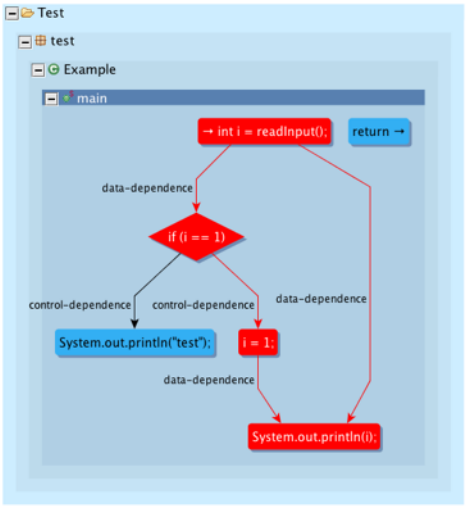
- Forward Program Slice

  Answers: What statements could execute as a result of this statement?

Example:

```
1. i = readInput();
2. if(i == 1)
3.     print("test");
   else
4.     i = 1;
5. print(i);   <-- detected failure
6. return; // terminate
```

Relevant lines: 1,2,4

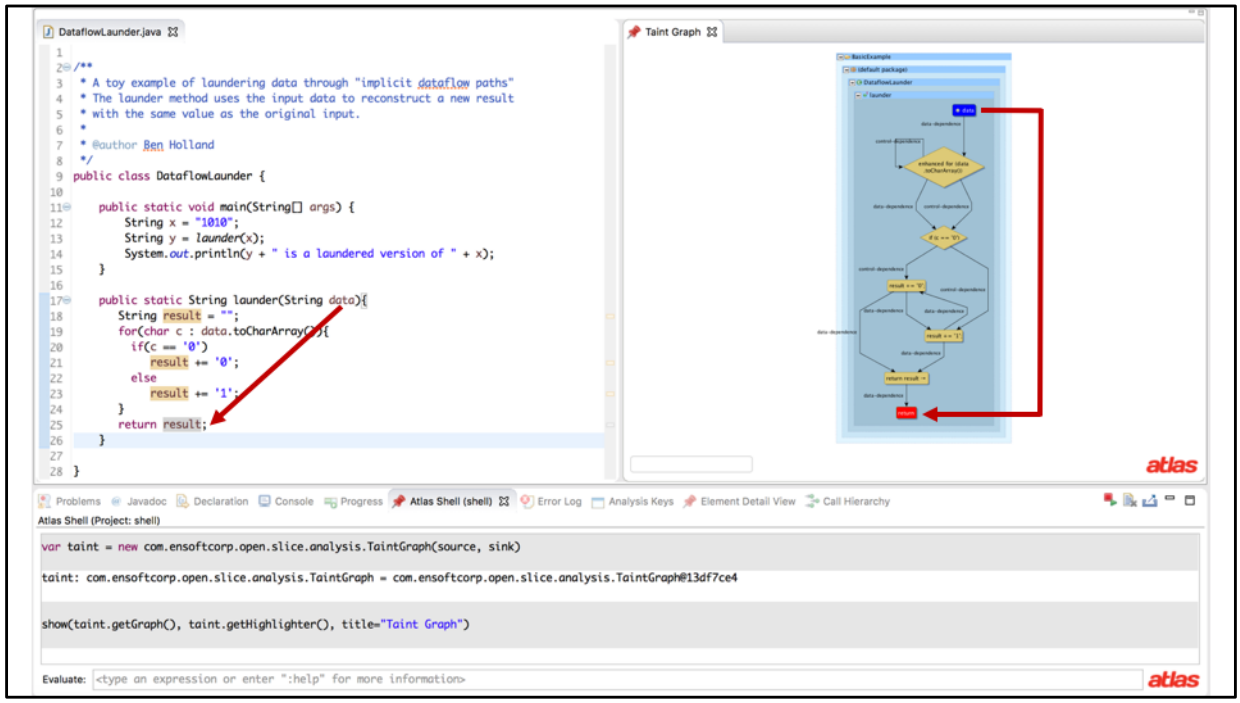


Taint Analysis

How can we track the flow of data from the source (*x*) to the sink (*y*)?

- Taint = (forward slice of *source)* intersection (reverse slice of *sink)*
- If a path exists from *source* to *sink* then the source *taints* the sink.

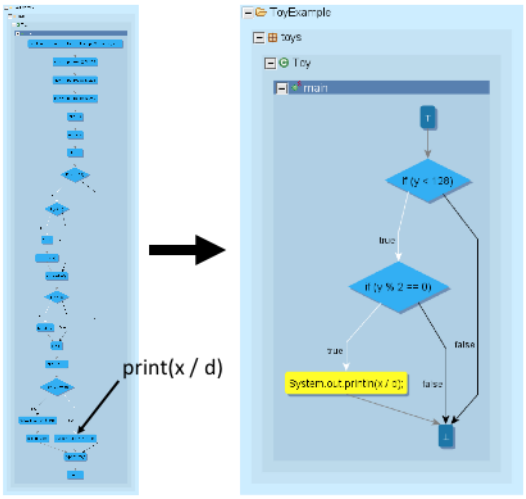
```
public class DataflowLaunder {

        public static void main(String[] args) {
                String x = "1010";
                String y = launder(x);
                System.out.println(y + " is a laundered version of " + x);
        }

        public static String launder(String data){
           String result = "";
           for(char c : data.toCharArray()){
             if(c == '0')
                result += '0';
             else
                result += '1';
           }
           return result;
        }

}
```



There exists a path from the *launder* method's input *data* parameter to it's return *result*, so we can say *result* is tainted by *data*. Since *x* is passed to the *data* parameter which taints *result* and the return value is assigned to *y*, we know that *x* taints *y*.

