

Control- and Data-Dependence

Yulei Sui

University of Technology Sydney, Australia

Control- and Data-Dependence

What are control- and data-dependence?

- **Control-dependence**

- Execution order between two program statements/instructions.
- Can program point B be reached from point A in the control-flow graph of a program?
- Obtained through traversing the ICFG of a program

- **Data-dependence**

- Definition-use relation between two program variables.
- Will the definition of a variable X be used and passed to another variable Y?
- Obtained through analyzing the PAG of a program
- Combining PAG with ICFG to yield more precise flow-sensitive and context-sensitive data-dependence.

Control- and Data-Dependence

Why learn control- and data-dependence?

A program dependence relation by its nature is the reachability property on a graph, particularly useful in program understanding, optimizations and bug detection.

Control- and Data-Dependence

Why learn control- and data-dependence?

A program dependence relation by its nature is the reachability property on a graph, particularly useful in program understanding, optimizations and bug detection.

- **Applications of control-dependence**

- Dead code elimination: If a subgraph of an ICFG is not connected from the entry block of a program, that subgraph is possibly dead code.

Control- and Data-Dependence

Why learn control- and data-dependence?

A program dependence relation by its nature is the reachability property on a graph, particularly useful in program understanding, optimizations and bug detection.

- **Applications of control-dependence**

- Dead code elimination: If a subgraph of an ICFG is not connected from the entry block of a program, that subgraph is possibly dead code.
- Identifying infinite loops: If the exit block is unreachable from the entry block, an infinite loop may exist.
- ...

Control- and Data-Dependence

Why learn control- and data-dependence?

A program dependence relation by its nature is the reachability property on a graph, particularly useful in program understanding, optimizations and bug detection.

- **Applications of control-dependence**

- Dead code elimination: If a subgraph of an ICFG is not connected from the entry block of a program, that subgraph is possibly dead code.
- Identifying infinite loops: If the exit block is unreachable from the entry block, an infinite loop may exist.
- ...

- **Applications of data-dependence**

- Pointer alias analysis: statically determine possible runtime values of a pointer to detect memory errors, such as null pointer dereferences and use-after-frees.

Control- and Data-Dependence

Why learn control- and data-dependence?

A program dependence relation by its nature is the reachability property on a graph, particularly useful in program understanding, optimizations and bug detection.

- **Applications of control-dependence**

- Dead code elimination: If a subgraph of an ICFG is not connected from the entry block of a program, that subgraph is possibly dead code.
- Identifying infinite loops: If the exit block is unreachable from the entry block, an infinite loop may exist.
- ...

- **Applications of data-dependence**

- Pointer alias analysis: statically determine possible runtime values of a pointer to detect memory errors, such as null pointer dereferences and use-after-frees.
- Taint analysis: if two program variables v_1 and v_2 are aliases (e.g., representing the same memory location), if v_1 is tainted by user inputs, then v_2 is also tainted.
- ...

Control-Dependence

We say that a program statement (ICFG node) `snk` is control-flow dependent on `src` if `src` can reach `snk` on the ICFG.

- Context-insensitive control-dependence
 - control-flow traversal without matching calls and returns.
 - fast but imprecise

Control-Dependence

We say that a program statement (ICFG node) `snk` is control-flow dependent on `src` if `src` can reach `snk` on the ICFG.

- Context-insensitive control-dependence
 - control-flow traversal without matching calls and returns.
 - fast but imprecise
- Context-sensitive control-dependence
 - control-flow traversal by matching calls and returns.
 - precise but maintains an extra abstract call stack (storing a sequence of callsite ID information) to mimic the runtime call stack.

Control-Dependence

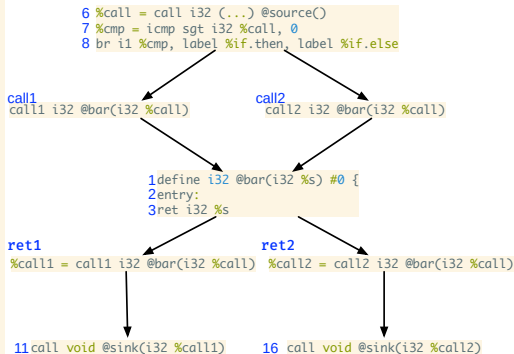
```
int bar(int s){
    return s;
}
int main(){
    int a = source();
    if (a > 0){
        int p = bar(a);
        sink(p);
    }else{
        int q = bar(a);
        sink(q);
    }
}
```

