Advanced Topics in Malware Analysis

Program Slicing

Brendan Saltaformaggio, PhD
Assistant Professor
School of Electrical and Computer Engineering

Slicing: Program Dependent Graphs

Georgia Tech

1

Learning Objectives

- Locate the set of statements that compose a program slice
- Construct a program dependent graph
- Assemble dynamic slices using a program dependency graph

Georgia Tech

2

What is Slicing?

- A **Slice** of variable **v** at statement **s** is the set of statements involved in computing the value of v at s. [Mark Weiser, 1982]

Control Dep.
Data Dep.

Slice(i @ 5) =

{1 (or START), 3, 4, 5}

- A slice is a **backwards traversal** of the Program Dependence Graph!

```
1. void sumUp(int n) {
2.     int sum = 0;
3.     int i = 1;
4.     while ( i < n ) {
5.         i = i + 1;
6.         sum = sum + i;
7.     }
8.     printf("%d", sum);
9. }
```

Georgia Tech

3

Remember the Program Dependence Graph?

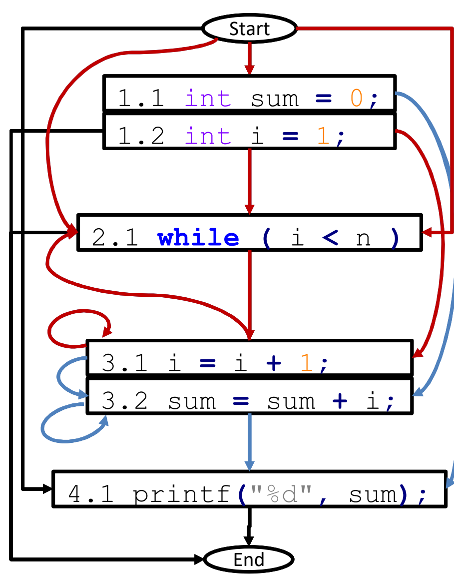
- A program dependence graph **PDG = (N, Ed, Ec)**
  - A finite set **N** of nodes which represent statements, possibly within basic blocks "**super-nodes**"
  - A finite set **Ed** of edges (**i, j**) representing that node $\mathbf{n_j}$ is data dependent on node n_i
  - A finite set **Ec** of edges (**i, j**) representing that (super-)node $\mathbf{n_j}$ is control dependent on node $\mathbf{n_i}$

Slice(i @ 3.1) =

{START, 1.2, 2.1, 3.1}

Slice

```
void sumUp(int n) {
    int sum = 0;
    int i = 1;
    while ( i < n ) {
        i = i + 1;
        sum = sum + i;
    }
    printf("%d", sum);
}
```



Control Dep.
Data Dep.

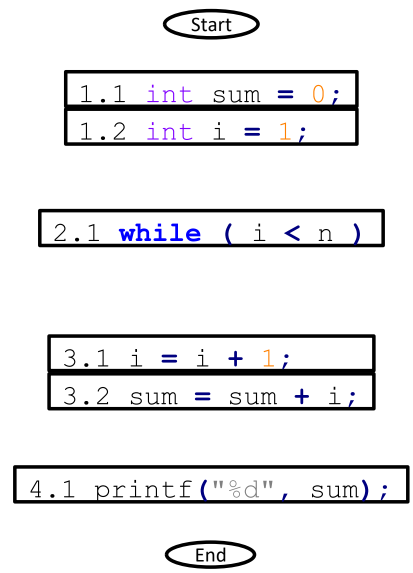
Georgia Tech

4

How To Compute A Slice Statically

- Build a **PDG = (N, Ed, Ec)**
  - A finite set **N** of nodes which represent statements, possibly within basic blocks "**super-nodes**"

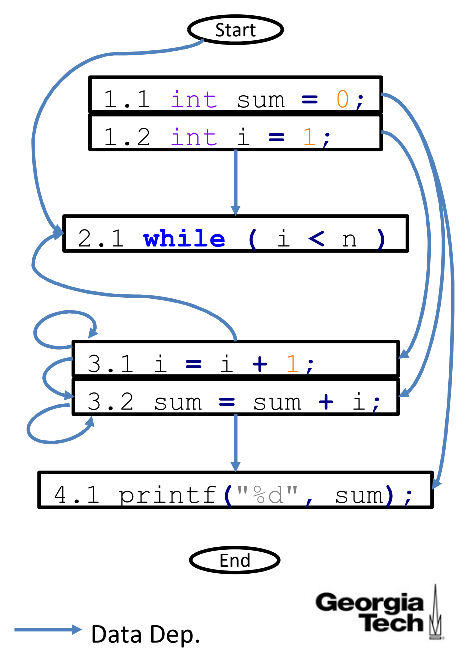
```
void sumUp(int n) {
    int sum = 0;
    int i = 1;
    while ( i < n ) {
        i = i + 1;
        sum = sum + i;
    }
    printf("%d", sum);
}
```