Concept: Projected Control Graph

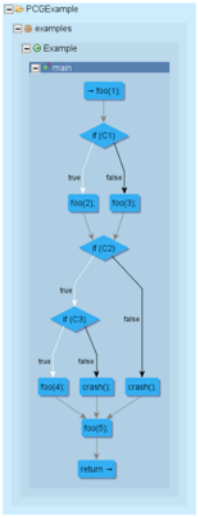
- Projected Control Graph (PCG)
  - Proposed by Ahmed Tamrawi in 2016
  - Efficiently groups program behaviors into equivalence classes of homomorphic behaviors
    - Parameterized by events of interest
    - Only event statements and necessary conditions are retained
  - Result is a compact structure-preserving model of a CFG w.r.t. events of interest



CFG PCG(E), E=Division Statement

Projected Control Graph

```
1  public void main() {
2     foo(1);
3     if (C1) {
4        foo(2);
5     } else {
6        foo(3);
7     }
8     if (C2) {
9        if (C3) {
10          foo(4);
11       } else {
12          crash();
13       }
14    } else {
15       crash();
16    }
17    foo(5);
18    return;
19 }
```



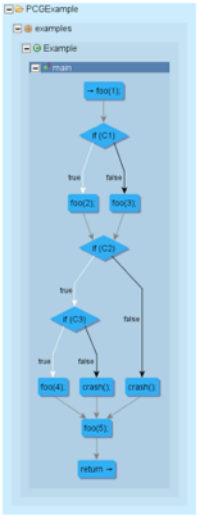
2^3=8 possible values for the tuple (C1, C2, C3)

C1	C2	C3	Behavior
False	False	False	
False	False	True	
False	True	False	
False	True	True	
True	False	False	
True	False	True	
True	True	False	
True	True	True	

73

Projected Control Graph

```
1  public void main() {
2     foo(1);
3     if (C1) {
4        foo(2);
5     } else {
6        foo(3);
7     }
8     if (C2) {
9        if (C3) {
10          foo(4);
11       } else {
12          crash();
13       }
14    } else {
15       crash();
16    }
17    foo(5);
18    return;
19 }
```



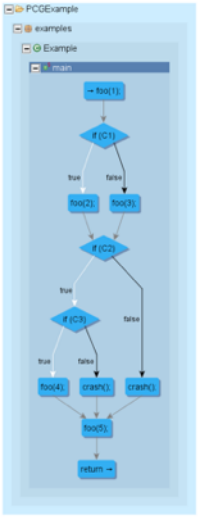
If C2 is false then C3 is not evaluated.

C1	C2	C3	Behavior
False	False	N/A	
False	False	N/A	
False	True	False	
False	True	True	
True	False	N/A	
True	False	N/A	
True	True	False	
True	True	True	

74

Projected Control Graph

```
1  public void main() {
2     foo(1);
3     if (C1) {
4        foo(2);
5     } else {
6        foo(3);
7     }
8     if (C2) {
9        if (C3) {
10          foo(4);
11       } else {
12          crash();
13       }
14    } else {
15       crash();
16    }
17    foo(5);
18    return;
19 }
```



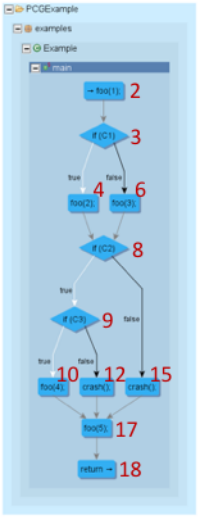
CFG has 6 paths.

C1	C2	C3	Behavior
False	False	N/A	
False	True	False	
False	True	True	
True	False	N/A	
True	True	False	
True	True	True	

75

Projected Control Graph

```
1  public void main() {
2     foo(1);
3     if (C1) {
4        foo(2);
5     } else {
6        foo(3);
7     }
8     if (C2) {
9        if (C3) {
10          foo(4);
11       } else {
12          crash();
13       }
14    } else {
15       crash();
16    }
17    foo(5);
18    return;
19 }
```



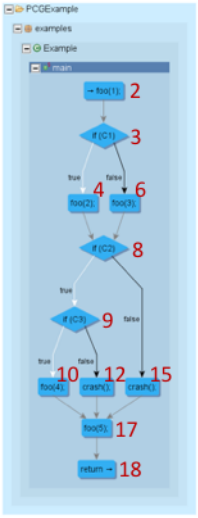
What paths include a “crash” event?

