Overview

Static Program Representations

Dynamic Program Representations

Summary

CS 6V81-05: System Security and Malicious Code Analysis
Understanding the Program Representations

Zhiqiang Lin

Department of Computer Science
University of Texas at Dallas

January 25th, 2012
Outline

1. Overview

2. Static Program Representations
   - Abstract Syntax Tree
   - Control Flow Graph
   - Program Dependence Graph
   - Points-to Graph
   - Call Graph

3. Dynamic Program Representations
   - Control Flow Trace, Address and Value Traces
   - Dynamic Dependence Graph (Dynamic Slicing)

4. Summary
1. Overview

2. Static Program Representations
   - Abstract Syntax Tree
   - Control Flow Graph
   - Program Dependence Graph
   - Points-to Graph
   - Call Graph

3. Dynamic Program Representations
   - Control Flow Trace, Address and Value Traces
   - Dynamic Dependence Graph (Dynamic Slicing)

4. Summary
Program needs a representation for the analysis

```c
#include <stdio.h>
int main()
{
    printf("pid=%d\n", getpid());
    return 0;
}
```
Program needs a representation for the analysis

```c
#include <stdio.h>
int main()
{
    printf("pid=%d\n", getpid());
    return 0;
}
```

c17f093d: 55 push %ebp
c17f093e: 89 e5 mov %esp, %ebp
c17f0940: 83 ec 08 sub $0x8, %esp
c17f0943: e8 40 2d 81 ff call c1003688 <mcount>

Programs (including assembly/disassembly) are written in text sequences of characters. Hard to work with (not even for machine). Convert to structured representation.
Program needs a representation for the analysis

```
1 #include <stdio.h>
2 int main()
3 {
4    printf("pid=%d\n", getpid());
5    return 0;
6 }
```

c17f093d: 55 push %ebp
c17f093e: 89 e5 mov %esp,%ebp
c17f0940: 83 ec 08 sub $0x8,%esp
c17f0943: e8 40 2d 81 ff call c1003688 <mcount>

Programs (including assembly/disassembly) are written in text

- Sequences of characters
- Hard to work with (not even for machine)
- Convert to structured representation
.... “For those who keep track of such things, checkers in the
research system typically traverse program paths
(flow-sensitive) in a forward direction, going across function
calls (inter-procedural) while keeping track of call-site-specific
information (context-sensitive) and toward the end of the effort
had some of the support needed to detect when a path was
infeasible (path-sensitive).” ...
For those who keep track of such things, checkers in the research system typically traverse program paths (flow-sensitive) in a forward direction, going across function calls (inter-procedural) while keeping track of call-site-specific information (context-sensitive) and toward the end of the effort had some of the support needed to detect when a path was infeasible (path-sensitive).

Outline

1. Overview

2. Static Program Representations
   - Abstract Syntax Tree
   - Control Flow Graph
   - Program Dependence Graph
   - Points-to Graph
   - Call Graph

3. Dynamic Program Representations
   - Control Flow Trace, Address and Value Traces
   - Dynamic Dependence Graph (Dynamic Slicing)

4. Summary
Outline

1. Overview

2. Static Program Representations
   - Abstract Syntax Tree
   - Control Flow Graph
   - Program Dependence Graph
   - Points-to Graph
   - Call Graph

3. Dynamic Program Representations
   - Control Flow Trace, Address and Value Traces
   - Dynamic Dependence Graph (Dynamic Slicing)

