Program Analysis

- The process of automatically analyzing the behavior of computer programs regarding a property such as correctness, robustness, safety and liveness.
- Program analysis focuses on two major areas
  - program optimization: improving the program’s performance while reducing the resource usage
  - program correctness: ensuring that the program does what it is supposed to do.
- Program analysis can be performed without executing the program (static program analysis), during runtime (dynamic program analysis) or in a combination of both.

Program Analysis on Security

- Program analysis in the context of identifying security vulnerabilities and defending security attacks
- Need for program analysis
  - A software related vulnerability is essentially a bug in the software
    - Identification
  - Defending software oriented attacks
    - Software transformation
Learning Goal

- Understand the various state-of-the-art program analysis approaches

- Useful Textbooks
  - Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, *Compilers Principles, Techniques, & Tools.*

Program Representation

Why Program Representations

- Original representations
  - Source code (cross languages).
  - Binaries (cross machines and platforms).
  - Source code / binaries + test cases.
- They are hard for machines to analyze.
- Software is translated into certain representations before analyses are applied.
Outline

• Control flow graphs
• Program dependence graphs
• Super control flow graphs
• Call graph

Control Flow Graph

• The most commonly used program representation.

Program representation: Basic blocks

• A basic block in program P is a sequence of consecutive statements with a single entry and a single exit point. Thus a block has unique entry and exit points.
• Control always enters a basic block at its entry point and exits from its exit point. There is no possibility of exit or a halt at any point inside the basic block except at its exit point. The entry and exit points of a basic block coincide when the block contains only one statement.
Basic blocks: Example

Example: Computing $x$ raised to $y$

```
1 begin
2   int x, y, power;
3   float z;
4   input(x, y);
5   if (y<0)
6     power=y;
7     else
8       power=y;
9       x=1;
10  while (power != 0){
11     z=x^power;
12     power=power-1;
13  }
14  if (y<0)
15    x=1/x;
16  output(z);
17  end
```

Basic blocks: Example (contd.)

Basic blocks

<table>
<thead>
<tr>
<th>Block</th>
<th>Lines</th>
<th>Entry point</th>
<th>Exit point</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2, 3, 4, 5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>11, 12</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

Control Flow Graph (CFG)

- A control flow graph (or flow graph) G is defined as a finite set $N$ of nodes and a finite set $E$ of edges. An edge $(i, j)$ in $E$ connects two nodes $n_i$ and $n_j$ in $N$. We often write $G= (N, E)$ to denote a flow graph $G$ with nodes given by $N$ and edges by $E$. 
Control Flow Graph (CFG)

• In a flow graph of a program, each basic block becomes a node and edges are used to indicate the flow of control between blocks.
• An edge \((i, j)\) connecting basic blocks \(b_i\) and \(b_j\) implies that control can go from block \(b_i\) to block \(b_j\).
• We also assume that there is a node labeled Start in \(N\) that has no incoming edge, and another node labeled End, also in \(N\), that has no outgoing edge.

CFG Example

\[ N = \{ \text{Start, 1, 2, 3, 4, 5, 6, 7, 8, 9, End} \} \]
\[ E = \{ (\text{Start, 1}), (1, 2), (1, 3), (2, 4), (3, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 8), (7, 9), (9, \text{End}) \} \]
Consider a flow graph $G = (N, E)$. A sequence of $k$ edges, $k > 0$, $(e_1, e_2, \ldots , e_k)$, denotes a path of length $k$ through the flow graph if the following sequence condition holds.

Given that $n_p, n_q, n_r, \text{ and } n_s$ are nodes belonging to $N$, and $0 < i < k$, if $e_i = (n_p, n_q)$ and $e_{i+1} = (n_r, n_s)$ then $n_q = n_r$.

Complete path: a path from start to exit
Subpath: a subsequence of a complete path

Paths: Sample paths through the exponentiation flow graph

Two feasible and complete paths:
$p_1 = (\text{Start, 1, 2, 4, 5, 6, 5, 7, 9, End})$
$p_2 = (\text{Start, 1, 3, 4, 5, 6, 5, 7, 9, End})$

Specified unambiguously using edges:
$p_1 = (\text{(Start, 1), (1, 2), (2, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 9), (9, End)})$

Bold edges: complete path.
Dashed edges: subpath.

Paths: Infeasible paths

A path $p$ through a flow graph for program $P$ is considered feasible if there exists at least one test case which when input to $P$ causes $p$ to be traversed.

$p_1 = (\text{Start, 1, 3, 4, 5, 6, 5, 7, 8, 9, End})$
$p_2 = (\text{Start, 1, 2, 4, 5, 7, 9, End})$
Number of paths

- There can be many distinct paths through a program. A program with no condition contains exactly one path that begins at node Start and terminates at node End.
- Each additional condition in the program can increases the number of distinct paths by at least one.
- Depending on their location, conditions can have a multiplicative effect on the number of paths.

A Simplified Version of CFG

- Each statement is represented by a node
  - For readability.
  - Not for efficient implementation.