C1	C2	C3	Behavior
False	False	N/A	2,3,6,8,15,17,18
False	True	False	2,3,6,8,9,12,17,18
False	True	True	2,3,6,8,9,10,17,18
True	False	N/A	2,3,4,8,15,17,18
True	True	False	2,3,4,8,9,12,17,18
True	True	True	2,3,4,8,9,10,17,18

76

Projected Control Graph

```
1  public void main() {
2     foo(1);
3     if (C1) {
4        foo(2);
5     } else {
6        foo(3);
7     }
8     if (C2) {
9        if (C3) {
10          foo(4);
11       } else {
12          crash();
13       }
14    } else {
15       crash();
16    }
17    foo(5);
18    return;
19 }
```



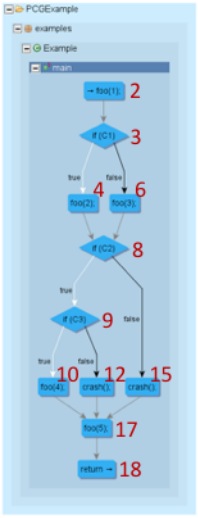
4 of 6 behaviors have “crash” events.
2 of 6 behaviors do not have “crash” events.

C1	C2	C3	Behavior
False	False	N/A	2,3,6,8,15,17,18
False	True	False	2,3,6,8,9,12,17,18
~~False~~	~~True~~	~~True~~	~~2,3,6,8,9,10,17,18~~
True	False	N/A	2,3,4,8,15,17,18
True	True	False	2,3,4,8,9,12,17,18
~~True~~	~~True~~	~~True~~	~~2,3,4,8,9,10,17,18~~

77

Projected Control Graph

```
1  public void main() {
2     foo(1);
3     if (C1) {
4        foo(2);
5     } else {
6        foo(3);
7     }
8     if (C2) {
9        if (C3) {
10          foo(4);
11       } else {
12          crash();
13       }
14    } else {
15       crash();
16    }
17    foo(5);
18    return;
19 }
```



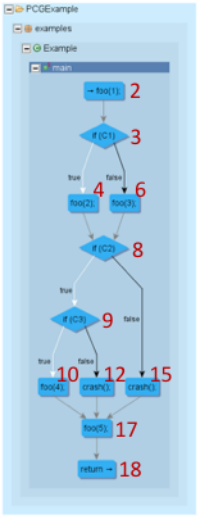
Consider necessary conditions for "crash" events.

C1	C2	C3	Behavior
False	False	N/A	2,3,6,8,15,17,18
False	True	False	2,3,6,8,9,12,17,18
~~False~~	~~True~~	~~True~~	~~2,3,6,8,9,10,17,18~~
True	False	N/A	2,3,4,8,15,17,18
True	True	False	2,3,4,8,9,12,17,18
~~True~~	~~True~~	~~True~~	~~2,3,4,8,9,10,17,18~~

78

Projected Control Graph

```
1  public void main() {
2     foo(1);
3     if (C1) {
4        foo(2);
5     } else {
6        foo(3);
7     }
8     if (C2) {
9        if (C3) {
10          foo(4);
11       } else {
12          crash();
13       }
14    } else {
15       crash();
16    }
17    foo(5);
18    return;
19 }
```



There are 3 unique behaviors w.r.t. “crash” events.

C1	C2	C3	Homomorphic Behavior
False	False	N/A	8,15
False	True	False	8,9,12
~~False~~	~~True~~	~~True~~	~~8,9~~
True	False	N/A	8,15
True	True	False	8,9,12
~~True~~	~~True~~	~~True~~	~~8,9~~

B1 B2 B3

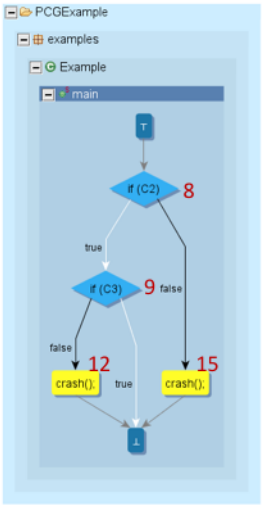
79

Homomorphism: In algebra, a homomorphism is a structure-preserving map between two algebraic structures of the same type (such as two groups, two rings, or two vector spaces). The word homomorphism comes from the ancient Greek language: ὁμός (homos) meaning "same" and μορφή (morphe) meaning "form" or "shape".

x*y=z => f(x) * f(y) = f(z)
=> f(x) * f(y) = f(x * y)

Projected Control Graph

```
1  public void main() {
2     foo(1);
3     if (C1) {
4        foo(2);
5     } else {
6        foo(3);
7     }
8     if (C2) {
9        if (C3) {
10          foo(4);
11       } else {
12          crash();
13       }
14    } else {
15       crash();
16    }
17    foo(5);
18    return;
19 }
```



2 of 3 homomorphic behaviors (B1, B2) have "crash" events.
1 of 3 homomorphic behaviors (B3) are void w.r.t. "crash" events.