4. Summary
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.
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
```
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
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
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

- Need simpler representation for analysis (at least, for dataflow analysis)
Outline

1 Overview

2 Static Program Representations
   - Abstract Syntax Tree
   - Control Flow Graph
   - Program Dependence Graph
   - Points-to Graph
   - Call Graph

3 Dynamic Program Representations
   - Control Flow Trace, Address and Value Traces
   - Dynamic Dependence Graph (Dynamic Slicing)

4 Summary
Control Flow Graph

A directed graph where

- Each node represents a statement
- Edges represent control flow
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.
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 $E \subseteq V \times V$
- 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)
**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.

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

1 Overview

2 Static Program Representations
   - Abstract Syntax Tree
   - Control Flow Graph
   - Program Dependence Graph
   - Points-to Graph
   - Call Graph

3 Dynamic Program Representations
   - Control Flow Trace, Address and Value Traces
   - Dynamic Dependence Graph (Dynamic Slicing)

4 Summary
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.
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

<table>
<thead>
<tr>
<th>Dominator</th>
</tr>
</thead>
<tbody>
<tr>
<td>A block $M$ <strong>dominates</strong> a block $N$ if every path from the entry that reaches block $N$ has to pass through block $M$.</td>
</tr>
</tbody>
</table>

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., the entry block).
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.
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*.
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)
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 $M$. 
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.
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 $N$. 
Post-Dominator and I-Post-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)
Post-Dominator and I-Post-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
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 $Y$

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

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

CD(5)=3
Outline

1 Overview

2 Static Program Representations
   - Abstract Syntax Tree
   - Control Flow Graph
   - Program Dependence Graph
   - Points-to Graph
   - Call Graph

3 Dynamic Program Representations
   - Control Flow Trace, Address and Value Traces
   - Dynamic Dependence Graph (Dynamic Slicing)

4 Summary
Points-to Graph

Aliases

Two expressions that denote the same memory location.
Points-to Graph

Aliases
Two expressions that denote the same memory location.

Introduced by
- pointers
- call-by-reference
- array indexing
- C unions
Points-to Graph

at a program point, compute a set of pairs of the form p -> x, where p MAY/MUST points to x.

```java
m(p) {
    r=new C();
    p->f = r;
    t = new C();
    if (...) 
        q=p;
    r->f = t;
}
```
Points-to Graph

At a program point, compute a set of pairs of the form \( p \rightarrow x \), where \( p \) MAY/MUST points to \( x \).

```plaintext
m(p) {
    r = new C();
    p->f = r;
    ⇒
    t = new C();
    if (...) 
    q = p;
    r->f = t;
}
```
Points-to Graph

at a program point, compute a set of pairs of the form \( p \rightarrow x \), where \( p \) \textbf{MAY/MUST} points to \( x \).

\[
m(p) \{
    \text{r} = \text{new C}(); \\
    \text{p} \rightarrow f = \text{r}; \\
    \Rightarrow \text{t} = \text{new C}(); \\
    \text{if ( . . . )} \\
    \text{q} = \text{p}; \\
    \text{r} \rightarrow f = \text{t};
\}
\]
Points-to Graph

at a program point, compute a set of pairs of the form \( p \rightarrow x \), where \( p \) MAY/MUST points to \( x \).

```java
m(p) {
    r = new C();
    p->f = r;
    t = new C();
    if (...) ⇒ q = p;
    r->f = t;
}
```
Points-to Graph

at a program point, compute a set of pairs of the form \( p \rightarrow x \), where \( p \) MAY/MUST points to \( x \).

\[
m(p) \{ \\
  r = \text{new } C(); \\
  p->f = r; \\
  t = \text{new } C(); \\
  \text{if } (...) \\
  q = p; \\
  \Rightarrow r->f = t; \\
\} \\
\]

\[
\text{Points-to Graph} \\
\]

\[
p \quad \rightarrow \quad f \quad \rightarrow \quad f \\
\quad \rightarrow \quad r \\
q \quad \rightarrow \quad f \\
\]

\[
t \\
\]
Points-to Graph

at a program point, compute a set of pairs of the form $p \rightarrow x$, where $p$ MAY/MUST points to $x$.