5

How To Compute A Slice Statically

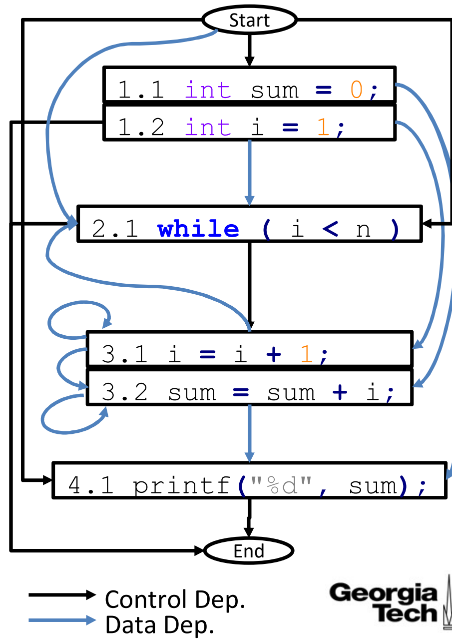
- Build a **PDG = (N, Ed, Ec)**
  - A finite set N of nodes which represent statements, possibly within basic blocks "**super-nodes**"
  - A finite set **Ed** of edges (**i, j**) representing that node n_j is data dependent on node n_i
  - **Recall**: **X** is data dependent on **Y** iff
    1) There exists a variable v that is defined at Y and used at X
    2) There exists a path of nonzero length from Y to X along which v is not re-defined



6

How To Compute A Slice Statically

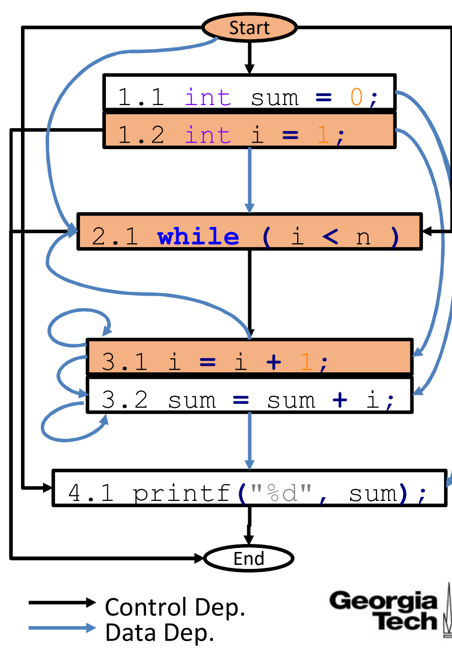
- Build a PDG = (N, Ed, Ec)
  - A finite set N of nodes which represent statements, possibly within basic blocks "super-nodes"
  - A finite set Ed of edges (i, j) representing that node n_j is data dependent on node n_i
  - A finite set Ec of edges (i, j) representing that (super-)node n_j is control dependent on node n_i
  - Recall: Y is control-dependent on X iff X directly determines whether Y executes:
    1) X is not strictly post-dominated by Y
    2) 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



7

How To Compute A Slice Statically

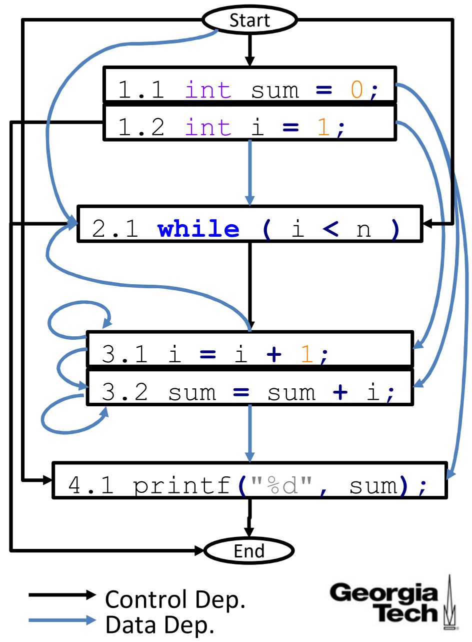
- Build a PDG = (N, Ed, Ec)
  - A finite set N of nodes which represents statements, possibly within basic blocks "super-nodes"
  - A finite set Ed of edges (i, j) representing that node n_j is data dependent on node n_i
  - A finite set Ec of edges (i, j) representing that (super-)node n_j is control dependent on node n_i
- Given a slice criterion, i.e., the starting point, a slice is computed by traversing the set of **backward-reachable** nodes in the program dependence graph
- Slice (I @ 3.1) = {START, 1.2, 2.1, 3.1}



8

How To Compute A Slice Statically

- Build a PDG = (N, Ed, Ec)
  - A finite set N of nodes which represents statements, possibly within basic blocks "super-nodes"
  - A finite set Ed of edges (i, j) representing that node n_j is data dependent on node n_i
  - A finite set Ec of edges (i, j) representing that (super-)node n_j is control dependent on node n_i
- Given a slice criterion, i.e., the starting point, a slice is computed by traversing the set of **backward-reachable** nodes in the program dependence graph
- Slice (sum @ 4.1) = ?