C1	C2	C3	Homomorphic Behavior
False	False	N/A	8,15
False	True	False	8,9,12
~~False~~	~~True~~	~~True~~	~~8,9~~
True	False	N/A	8,15
True	True	False	8,9,12
~~True~~	~~True~~	~~True~~	~~8,9~~

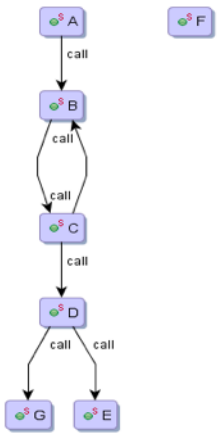
B1
B2
B3

80

Exercise: Inter-procedural Communication

- How can two functions communicate data? Hint: Consider two types of program memory.
  - Reading from and writing to global variables (heap memory)
  - Parameter passes (stack memory)
  - Return values (stack memory)

How do we produce a call graph?



82

Idea 1: Reachability Analysis (RA)

- For Every Method ***m***
  - For Every Callsite
    - Add an edge (if one does not already exist) from ***m*** to any method with the same name as the method called in the callsite
- Refinement: Match methods with the same name AND the same return types and parameter types/counts
- Sound but not precise
  - We end up with a call graph that always has a call edge that *could* happen, but we have a lot of edges that also can not happen.
  - WHY? -> Static Dispatch vs. Dynamic Dispatch

83

Static Dispatch vs. Dynamic Dispatch

- Static Dispatch
  - Resolvable at compile time
  - Includes calls to Constructors and methods marked “static” (ex: main method)
  - Static methods do not require an object instance, they can be called directly

  Examples

  Animal a = new Dog(); // static dispatch to new Dog()
  Animal.runSimulation(a); // static dispatch to runSimulation()

Static Dispatch vs. Dynamic Dispatch

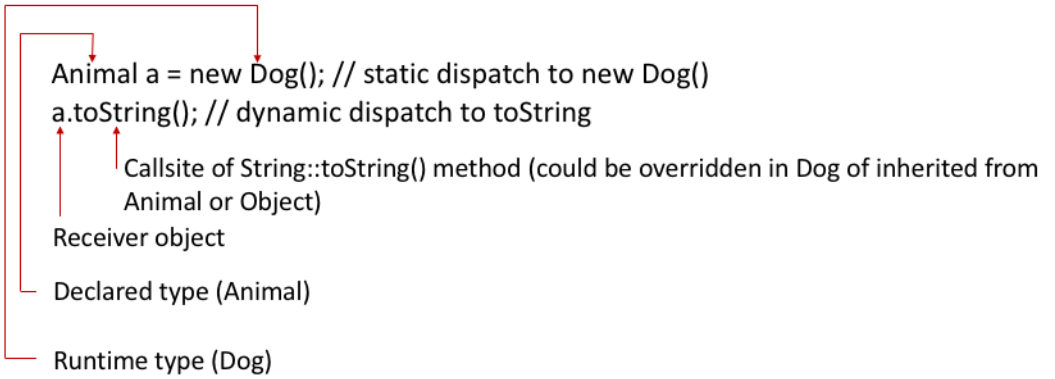
- Dynamic Dispatch
  - Resolvable at runtime
  - Includes calls to member methods (virtual methods)
  - Requires an object instance, they can be called directly
  - Very common in OO languages such as Java

  Examples

  ```
  Animal a = new Dog(); // static dispatch to new Dog()
  a.toString(); // dynamic dispatch to toString
  a.getName(); // dynamic dispatch to dog's getName method
  ```

85

Terminology (slight digression)



86

Idea 2: Class Hierarchy Analysis (CHA)

- Compute the type hierarchy
- For each callsite, if the dispatch is static add the edge like normal, otherwise:
  - Perform RA with the added constraint that the target method must be in either
    - 1) The direct lineage from Object to the receiving object's declared type (in the case that the target method is inherited)
    - 2) The subtype hierarchy of the receiving object's declared type
- Sound and more precise than RA
  - We end up with a call graph that always has a call edge that *could* happen, and we have less edges that can not happen.
  - We can still do better…in terms of precision
  - Checkout the case of: Object o = …; o.toString();
    - We have to add an edge to every toString() method!

87

Idea 3: Rapid Type Analysis (RTA)

- Summary:
  - Look at the allocation types that were made
  - We can't have a call to a method in a type that we never had an instance of (in the case of dynamic dispatches)
- Implementation
  - Start with CHA
  - Worklist style algorithm starting at the main method
    - Gather new allocation types
    - For each allocation type that matches a type in the CHA call graph, add a new edge
- Described in detail in "Fast Static Analysis of C++ Virtual Function Calls – IBM", 1996.
  - More precise than CHA in practice, but unsound!