```java
m(p) {
    r = new C();
    p->f = r;
    t = new C();
    if (...) q = p;
    ⇒ r->f = t;
}
```

$p->f->f$ and $t$ are aliases
1. Overview

2. Static Program Representations
   - Abstract Syntax Tree
   - Control Flow Graph
   - Program Dependence Graph
   - Points-to Graph
   - Call Graph

3. Dynamic Program Representations
   - Control Flow Trace, Address and Value Traces
   - Dynamic Dependence Graph (Dynamic Slicing)

4. Summary
Call Graph

- Call graph
  - Nodes are procedures
  - Edges are calls
- Hard cases for building call graph
  - Calls through function pointers
Outline

1. Overview

2. Static Program Representations
   - Abstract Syntax Tree
   - Control Flow Graph
   - Program Dependence Graph
   - Points-to Graph
   - Call Graph

3. Dynamic Program Representations
   - Control Flow Trace, Address and Value Traces
   - Dynamic Dependence Graph (Dynamic Slicing)

4. Summary
Outline

1 Overview

2 Static Program Representations
   - Abstract Syntax Tree
   - Control Flow Graph
   - Program Dependence Graph
   - Points-to Graph
   - Call Graph

3 Dynamic Program Representations
   - Control Flow Trace, Address and Value Traces
   - Dynamic Dependence Graph (Dynamic Slicing)

4 Summary
Control flow trace, address and value traces

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

N=2:
Control flow trace, address and value traces

N=2:

11: Sum=0
Control flow trace, address and value traces

1: sum=0
2: i=1

3: while (i<N) do
   4: i=i+1
   5: sum=sum+i

6: print(sum)

N=2:
- 1₁: Sum=0
- 2₁: i=1
Control flow trace, address and value traces

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

N=2:
- 1₁: Sum=0
- 2₁: i=1
- 3₁: while (i<N) do
Control flow trace, address and value traces

N=2:

1\_1: Sum=0
2\_1: i=1
3\_1: while(i<N)do
4\_1: i=i+1
5\_1: sum=sum+i
6\_1: print(sum)

3\_1: while(i<N)do
4\_1: i=i+1
5\_1: sum=sum+i

...
Control flow trace, address and value traces

N=2:

1. $\text{Sum}=0$
2. $i=1$
3. while ($i<N$) do
4. $i=i+1$
5. $\text{sum}=\text{sum}+i$
6. print (sum)
Control flow trace, address and value traces

N=2:
- 1₁: Sum=0
- 2₁: i=1
- 3₁: while(i<N)do
  - 4₁: i=i+1
  - 5₁: sum=sum+i
- 3₂: while(i<N)do
  - 6: print(sum)
Control flow trace, address and value traces

N=2:
- $1_1$: Sum=0
- $2_1$: i=1
- $3_1$: while(i<N) do
  - $4_1$: i=i+1
  - $5_1$: sum=sum+i
- $3_2$: while(i<N) do
  - $4_2$: i=i+1

Example:
- $1$: sum=0
- $2$: i=1
- $3$: while (i<N) do