Dominator

- X dominates Y if all possible program paths from START to Y have to pass X.

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

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

\[\text{DOM}(6)=\{1,3,6\} \]
Dominator

- X strictly dominates Y if X dominates Y and X≠Y

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

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

SDOM(6)={1,3}

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

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

IDOM(6)={3}

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

Postdominator

- X post-dominates Y if every possible program path from Y to End has to pass X.
  - Strict post-dominator, immediate post-dominance.

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

SPDOM(4)={3,6} IPDOM(4)=3
### Back Edges

- A back edge is an edge whose head dominates its tail
  - Back edges often identify loops

```plaintext
1. sum=0
2. i=1
3. while (i < N) do
   4. i = i + 1
   5. sum = sum + i
5. print (sum)
```

An directed edge \((x, y)\) is considered to be directed from \(x\) to \(y\); \(y\) is called the head and \(x\) is called the tail of the arrow.

### Outline

- Control flow graphs
- Program dependence graphs
- Super control flow graphs
- Call graph

### Program Dependence Graph

- The second widely used program representation.
- Nodes are constituted by statements instead of basic blocks.
- Two types of dependences between statements
  - Data dependence
  - Control dependence
Data Dependence

- X is data dependent on Y if (1) there is a variable v that is defined at Y and used at X and (2) there exists a path of nonzero length from Y to X along which v is not re-defined.

Computing Data Dependence is Hard in General

- Aliasing
  - A variable can refer to multiple memory locations/objects.

```
1: int x, y, z ...;
2: int * p;
3:     x = ...;
4:     y = ...;
5:     p = & x;
6:     p = p + z;
```

```
1:     foo (ClassX x, ClassY y) {
2:         x.field = ...;
3:         ... = y.field;
4:     }
```

```
o1 = new ClassX();
o2 = new ClassY();
too (o1, o2);
```

Control Dependence

- Intuitively, Y is control-dependent on X iff X directly determines whether Y executes (statements inside one branch of a predicate are usually control dependent on the predicate)
  - X is not strictly post-dominated by Y
  - There is a path from X to End that does not pass Y or X = Y
  - 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
  - No such paths for nodes in a path between X and Y.

```
X
```

```
<table>
<thead>
<tr>
<th>Not post-dominated by Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Every node is post-dominated by Y</td>
</tr>
</tbody>
</table>
```
Control Dependence - Example

Y is control-dependent on X iff X directly determines whether Y executes
X is not strictly post-dominated by Y
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

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
CD(3)=3, tricky!

Control Dependence is not Syntactically Explicit

Y is control-dependent on X iff X directly determines whether Y executes
X is not strictly post-dominated by Y
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

1: sum=0
2: i=1
3: while (i<N) do
4: i=i+1
5: if (i%2==0)
6: continue;
7: sum=sum+i
endwhile
8: print(sum)

Control Dependence is Tricky!

Y is control-dependent on X iff X directly determines whether Y executes
X is not strictly post-dominated by Y
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

Can a statement control depends on two predicates?
Control Dependence is Tricky!

Y is control-dependent on X iff X directly determines whether Y executes.
  * X is not strictly post-dominated by Y
  * there exists a path from X to Y e.t. every node in the path other than X and Y is post-dominated by Y

Can one statement control depends on two predicates?

1: if (p1 || p2)
2: s1;
3: s2;

What if?
1: if (p1 && p2)
2: s1;
3: s2;

The Use of PDG

- A program dependence graph consists of control dependence graph and data dependence graph
- Why it is so important to software reliability?
  - In debugging, what could possibly induce the failure?
  - In security

```java
p=getpassword();
...
send(p);
if (p=="zhang") {
    send(m);
}
```

Outline

- Control flow graphs
- Program dependence graphs
- Super control flow graphs
- Call graph
Super Control Flow Graph (SCFG)

- Besides the normal intraprocedural control flow graph, additional edges are added connecting:
  - Each call site to the beginning of the procedure it calls.
  - The return statement back to the call site.

```
1:     for (i=0; i<n; i++) {
2:        t1= f(0);
3:        t2 = f(243);
4:        x[i] = t1 + t2 + t3;
5:    }
6:  int  f (int v) {
7:    return (v+1);
8:  }
```

Outline

- Control flow graphs
- Program dependence graphs
- Super control flow graphs
- Call graph

Call Graph (CG)

- Each node represents a function; each edge represents a function invocation

```
void A( ) {
 B( );
 C( );
}

void B( ) {
 L1:  D( );
 L2:  D( );
}

void C ( ) {
 D1( );
 A1( );
}

void D () {
 D1( );
 A1( );
}
```
### Use of CG
- CFI (control flow integrity)
- Android framework access control inconsistencies
- Hidden behavior detection in Android apps

### Many Other Representations
- Points-to Graph.
- Static single assignment (SSA).

### Tools
- C/C++: LLVM, CIL
- Java: SOOT, Wala
- Binary: Valgrind, Pin