88

Idea 4: Rapid Type Analysis Improvements

- Method Type Analysis (MTA)
  - Idea: Restrict parent method types to types that could be passed through the method's parameters
  - Idea: Consider the statically typed return type of the dynamic dispatch callsite
- Field Type Analysis (FTA)
  - Idea: Consider that a method could write to a global variable, so any allocations reachable by a method are also reachable by a method that reads the same global variable
- Hybrid Type Analysis (XTA)
  - Combines MTA and FTA
  - Precision? More precise than RTA
  - Sound? No…Exceptions are not considered
- Paper: Scalable Propagation-Based Call Graph Construction Algorithms

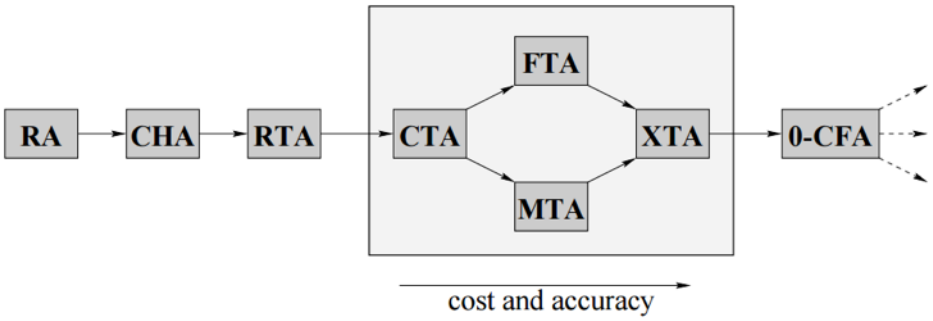
89

Idea 5: Variable Type Analysis

- Idea: Track the allocation types to variables the callsites are made on.
  - This is a points-to analysis
- Implementation (one strategy)
  - For each new allocation, assign an ID to the new allocation site add each allocation site to the worklist
  - While the worklist is not empty
    - Remove an item from the worklist and propagate its point-to set (set of allocation ids) to every data flow successor (every assignment of the variable to another variable), adding each new variable to the worklist
    - If a callsite is made on the variable, look up the type of the allocation(s) and add a call edge (if parameters are reachable as a result, add the parameters to the worklist)

90

Call Graph Construction Algorithm Overview

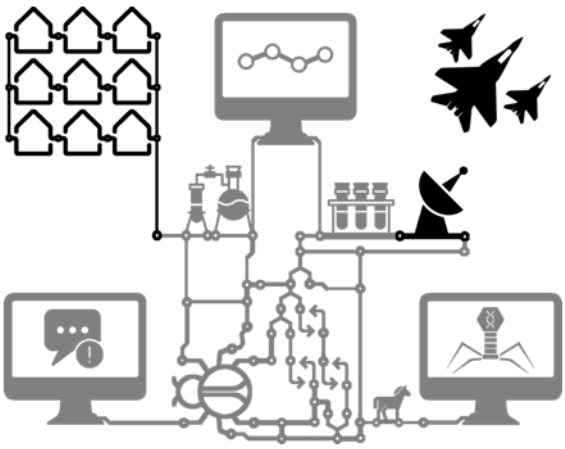


91

Things to think about…

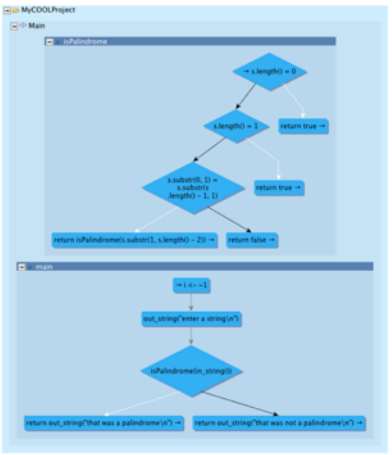
- Can we create a call graph of a library without an application?
  - How should we deal with “callback edges”, that is the case when an application overrides a method in the library so that a dynamic dispatch callsite in the library actually goes to the application.
  - If we don’t have all of the code we can’t know what happens at runtime, but we can say something about what can’t happen.
- How does Java Reflection affect call graph construction?
  - Java Reflection is a skeleton in the closest for most researchers, in most cases its not handled.

Going Beyond



Detour: COOL Program Analyzer

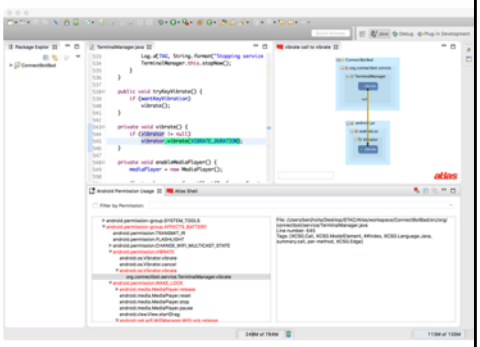
- Adding Atlas support for Stanford's Classroom Object Oriented Language (COOL)
  - COOL Compiler: Lexer → Parser → Semantic Analysis / Type Checking → ~~Code Generation~~ → Program Graph
- Current Status:
  - Eclipse integration
    - Project nature, build system
  - Atlas Program Graph Indexer
    - Type hierarchy
    - Containment relationships
    - Function / Global variable signatures
    - Function Control Flow Graph
    - Data Flow Graph (in progress)
    - Inter-procedural relationships:
      - Call Graph (implemented via compliance to XCSG!)