  - $4$: i=i+1
  - $5$: sum=sum+i
- $6$: print(sum)
Control flow trace, address and value traces

N=2:

- $1_1$: Sum=0
- $2_1$: i=1
- $3_1$: while (i<N) do
- $4_1$: i=i+1
- $5_1$: sum=sum+i
- $3_2$: while (i<N) do
- $4_2$: i=i+1
- $5_2$: sum=sum+i

1: sum=0
2: i=1
3: while (i<N) do
4: i=i+1
5: sum=sum+i
6: print(sum)
Control flow trace, address and value traces

Overview
Static Program Representations
Dynamic Program Representations
Summary

N=2:

- 1₁: Sum=0
- 2₁: i=1
- 3₁: while(i<N)do
  - 4₁: i=i+1
  - 5₁: sum=sum+i
- 3₂: while(i<N)do
  - 4₂: i=i+1
  - 5₂: sum=sum+i
- 3₃: while(i<N)do

---

1: sum=0
2: i=1
3: while (i<N) do
  4: i=i+1
  5: sum=sum+i
6: print(sum)
Control flow trace, address and value traces

N=2:

1. \text{Sum}=0
2. \text{i}=1
3. \text{while (i} < \text{N)} \text{do}
4. \text{i}=\text{i}+1
5. \text{sum}=\text{sum}+\text{i}
6. \text{print (sum)}

1. \text{i}=1
2. \text{i}=1
3. \text{while (i} < \text{N)} \text{do}
4. \text{i}=\text{i}+1
5. \text{sum}=\text{sum}+\text{i}
6. \text{print (sum)}

\text{N=2:}

- 1\text{\_1: Sum}=0
- 2\text{\_1: i}=1
- 3\text{\_1: while (i} < \text{N)} \text{do}
- 4\text{\_1: i}=\text{i}+1
- 5\text{\_1: sum}=\text{sum}+\text{i}
- 3\text{\_2: while (i} < \text{N)} \text{do}
- 4\text{\_2: i}=\text{i}+1
- 5\text{\_2: sum}=\text{sum}+\text{i}
- 3\text{\_3: while (i} < \text{N)} \text{do}
- 6\text{\_1: print (sum)}
Control flow trace, address and value traces

N=2:

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)

- <...xi,...> x is a program point, xi is an execution point
- <...8048057_{27}, 804805a_{29}, ...>
Outline

1. Overview

2. Static Program Representations
   - Abstract Syntax Tree
   - Control Flow Graph
   - Program Dependence Graph
   - Points-to Graph
   - Call Graph

3. Dynamic Program Representations
   - Control Flow Trace, Address and Value Traces
   - Dynamic Dependence Graph (Dynamic Slicing)

4. Summary
What is a slice?

S.. . . = f (v)

Slice of v at S is the set of statements involved in computing v’s value at S.

*MarkWeiser*, 1982

- Data dependence
- Control dependence
What is a slice?

void main () {
    int I=0;
    int sum=0;
    while (I<N) {
        sum=add(sum,I);
        I=add(I,1);
    }
    print(sum);
    print(I);
}

S::... = f (v)

Slice of v at S is the set of statements involved in computing v’s value at S.

MarkWeiser, 1982

- Data dependence
- Control dependence
What is a slice?