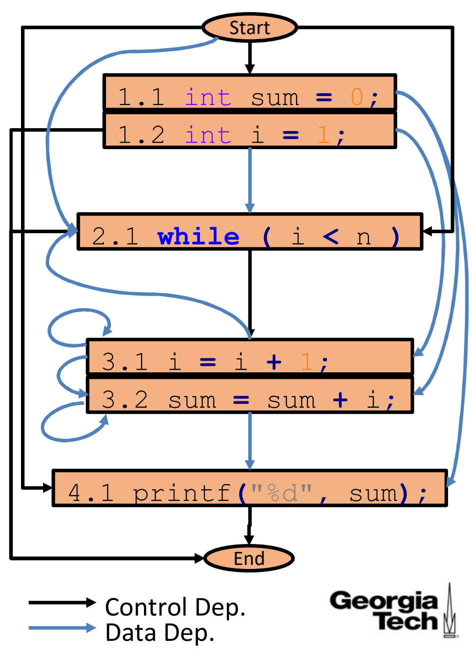


Georgia Tech

9

How To Compute A Slice Statically

- Build a PDG = (N, Ed, Ec)
  - A finite set N of nodes which represents statements, possibly within basic blocks "super-nodes"
  - A finite set Ed of edges (i, j) representing that node n_j is data dependent on node n_i
  - A finite set Ec of edges (i, j) representing that (super-)node n_j is control dependent on node n_i
- Given a slice criterion, i.e., the starting point, a slice is computed by traversing the set of **backward-reachable** nodes in the program dependence graph
- Slice (sum @ 4.1) = {START, 1.1, 1.2, 2.1, 3.1, 3.2, 4.1}



Georgia Tech

10

But I Thought We Were Done With Static Analysis??

- We are!
- Static slices are extremely imprecise
    1. Don't have dynamic control flow information
    2. Static alias analysis is very difficult (as you know)
- This makes the underlying Program Dependence Graph very imprecise
- So slicing is generally only performed dynamically

```
if (P)
  x=f(…);
else
  x=g(…);
…=x;
```

```
int x, y, z;
int * p;
x = 5;
y = 10;
z = 8;
p = &x;
p = p + z;
z = *p;
```

Georgia Tech

11

Advanced Topics in Malware Analysis

Program Slicing

Brendan Saltaformaggio, PhD

Assistant Professor

School of Electrical and Computer Engineering

Dynamic Slicing

Georgia Tech

12

Static VS. Dynamic slicing Example

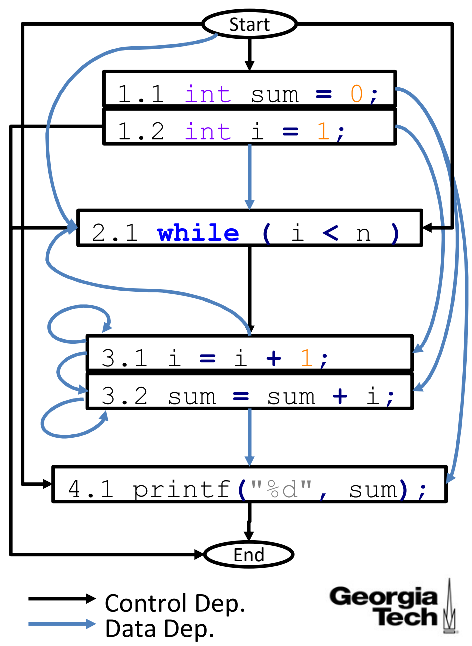
StaticSlice(sum @ 4.1) =
{START, 1.1, 1.2, 2.1, 3.1, 3.2, 4.1}

Execution Trace (n = 0)

```
1.1 int sum = 0;
1.2 int i = 1;
2.1 while ( i < n )
4.1 printf("%d", sum);
```

DynamicSlice(sum @ 4.1) = {1.1, 4.1}

- Much better right?
- Dynamic analysis concretizes many assumptions that restrict static analysis!



13

Dynamic Slicing

- Korel and Laski, 1988
- Dynamic slicing makes use of all information about a particular execution of a program
- Dynamic slices are computed by constructing a Dynamic Program Dependence Graph
  - Each node is an **executed statement** (instruction)
  - An edge is **present** between two nodes **if there exists a data or control dependence**
  - A dynamic slice **criterion** is a **triple <Variable, Execution Point, Input>**
- The set of statements reachable in the DPDG from a criterion constitute the slice
- Dynamic slices are smaller, more precise, more helpful to the user (or malware analyst)

14

Computing Dynamic Slices

- Data dependence
  - Do we still care about aliasing? No! ☺
  - A backward linear scan over the trace is able to recover **all** data dependences
  - We now only need to compute a dynamic Def/Use chain!
- Control Dependence
  - Recall that we need to find the predicate instance that a statement is control dependent on
  - Can we simply traverse backwards and find the closest predicate?
  - No! ☹
  - Recall our execution trace from before:
    - Is 4.1 control dependent on 2.1? No!
  - We need to define the notion of dynamic control dependence!