https://github.com/benjholla/AtlasCOOL (currently private, pending release approval)

Detour: Android Essentials

- Android Permissions
  - Maps Permission Protected Functions to Permission
  - Maps Permission to Permission Group
  - Maps Permission to Protection Level
- Extending Program Graph with XML Resources
  - Literal values for language, screen orientation, etc.
  - UI interface and event callbacks (buttons, forms, etc.)
- Intent Resolution
  - Communication between application components and even inter-application



https://github.com/EnSoftCorp/android-essentials-toolbox

Challenge with Compiled Code

- In compiled code high-level source constructs such as *for loops* and *while loops* do not exist.
  - Low-level code consists of goto's and labels

Goal: Identify loops in Control Flow Graphs (CFGs)

Motivation for Recovering Loops

- Most of the execution time is spent in loops
- 90/10 law
  - 90% of the time is spent in 10% of the code
  - 10% of the time is spent in the remaining 90% of the code

Loop Definition

A loop in the CFG:

- Has a set of child nodes
- A loop has a *loop header* such that:
  - Control to all child nodes in the loop always goes through the loop header
  - Has a back edge from one of its child nodes to the loop header

Remember

- Node X *dominates* node Y if all paths from the entry node to Y go through X
- A *depth-first search* of a graph starts at the root (CFG entry node) and explores as far as possible along each branch before backtracking.

Loop Recovery Intuitions

- Header of a loop dominates all child nodes in loop body
- Back edges are edges whose heads dominate their tails
  - An edge X→Y such that Y dominates X
- Loop identification is essentially back edge identification

Loop Recovery Algorithm

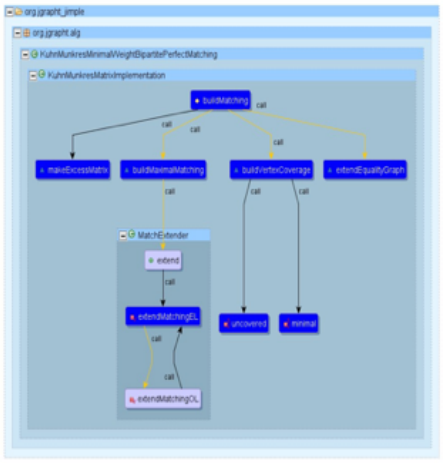
```
foreach node H in dominator tree
        foreach node N such that ∃ an edge N→H
                define loop:
                        header = H
                        back edge = N→H
                        loop body = nodes found in a backwards
                         DFS traversal from N to H
```

Loop Recovery Algorithm

- DLI algorithm described in [1] presents an efficient algorithm for identifying loops in irreducible graphs.

[1] Wei, Tao, et al. "A new algorithm for identifying loops in decompilation." International Static Analysis Symposium. Springer Berlin Heidelberg, 2007.

Loop Call Graph



Called Inside Loop

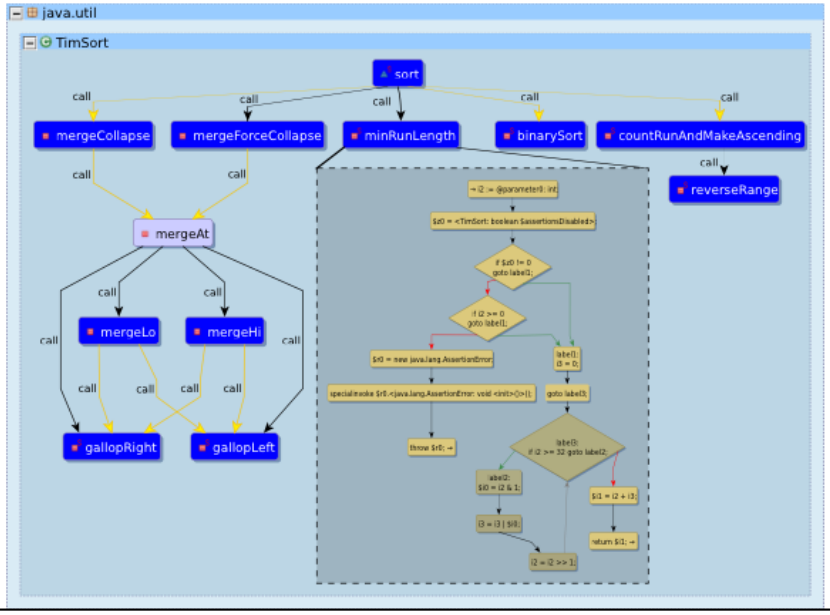
Called Outside Loop

Nodes:
- Methods containing loops (blue)
- Methods reaching methods containing loops (white)

Edges:
- Call relationships
- Color attributes to show placement of call site in loop