```c
void main () {
    int I=0;
    int sum=0;
    while (I<N) {
        sum=add(sum,I);
        I=add(I,1);
    }
    print(sum);
    print(I);
}
```

S:. . . = f (v)

**Slice** of v at S is the set of statements involved in computing v’s value at S.

*Mark Weiser*, 1982

- Data dependence
- Control dependence
What is a slice?

```c
void main () {
    int I=0;
    int sum=0;
    while (I<N) {
        sum=add(sum,I);
        I=add(I,1);
    }
    print(sum);
    print(I);
}
```

Slice of $v$ at $S$ is the set of statements involved in computing $v$’s value at $S$.

*MarkWeiser*, 1982

- Data dependence
- Control dependence
What is a slice?

```c
void main () {
    int l=0;
    int sum=0;
    while (l<N) {
        sum=add(sum,l);
        l=add(l,1);
    }
    print(sum);
    print(l);
}
```

Sliced code:

```c
S:: . . = f (v)
```

**Slice** of \( v \) at \( \text{S} \) is the set of statements involved in computing \( v \)'s value at \( \text{S} \).

*Mark Weiser, 1982*

- Data dependence
- Control dependence
What is a slice?

void main () {
    int I=0;
    int sum=0;
    while (I<N) {
        sum=add(sum,I);
        I=add(I,1);
    }
    print(sum);
    print(I);
}

S: . . . = f (v)

Slice of v at S is the set of statements involved in computing v’s value at S.

Mark Weiser, 1982

- Data dependence
- Control dependence
Why is a static slice imprecise?

All possible program paths

\[ S1: x = \ldots \quad \text{S2: } x = \ldots \]

\[ \text{L1: } \ldots = x \]

Use of Pointers - static alias analysis is very imprecise

\[ S1: a = \ldots \quad \text{S2: } b = \ldots \]

\[ \text{L1: } \ldots = *p \]

Use of function pointers - hard to know which function is called, conservative expectation results in imprecision
Dynamic Slicing

- Korel and Laski, 1988
- Dynamic slicing makes use of all information about a particular execution of a program and computes the slice based on an execution history (trace)
  - Trace consists control flow trace and memory reference trace
- A dynamic slice query is a triple
- Smaller, more precise, more helpful to the user
Dynamic Slicing Example

For input N=2

1: b=0
2: a=2
3: for i = 1 to N do
4:   if ((i++)%2==1) then
5:     a = a+1
6:   else
7:     b = a*2
8:   endif
9: done
10: z = a+b
11: print(z)
Dynamic Slicing Example

1: b=0
2: a=2
3: for i= 1 to N do
4: if ((i++)%2==1) then
5:   a = a+1
else
6:   b = a*2
endif
done
7: z = a+b
8: print(z)

For input N=2

1:1 b=0
   [b=0]
Dynamic Slicing Example

1: b=0
2: a=2
3: for i= 1 to N do
4:  if ((i++)%2==1) then
5:    a = a+1
   else
6:    b = a*2
   endif
  done
7: z = a+b
8: print(z)