Execution Trace (n = 0)

```
1.1 int sum = 0;
1.2 int i = 1;
2.1 while ( i < n )
4.1 printf("%d", sum);
```

15

Advanced Topics in Malware Analysis

Program Slicing

Brendan Saltaformaggio, PhD

Assistant Professor

School of Electrical and Computer Engineering

Dynamic Data Dependence

16

Dynamic Data Dependence

- Instrument the executable to log the Def/Use of each instruction
- At runtime, all memory locations will be resolved

```
.text:00 pow               proc near
.text:00
.text:00 var_ReturnVal     = dword ptr -1Ch
.text:00 var_Y             = dword ptr -18h
.text:00 var_X             = dword ptr -14h
.text:00 var_Z             = dword ptr -8
.text:00 var_Power         = dword ptr -4
.text:00
.text:00 push  rbp                     printf(“00 D: rsp %d, U: rsp rbp”, rsp-4)
.text:01 mov   rbp, rsp                printf(“01 D: rbp, U: rsp”)
.text:04 mov   [rbp+var_X], edi        printf(“04 D: %d, U: rbp edi”, rbp+var_X)
.text:07 mov   [rbp+var_Y], esi        printf(“07 D: %d, U: rbp esi”, rbp+var_Y)
.text:0A cmp   [rbp+var_Y], 0          printf(“0A D: rflags, U: rbp %d”, rbp+var_Y)
.text:0E jns   short loc_1A            printf(“0E D: U: rflags”)
.text:10 mov   eax, [rbp+var_Y]        printf(“10 D: eax, U: rbp %d”, rbp+var_Y)
.text:13 neg   eax                     printf(“13 D: eax, U: eax”)
.text:15 mov   [rbp+var_Power], eax    printf(“15 D: %d U: rbp eax”, rbp+var_Power)
.text:18 jmp   short loc_20            printf(“18 D: U:”)
```

Georgia Tech

17

Dynamic Data Dependence

- A backward linear scan over the trace is able to recover **all** data dependences

Trace Output (assume: RSP=0x460020)

00 D: rsp 0x46001c, U: rsp rbp

01 D: rbp, U: rsp

04 D: 0x460008, U: rbp edi

07 D: 0x460004, U: rbp esi

0A D: rflags, U: rbp 0x460004

0E D: U: rflags

10 D: eax, U: rbp 0x460004

13 D: eax, U: eax

15 D: 0x460018 U: rbp eax

18 D: U:

```
.text:00 pow               proc near
.text:00
.text:00 var_ReturnVal = dword ptr -1Ch
.text:00 var_Y         = dword ptr -18h
.text:00 var_X         = dword ptr -14h
.text:00 var_Z         = dword ptr -8
.text:00 var_Power     = dword ptr -4
.text:00
.text:00 push     rbp
.text:01 mov      rbp, rsp
.text:04 mov      [rbp+var_X], edi
.text:07 mov      [rbp+var_Y], esi
.text:0A cmp      [rbp+var_Y], 0
.text:0E jns      short loc_1A
.text:10 mov      eax, [rbp+var_Y]
.text:13 neg      eax
.text:15 mov      [rbp+var_Power], eax
.text:18 jmp      short loc_20
```

DD(01) = ?

DD(01) = {00}

DD(07) = ?

DD(07) = {START, 01}

DD(10) = ?

DD(10) = {01, 07}

Georgia Tech

18

Dynamic Control Dependence (DCD)

- Computing control dependence from only a dynamic trace is challenging
- Linear scan will not work
  - This is because an execution trace has no notion of the "curly bracket" i.e., **{ }**

Execution Trace (n = 0)

```
1.1 int sum = 0;
1.2 int i = 1;
2.1 while ( i < n )
4.1 printf("%d", sum);
```

VS.

Execution Trace (n = 0)

```
1.1 int sum = 0;
1.2 int i = 1;
2.1 while ( i < n ) {
    }
4.1 printf("%d", sum);
```

- Can we just fall back to our previous static control dependence??
  - Maybe …

19

DCD - Offline TRACE Analysis

1. Assume there are no recursive functions
2. Assume we have already built a dictionary **CD(i) = {$\mathbf{cdi_1}$, $\mathbf{cdi_2}$, … $\mathbf{cdi_n}$}** which maps any instruction i to the set of **static control dependencies** for **i**, say **$\mathbf{cdi_1}$**, **$\mathbf{cdi_2}$, … $\mathbf{cdi_n}$**
3. Collect a dynamic control flow trace
4. To find the control dependence of any instruction **j** in that trace: Traverse backward starting from **j** & find the closest **x, s.t. x** is in **CD(j)**
5. **j** is therefore dynamically control dependent on **x** *(Cont'd)*

20

DCD - Offline TRACE Analysis (Cont'd)

5. **j** is therefore dynamically control dependent on **x**
   - VERY problematic in the presence of recursion
   - Notice that each instruction in the trace can only be dynamically control dependent on a single instruction!
     - Recall that one instruction could be statically control dependent on two!
     - This is because within a concrete trace you can only arrive at that instruction via a single path!

21

Are We Done Yet?