55K LOC

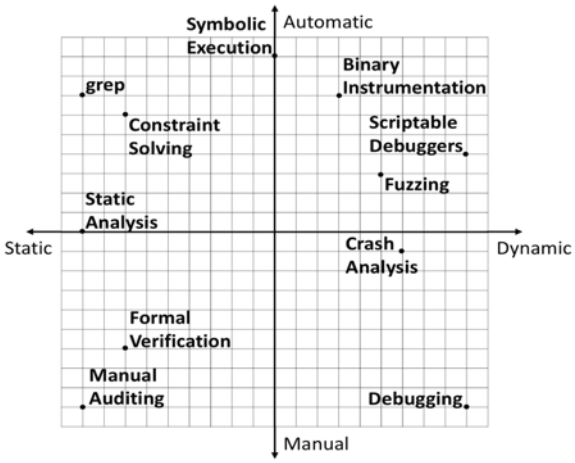
Loop Call Graph



55K LOC

Exploring Recent Trends in Program Analysis

A Spectrum of Program Analysis Techniques



Source: Contemporary Automatic Program Analysis, Julian Cohen, Blackhat 2014

Symbolic Execution

- Replace concrete assignment values with symbolic values
- Perform operations on symbolic values abstractly
- At each branch fork the abstracted logic
  - Dealing with path explosion problem is a challenge
- Utilize SAT/SMT solvers to determine if the constraints are satisfiable for a path of interest
  - Example: fail occurs if y * 2 = z = 12 is satisfiable
    - Solve(y * 2 = 12, y), y = 6 satisfies the constraint
    - Failure occurs when read() returns 6

```
int f() {
 ...
 y = read();
 z = y * 2;
 if (z == 12) {
  fail();
 } else {
  printf("OK");
 }
}
```

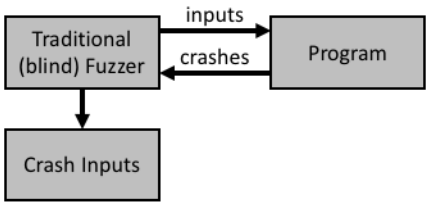
On what inputs does the code fail?

Concolic Execution

- A hybrid of dynamic analysis and symbolic execution
  - Perform symbolic execution on some variables and concrete execution on other variables
  - Symbolic variables could be made concrete in order to:
    - Move past symbolic limitations such as challenges in modeling the program environment (example network interaction)
    - Deal with path explosion problem and satisfiability problem by replacing difficult symbolic values with concrete values to simplify analysis
  - Pays cost in time for symbolic computations and execution time of program
- Several well known tools:
  - Angr - http://angr.io
  - KLEE - https://klee.github.io
  - DART - https://dl.acm.org/citation.cfm?id=1065036
  - CREST (formerly CUTE) - https://code.google.com/archive/p/crest
  - Microsoft SAGE - https://patricegodefroid.github.io/public_psfiles/ndss2008.pdf

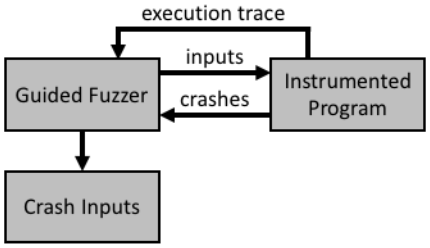
Traditional/Dumb (Blind) Fuzzing

- Start with a test corpus of well formed program inputs
- Apply random or systematic mutations to program inputs
- Run program with mutated inputs and observe whether or not the program crashes
- Repeat until the program crashes



Smart (Guided) Fuzzing

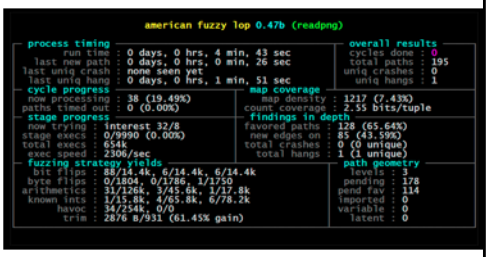
- Start with a test corpus of well formed program inputs
- Apply random or systematic mutations to program inputs
- Instrument the program branch points
- Run the instrumented program with mutated inputs and 1) observe whether or not the program crashes and 2) record the program execution path coverage
- If the input results in new program paths being explored then prioritize mutations of the tested input
- Repeat until the program crashes



Heuristics guide genetic algorithm to generate program inputs that push the fuzzer deeper into the program control flow, avoiding the common pitfalls of fuzzers to only test "shallow" code regions.

AFL (American Fuzzy Lop) Fuzzer

- Recognized as the current state of art implementation of guided fuzzing
  - Effective mutation strategy to generate new inputs from initial test corpus
  - Lightweight instrumentation at branch points
  - Genetic algorithm promotes mutations of inputs that discover new branch edges
    - Aims to explore all code paths
  - Huge trophy case of bugs found in wild
    - 371+ reported bugs in 161 different programs as of March 2018
    - http://lcamtuf.coredump.cx/afl/
- A game of economics. AFL tends to "guess" the correct input faster than a smart tool "computes" the correct input.