For input N=2
- 1₁:b=0
- 2₁:a=2

[b=0]
Dynamic Slicing Example

For input N=2
- 1\_1 : b=0       [b=0]
- 2\_1 : a=2
- 3\_1 : for i = 1 to N do       [i=1]

```plaintext
1: b=0
2: a=2
3: for i= 1 to N do
4: if ((i++)%2==1) then
5:     a = a+1
    else
6:     b = a*2
    endif
    done
7: z = a+b
8: print(z)
```
Dynamic Slicing Example

1: b = 0
2: a = 2
3: for i = 1 to N do
4: if ((i++)%2 == 1) then
5:   a = a + 1
   else
6:     b = a * 2
endif
done
7: z = a + b
8: print(z)

For input N=2

1₁:b=0          [b=0]
2₁:a=2
3₁:for i = 1 to N do          [i=1]
4₁:if((i++)%2 == 1)then [i=1]
Dynamic Slicing Example

1: b=0
2: a=2
3: for i= 1 to N do
4: if ((i++)%2==1) then
5: a = a+1
   else
6: b = a * 2
   endif
7: done
8: z = a+b
9: print(z)

For input N=2
- 1_1:b=0 [b=0]
- 2_1:a=2
- 3_1:for i = 1 to N do [i=1]
- 4_1:if((i++)%2 == 1)then [i=1]
- 5_1:a=a+1 [a=3]
- 6_1:b=a*2 [b=6]
- 7_1:z=a+b [z=9]
- 8_1:print(z) [z=9]
Dynamic Slicing Example

For input N=2

1: b=0
2: a=2
3: for i= 1 to N do
4:   if ((i++)%2==1) then
5:     a = a+1
6:     b = a*2
    else
6:   endif
7: done
8: z = a+b
9: print(z)

- 1: b=0
- 2: a=2
- 3: for i=1 to N do
- 4:   if(i%2 == 1) then
- 5:     a = a+1
- 6:     b = a*2
- 7:   endif
- 8: done
- 9: z = a+b
- 10: print(z)
Dynamic Slicing Example

1: b=0
2: a=2
3: for i= 1 to N do
4: if ((i++)%2==1) then
5: a = a+1
   else
6: b = a*2
   endif
done
7: z = a+b
8: print(z)

For input N=2

- 1₁: b=0 [b=0]
- 2₁: a=2
- 3₁: for i = 1 to N do [i=1]
- 4₁: if((i++)%2 == 1)then [i=1]
- 5₁: a=a+1 [a=3]
- 3₂: for i=1 to N do [i=2]
- 4₂: if(i%2 == 1)then [i=2]
Dynamic Slicing Example

1: b=0
2: a=2
3: for i= 1 to N do
4:   if ((i++)%2==1) then
5:     a = a+1
else
6:     b = a*2
endif
done
7: z = a+b
8: print(z)

For input N=2
- 1₁:b=0          [b=0]
- 2₁:a=2
- 3₁:for i = 1 to N do    [i=1]
- 4₁:if((i++)%2 == 1)then [i=1]
- 5₁:a=a+1         [a=3]
- 3₂:for i=1 to N do   [i=2]
- 4₂:if(i%2 == 1)then  [i=2]
- 6₁:b=a*2         [b=6]
Dynamic Slicing Example

1: \( b=0 \)
2: \( a=2 \)
3: \( \text{for } i=1 \text{ to } N \text{ do} \)
4: \( \text{if } ((i++)\%2==1) \text{ then} \)
5: \( \quad a = a+1 \)
else
6: \( \quad b = a*2 \)
endif
done
7: \( z = a+b \)
8: print(z)

For input \( N=2 \)

- \( 1_1:b=0 \) \[b=0\]
- \( 2_1:a=2 \)
- \( 3_1:\text{for } i=1 \text{ to } N \text{ do} \) \[i=1\]
  - \( 4_1:\text{if}((i++)\%2 == 1)\text{then} \) \[i=1\]
  - \( 5_1:a=a+1 \) \[a=3\]
  - \( 3_2:\text{for } i=1 \text{ to } N \text{ do} \) \[i=2\]
    - \( 4_2:\text{if}(i\%2 == 1)\text{then} \) \[i=2\]
    - \( 6_1:b=a*2 \) \[b=6\]
  - \( 7_1:z=a+b \) \[z=9\]
Dynamic Slicing Example