- No
- The previously mentioned algorithms are essentially offline graph construction
  - This implies offline traversals of (extremely) long data reference and control flow traces
- Can we do better?
- Yes
- Efficient online algorithms:
  - Online data dependence detection
  - Online control dependence detection
- The output of our dynamic dependence detection tool will only be the dependencies of each instruction and not a full trace

22

Advanced Topics in Malware Analysis
Program Slicing

Brendan Saltaformaggio, PhD
Assistant Professor
School of Electrical and Computer Engineering

Dynamic Data Dependence Detection

Georgia Tech

23

Efficient Dynamic Data Dependence Detection

- Full Shadow Registers & Memory
- **Basic idea:** Store metadata in a structure which models every real data location
- This enables **online def/use tracking**!
- For each data location that an instruction can define, keep a "**shadow**" **copy** to mark the **last instruction** which defined it
- Implemented **via a dictionary** (or similar data structure) which maps data locations (i.e., registers or memory) to the last instruction which defined them

```
.text:00 push    rbp printf("00 D: rsp %d, U: rsp rbp", rsp-4)
```

```
.text:00 push    rbp   printf("00 DD: %s, %s", shadow_regs[rsp], shadow_regs[rbp]);
                       shadow_regs[rsp]=".text:00"; shadow_mem[&[rsp]]=".text:00";
```

Georgia Tech

24

Efficient Dynamic Data Dependence Detection

- Update shadows before or after instruction executes?
  - E.g., **`shadow_mem[&[rsp]]="…"`** vs. **`shadow_mem[&[rsp]-4]="…"`**
- Also, granularity considerations:
  - Shadow each flag? Shadow memory in 1,4,8 bytes?

25

Efficient Dynamic Data Dependence Detection

```
.text:00 var_Y          = dword ptr -18h
.text:00 var_X          = dword ptr -14h
.text:00 var_Z          = dword ptr -8
.text:00 var_Power      = dword ptr -4
.text:00
.text:00 push    rbp    printf("00 DD: %s, %s", shadow_regs[rsp], shadow_regs[rbp]);
                        shadow_regs[rsp]=".text:00"; shadow_mem[&[rsp]]=".text:00";
.text:01 mov     rbp, rsp    printf("01 DD: %s", shadow_regs[rsp]);
                             shadow_regs[rbp]=".text:01";
.text:04 mov     [rbp+var_X], edi
                printf("04 DD: %s, %s", shadow_regs[rbp], shadow_regs[edi]);
                shadow_mem[&[rbp+var_X]]=".text:04";
.text:07 mov     [rbp+var_Y], esi
                printf("07 DD: %s, %s", shadow_regs[rbp], shadow_regs[esi]);
                shadow_mem[&[rbp+var_Y]]=".text:07";
.text:0A cmp     [rbp+var_Y], 0
                printf("0A DD: %s, %s", shadow_regs[rbp], shadow_mem[&[rbp+var_Y]]);
                shadow_regs[rflags]=".text:0A";
.text:0E jns     short loc_1A      printf("0E DD: %s", shadow_regs[rflags]);
.text:10 mov     eax, [rbp+var_Y]
                printf("10 DD: %s, %s", shadow_regs[rbp], shadow_mem[&[rbp+var_Y]]);
                shadow_regs[eax]=".text:10";
.text:13 neg     eax         printf("13 DD: %s", shadow_regs[eax]);
                             shadow_regs[eax]=".text:13";
.text:15 mov     [rbp+var_Power], eax
                printf("15 DD: %s, %s", shadow_regs[rbp], shadow_regs[eax]);
                shadow_mem[&[rbp+var_Power]]=".text:15";
.text:18 jmp     short loc_20
```

26

Efficient Dynamic Data Dependence Detection

- Beware of space constraints!
  - You cannot pre-allocate **4 bytes** of metadata for every byte of **RAM**!
- Implementation tradeoffs are everywhere!
  - Time vs space, packed vs unpacked data, …
- Read:
  - Nethercote, N. & Seward, J. (2007). How to shadow every byte of memory used by a program. *Proceedings of the 3rd International Conference on Virtual Execution Environments*. (yes, the valgrind author).
- Shadow data is also used to implement taint tracking
  - **Example**: Mark the initial input as "tainted" (1 bit shadow metadata) & taint everywhere the data flows during execution
  - **Application**: Can this program leak sensitive data?

27

Efficient Dynamic Control Dependence Detection

- We can perform a similar online tracking of dynamic control dependence
- Recall our definition of control dependence:
- Y is control-dependent on X iff
  1. X is not strictly post-dominated by Y
  2. 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
- We can watch for these conditions at runtime & determine control dependence from
  the final execution trace
  - **Benefits**: 1) **May** not require ahead-of-time static control dependence analysis
    2) Does not require logging the huge dynamic control flow trace
  - **Problems**: Not as straightforward as the offline approach

28

Advanced Topics in Malware Analysis
Program Slicing

Brendan Saltaformaggio, PhD
Assistant Professor
School of Electrical and Computer Engineering

Regions

29

Dynamic "Regions"

- In order to monitor the execution for our control dependence conditions, we need to introduce the concept of an execution "**region**"
- **Region**: The set of executed statements between a predicate instance and its immediate post-dominator
- We can then say that each statement instance $\mathbf{x}_i$ is dynamically control dependent on the predicate instance leading $\mathbf{x}_i$'s nearest enclosing region
- Thanks to our definition of immediate post-dominated, regions are either nested or disjoint, but can never overlap!
- Read:
  - Xin, B. & Zhang, X. (2007). Efficient online detection of dynamic control dependence. *Proceedings of the 2007 International Symposium on Software Testing and Analysis*.