DARPA's Cyber Grand Challenge (CGC)

- "Cyber Grand Challenge (CGC) is a contest to build high-performance computers capable of playing in a Capture-the-Flag style cyber-security competition."

https://www.darpa.mil/program/cyber-grand-challenge

DARPA's Cyber Grand Challenge (CGC)

- Fully automatic reasoning to:
  - Detect program vulnerabilities
  - Patch programs to prevent exploitation
  - Develop and execute vulnerability exploits against competitors
- No human players!

CGC Results (Reading Between the Lines)

- All teams published the same essential combination of strategies
  - Guided fuzzing (nearly every team modified AFL)
  - Symbolic execution to assist fuzzer sometimes aided by classical program analyses (points-to, reachability, slicing, etc.)
  - Some state space pruning and prioritization scheme catered to expected vulnerability types
- Effective patches were more often generic patches which addressed the class of vulnerabilities not the one-off vulnerability that was given
  - Example: Adding stack guards for memory protection
- Competitor scores were close!
  - The difference between 1st and 7th place was not substantial
- Classes of vulnerabilities were known *a priori*

Context Matters!

- Head problems vs. hand problems
  - Design vs. implementation errors
- Implementation errors may be eventually largely eliminated with advances in programming languages, compilers, theorem provers, etc.
- Security design problems tend to be closely tied to failures in threat modeling, which is largely a human task
  - Many security problems arise due to reuse of software in changing contexts
    - Example: SCADA devices designed for isolated networks being placed on the internet

DARPA's High-Assurance Cyber Military Systems (HACMS) Program

- "formal methods-based approach to enable semi-automated code synthesis from executable, formal specifications"
- Creation of an unhackable drone!

https://www.darpa.mil/program/high-assurance-cyber-military-systems

HACMS Results (Reading Between the Lines)

- Failures tended to stem back to human failures to fully account for the attack model
  - Example: Red Team debrief noted that Red team sent a radio reboot command that dropped that shut off drone engines for 3 seconds (enough to crash the drone). Blue's response was "Oh! We didn't think of that!"
- The "unhackable" drone produced by the program was not protected from the later discovered Meltdown and Spectre exploits

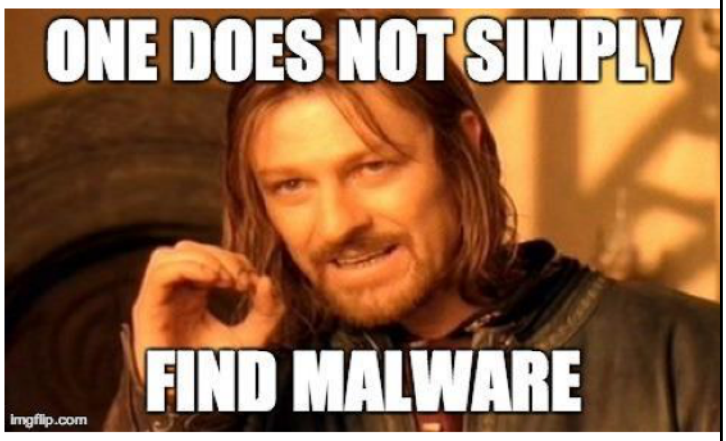
DARPA's APAC Program

- Automated Program Analysis For Cybersecurity (APAC)
- Scenario: Hardened devices, internal app store, untrusted contractors, expert adversaries
- Focused on Android

https://www.darpa.mil/program/automated-program-analysis-for-cybersecurity

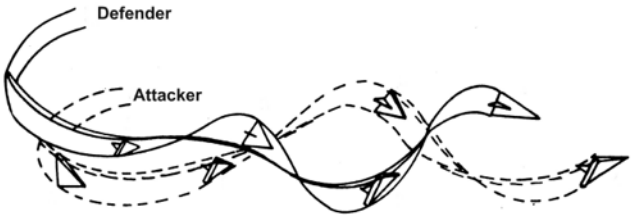
APAC Results (Reading Between the Lines)

- Is a bug malware? What if its planted intentionally? We know bug finding is hard…
- Bugs have plausible deniability and malicious intent cannot be determined from code.
- Novel attacks have escaped previous threat models.
- Need precision tools to detected **novel** and **sophisticated** malware in advance!



Program Analysis, OODA, and YOU

- "Security is a process, not a product" – Bruce Schneier



Program Analysis, OODA, and YOU

Our opponent

- Time
- Evolution of malware

"...IA > AI, that is, that intelligence amplifying systems can, at any given level of available systems technology, beat AI systems. That is, a machine and a mind can beat a mind-imitating machine working by itself." – Fred Brooks

Speeding through OODA with Atlas

```
1  public class TestClass {
2
3      public void A() {
4          B();
5      }
6
7      public void B() {
8          C();
9      }
10
11     public void C() {
12         B();
13         D();
14     }
15
16     public void D() {
17         G();
18         E();
19     }
20
21     public void E() {
22
23     }
24
25     public void F() {
26
27     }
28
29     public void G(){
30
31     }
32
33 }
```

Program Declarations, Control Flow, and Data Flow

atlas professional

Queryable Graph Database

2-way Source Correspondence