```
1: b=0
2: a=2
3: for i= 1 to N do
4:   if ((i++)%2==1) then
5:     a = a+1
   else
6:     b = a*2
   endif
done
7: z = a+b
8: print(z)
```

For input N=2

- 1₁ : b=0 [b=0]
- 2₁ : a=2
- 3₁ : for i = 1 to N do [i=1]
- 4₁ : if((i++)%2 == 1)then [i=1]
- 5₁ : a=a+1 [a=3]
- 3₂ : for i=1 to N do [i=2]
- 4₂ : if(i%2 == 1)then [i=2]
- 6₁ : b=a*2 [b=6]
- 7₁ : z=a+b [z=9]
- 8₁ : print(z) [z=9]
Dynamic Slicing Example

1: b = 0
2: a = 2
3: for i = 1 to N do
4: if ((i++)%2 == 1) then
5:     a = a + 1
6: else
7:     b = a * 2
8: endif
9: done
10: z = a + b
11: print(z)

For input N=2

- 1_1: b = 0 [b=0]
- 2_1: a = 2
- 3_1: for i = 1 to N do [i=1]
- 4_1: if ((i++)%2 == 1) then [i=1]
- 5_1: a = a + 1 [a=3]
- 6_1: for i = 1 to N do [i=2]
- 7_1: if (i%2 == 1) then [i=2]
- 8_1: b = a * 2 [b=6]
- 9_1: z = a + b [z=9]
- 10_1: print(z) [z=9]
Dynamic Slicing Example

1:  b=0
2:  a=2
3:  for i= 1 to N do
4:    if ((i++)%2==1) then
5:      a = a+1
else
6:      b = a*2
endif
done
7:  z = a+b
8:  print(z)

For input N=2

1_1:b=0        [b=0]
2_1:a=2
3_1:for i = 1 to N do   [i=1]
4_1:if((i++)%2 == 1)then [i=1]
5_1:a=a+1        [a=3]
3_2:for i=1 to N do   [i=2]
4_2:if(i%2 == 1)then  [i=2]
6_1:b=a*2        [b=6]
7_1:z=a+b        [z=9]
8_1:print(z)     [z=9]
Dynamic Slicing Example

1: b=0
2: a=2
3: for i= 1 to N do
4: if ((i++)%2==1) then
5:     a = a+1
  else
6:     b = a*2
endif
done
7: z = a+b
8: print(z)

For input N=2

1.\textsuperscript{1} : b=0  \quad \textcolor{green}{[b=0]}
2.\textsuperscript{1} : a=2
3.\textsuperscript{1} : for i = 1 to N do  \quad \textcolor{green}{[i=1]}
4.\textsuperscript{1} : if((i++)%2 == 1)then  \quad \textcolor{green}{[i=1]}
5.\textsuperscript{1} : a=a+1  \quad \textcolor{green}{[a=3]}
6.\textsuperscript{1} : for i=1 to N do  \quad \textcolor{green}{[i=2]}
7.\textsuperscript{1} : if(i%2 == 1)then  \quad \textcolor{green}{[i=2]}
6.\textsuperscript{1} : b=a*2  \quad \textcolor{green}{[b=6]}
7.\textsuperscript{1} : z=a+b  \quad \textcolor{green}{[z=9]}
8.\textsuperscript{1} : print(z)  \quad \textcolor{green}{[z=9]}
Dynamic Slicing Example

1: b=0
2: a=2
3: for i = 1 to N do
4: if ((i++)%2==1) then
5: a = a+1
else
6: b = a*2
endif
done
7: z = a+b
8: print(z)

For input N=2

1₁: b=0 [b=0]
2₁: a=2
3₁: for i = 1 to N do [i=1]
4₁: if((i++)%2 == 1)then [i=1]
5₁: a=a+1 [a=3]
3₂: for i=1 to N do [i=2]
4₂: if(i%2 == 1)then [i=2]
6₁: b=a*2 [b=6]
7₁: z=a+b [z=9]
8₁: print(z) [z=9]
Dynamic Slicing Example

1: b=0
2: a=2
3: for i = 1 to N do
4: if ((i++)%2==1) then
5:   a = a+1
   else
6:   b = a*2
endif
done
7: z = a+b
8: print(z)

For input N=2

- 1₁: b=0       [b=0]
- 2₁: a=2
- 3₁: for i = 1 to N do [i=1]
- 4₁: if((i++)%2 == 1)then [i=1]
- 5₁: a=a+1       [a=3]
- 3₂: for i=1 to N do [i=2]
- 4₂: if(i%2 == 1)then [i=2]
- 6₁: b=a*2       [b=6]
- 7₁: z=a+b       [z=9]
- 8₁: print(z)    [z=9]
Issues about Dynamic Slicing

- Precision - perfect
- Running history - very big (GB)
- Algorithm to compute dynamic slice - slow and very high space requirement.
Comments

Want to know more?
- Frank Tip’s survey paper (1995)

Static slicing is very useful for static analysis
- Code transformation, program understanding, etc.
- Points-to analysis is the key challenge

Dynamic slicing
- Precise
  - Good for vulnerability identification.
- Solution space is much larger.
- There exist hybrid techniques.
How are dynamic slices computed?

- **Execution traces**
  - control flow trace – dynamic control dependences
  - memory reference trace – dynamic data dependences

- Construct a **dynamic dependence graph**
- Traverse **dynamic dependence graph** to compute slices
Outline

1 Overview

2 Static Program Representations
   - Abstract Syntax Tree
   - Control Flow Graph
   - Program Dependence Graph
   - Points-to Graph
   - Call Graph

3 Dynamic Program Representations
   - Control Flow Trace, Address and Value Traces
   - Dynamic Dependence Graph (Dynamic Slicing)

4 Summary
Summary

Static Representation
- AST
- Control Flow Graph
- Program Dependency Graph (static slicing)
- Points-to Graph
- Call Graph

Dynamic Representation
- Traces (control flow, value, address)
- Dynamic Dependency (dynamic slicing)
References

5. A brief survey of program slicing http://dl.acm.org/citation.cfm?id=1050865