30

Region Examples

- **Region**: The set of executed statements between a **predicate instance** and its immediate **post-dominator**
- We can then say that each statement instance $\mathbf{x_i}$ is dynamically **control dependent** on the predicate instance leading $\mathbf{x_i}$**'s nearest enclosing region**
- Regions are either **nested** or **disjoint**, but can **never overlap**!

Execution Trace

```
1_1. for(i=0; i<N, i++)
2_1. if(i%2 == 0)
3_1. p = &a[i];
4_1. foo(p);
…
1_2. for(i=0; i<N, i++)
2_2. if(i%2 == 0)
4_2. foo(p);
…
1_3. for(i=0; i<N, i++)
6_1. a = a+1;
```

Program

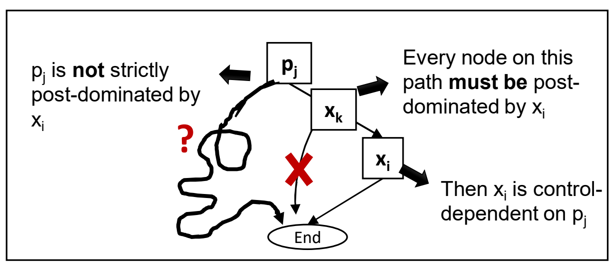
```
1. for(i=0; i<N, i++) {
2.     if(i%2 == 0)
3.         p = &a[i];
4.     foo(p);
5. }
6. a = a+1;
```

Georgia Tech

31

Regions: Property One

Each statement instance x_i is dynamically control dependent on the predicate instance leading x_i's nearest enclosing region



- **Proof**: Let the predicate instance be p_j and assume x_i is not control dependent on p_j
- Therefore, either:

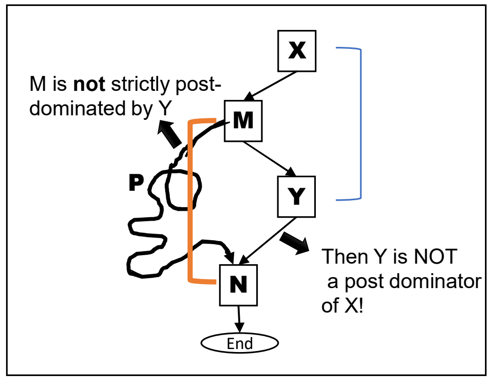
1. No path exists from p_j to exit which that does not pass through x_i. This would indicate that x_i is a post-dominator of p_j, contradicting the condition that x_i is in the region delimited by p_j and its immediate post-dominator; OR
2. There is a node x_k in between p_j and x_i such that x_k has a path to exit that does not pass x_i. Since p_j's immediate post-dominator is also a post-dominator of x_k, x_k and p_j's post-dominator form a smaller region that includes x_i, contradicting that pj leads the enclosing region of xi

Georgia Tech

32

Regions: Property Two

- Regions are either nested or disjoint, but can never overlap!
- Proof: Assume there are two regions (x, y) and (m, n) that overlap
- Let m reside in (x, y). Thus, y resides in (m, n)
- This implies there is a path P from m to exit without passing y
- Therefore, the path from x to m and P constitute a path from x to exit without passing y, contradicting the condition that y is a post-dominator of x



Georgia Tech

33

Detecting Regions At Runtime

- **Observation**: During execution, Regions follow the LIFO pattern
  - Otherwise, some regions must overlap
- **Implication**: The current sequence of nested regions for the current execution point can be maintained by a stack, called a control dependence stack (CDS)
- Each region is nested within the region immediately preceding it in the stack
- The enclosing region for the current execution point is always the top entry in the stack
  - Therefore any execution point is control dependent on the predicate that leads the top entry
- An entry is pushed onto CDS if a branching instruction (predicated jump) is executed
- The current entry is popped if the immediate post-dominator of the branching point is executed, denoting the end of the current region

Georgia Tech

34

Advanced Topics in Malware Analysis
Program Slicing

Brendan Saltaformaggio, PhD
Assistant Professor
School of Electrical and Computer Engineering

Algorithms

Georgia Tech

35

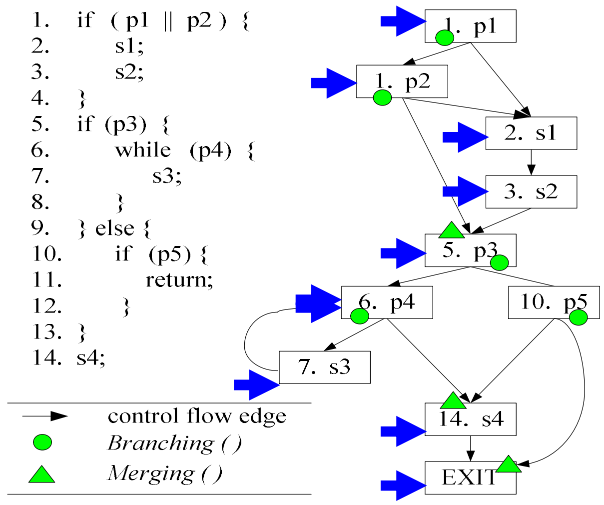
“Idealistic” Algorithm

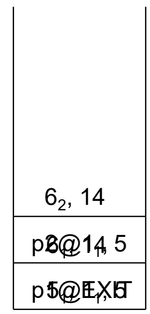
```
Branch (xi)
{
     CDS.push(<xi, IPD(xi)>);
}

Merge (tj)
{
      while (CDS.top( ).second == tj)
          CDS.pop( );
}

GetCurrentCD ( )
{
     return CDS.top( ).first;
}
```

Georgia Tech

36

"Idealistic" Algorithm Example

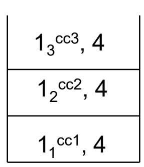


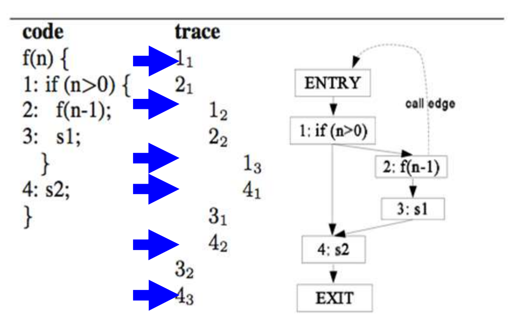


37

Even Handles Recursion!

- Annotate CDS entries with calling context
- Consider this recursive code and execution trace



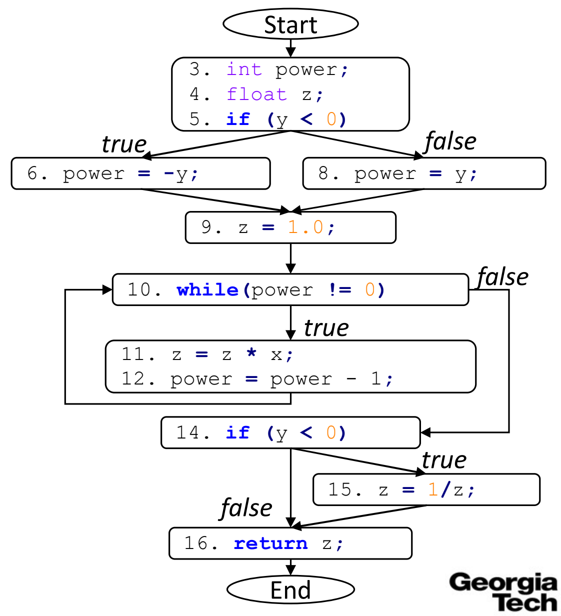


Source: B. Xin, X. Zhang. Efficient Online Detection of Dynamic Control Dependence, International Symposium on Software Testing and Analysis, 2007.

38

Challenges in Practice

1. You may not know the immediate post-dominator ahead of time!
   - This requires first collecting the control flow trace, then computing the IPDs, then updating the control dependence



Georgia Tech

39

Challenges in Practice

2. If you only have execution traces, then it is not clear what is a branch until you execute both paths!
   - In general, implementations use a hybrid of offline and online analyses to compute control dependence

Execution Trace (x=0, y=0)

```
3.  int power;
4.  float z;
5.  if (y < 0)
8.  power = y;
9.  z = 1.0;
10. while(power != 0)
14. if (y < 0)
16. return z;
```

Execution Trace (x=1, y=1)

```
3.  int power;
4.  float z;
5.  if (y < 0)
8.  power = y;
9.  z = 1.0;
10. while(power != 0)
11. z = z * x;
12. power = power - 1;
10. while(power != 0)
14. if (y < 0)
16. return z;
```

Georgia Tech

40

Advanced Topics in Malware Analysis

Program Slicing

Brendan Saltaformaggio, PhD

Assistant Professor

School of Electrical and Computer Engineering

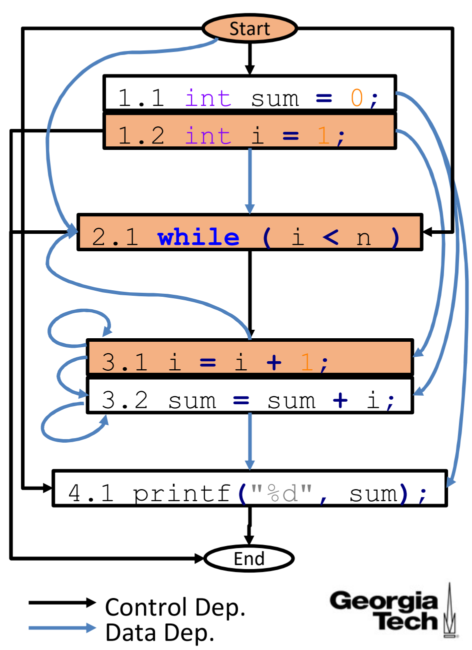
Dynamic Slice Concepts

Georgia Tech

41

Wrap up of Dynamic Slice Concepts

- We have introduced the concept of slicing, slice criterions, and dynamic slicing
- We have seen **two** approaches to computing a dynamic slice:
  1. **Offline dynamic slicing algorithms** based on backwards traversal over traces is not efficient
  2. **Online algorithms** that detect dependences are efficient but control dependence can be complex/limited
- In fact, this form of slicing is called **Backward Slicing**
- Slice (I @ 3.1) = {START, 1.2, 2.1, 3.1}



Georgia Tech

42

Forward Dynamic Slice Computation

- The approaches we have discussed so far are for backward slicing
  - Dependence graphs are traversed **backwards** from a slicing criterion
  - The space complexity is O(execution length)

- There is an orthogonal concept of Forward Slice Computation
- A forward slice of a program with respect to a program point p and variable v consists of all statements and predicates in the program that may be affected by the value of v at p
- Given a slice criterion, i.e., the starting point, a forward slice is computed by traversing the set of **forward-reachable** nodes in the program dependence graph

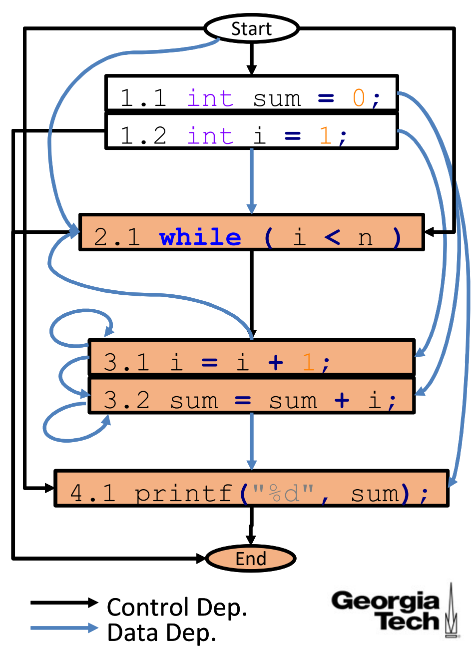
43

Forward Slice Computation

- **Step 1:** Build a PDG (as discussed previously)

- **Step 2:** Traverse the **forward-reachable** nodes in the PDG from the slice criterion

ForwardStaticSlice (I @ 3.1) =
{2.1, 3.1, 3.2, 4.1, END}



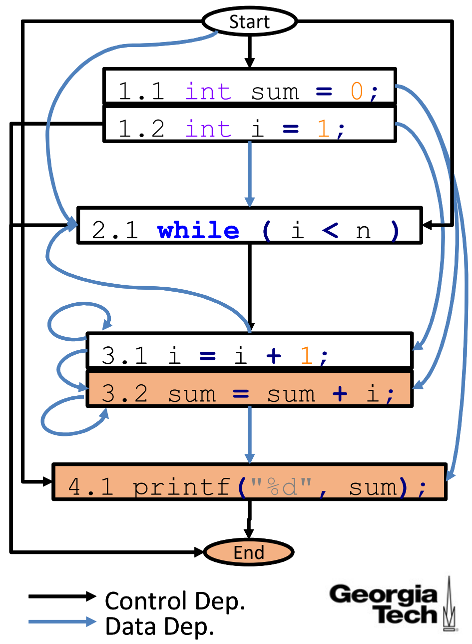
44

Forward Slice Computation

- **Step 1:** Build a PDG (as discussed previously)
- **Step 2:** Traverse the **forward-reachable** nodes in the PDG from the slice criterion

ForwardStaticSlice (I @ 3.1) =
{2.1, 3.1, 3.2, 4.1, END}

ForwardStaticSlice (sum @ 3.2) =
{3.2, 4.1, END}



45

The Importance of Forward Slicing

- Slicing is by far the **most commonly** employed dynamic analysis
- Answer the question: "*What program components might be affected by a particular computation?*"
- Useful in determining **which statements** in a program can be **affected** by changes in the value of **v** at statement **s**

46

The Importance of Forward Slicing: Example 1

- As software is maintained, modifications to parts of the program can lead to unforeseen side effects. When part of a program is changed, the effect of the change “ripples” through the program. Forward slicing exposes these effects.

47

The Importance of Forward Slicing: Example 2

- When analyzing malware, we want to focus on only the most important payloads (and not waste time on others). When the malware receives an input or checks a condition, that value will only affect some payloads. Forward slicing can give you foresight into what values will be important later in the execution.

48

Finally: Chopping

- Given a source criterion S and a target criterion T, determine what statements transmit the effects of S to T
- Simply the intersection of forward and backward slices

- **Step 1:** Compute a backward slice from the target criterion T
  - Recall: This is the set of all prior statements involved in computing the value of T.v at T.s
- **Step 2:** Compute a forward slice from the source criterion S
  - Recall: This is the set of all future statements affected by the value of S.v at S.s
- **Step 3:** Compute the intersection of the two graphs

49

Lesson Summary

- Locate the set of statements that compose a program slice
- Construct a program dependent graph
- Assemble dynamic slices using a program dependency graph

50