Control-Dependence

```
define i32 @bar(i32 %s) #0 {  
1 entry:  
2 ret i32 %s  
3}  
  
define i32 @main() #0 {  
4 entry:  
5 %call = call i32 (...) @source()  
6 %cmp = icmp sgt i32 %call, 0  
7 br i1 %cmp, label %if.then, label %if.else  
8  
9 if.then:                ; preds = %entry  
9 %call1 = call i32 @bar(i32 %call)  
10 call void @sink(i32 %call1)  
11 br label %if.end  
12  
13 if.else:                ; preds = %entry  
13 %call2 = call i32 @bar(i32 %call)  
14 call void @sink(i32 %call2)  
15 br label %if.end  
16  
17 if.end:                 ; preds = %if.else, %if.then  
17 ret i32 0  
18 }
```

Control-Dependence

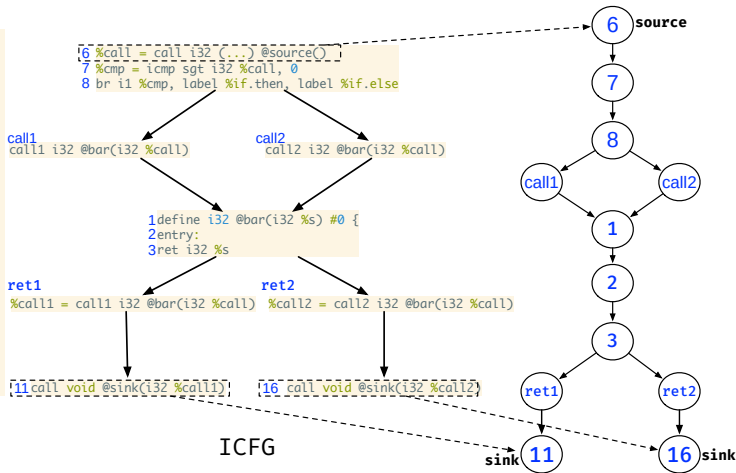
```
define i32 @bar(i32 %s) #0 {  
1 entry:  
2 ret i32 %s  
3}  
  
define i32 @main() #0 {  
4 entry:  
5 %call = call i32 (...) @source()  
6 %cmp = icmp sgt i32 %call, 0  
7 br i1 %cmp, label %if.then, label %if.else  
8  
9 if.then:      ; preds = %entry  
9 %call1 = call i32 @bar(i32 %call)  
10 call void @sink(i32 %call1)  
11 br label %if.end  
12  
13 if.else:     ; preds = %entry  
13 %call2 = call i32 @bar(i32 %call)  
14 call void @sink(i32 %call2)  
15 br label %if.end  
16  
17 if.end:      ; preds = %if.else, %if.then  
17 ret i32 0  
18}
```



ICFG

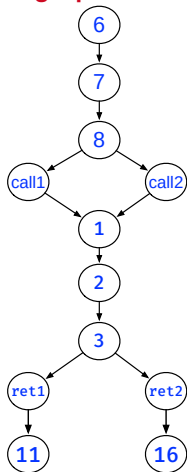
Control-Dependence

```
define i32 @bar(i32 %s) #0 {  
1 entry:  
2 ret i32 %s  
3}  
  
define i32 @main() #0 {  
4 entry:  
5 %call = call i32 (...) @source()  
6 %cmp = icmp sgt i32 %call, 0  
7 br i1 %cmp, label %if.then, label %if.else  
8  
9 if.then:                ; preds = %entry  
10 %call1 = call i32 @bar(i32 %call)  
11 call void @sink(i32 %call1)  
12  
13 if.else:                ; preds = %entry  
14 %call2 = call i32 @bar(i32 %call)  
15 call void @sink(i32 %call2)  
16  
17 if.end:                 ; preds = %if.else, %if.then  
18 ret i32 0  
19}
```



Context-Insensitive Control-Dependence

Obtaining a path from source to sink on ICFG



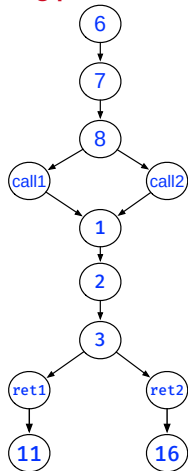
Basic DFS on ICFG: source \rightarrow sink

```
visited: set<NodeID>  
path: vector<NodeID>
```

```
DFS(visited, path, src, dst)  
  visited.insert(src);  
  path.push_back(src);  
  if src == dst then  
    Print path;  
  foreach edge e  $\in$  outEdges(src) do  
    if (e.dst  $\notin$  visited)  
      DFS(visited, path, e.dst, dst);  
  visited.erase(src);  
  path.pop_back();
```

Context-Insensitive Control-Dependence

Obtaining paths from node 6 to node 11 on the ICFG



Basic DFS on ICFG: source \rightarrow sink

```
visited: set<NodeID>
path: vector<NodeID>

DFS(visited, path, src, dst)
  visited.insert(src);
  path.push_back(src);
  if src == dst then
    Print path;
  foreach edge e  $\in$  outEdges(src) do
    if (e.dst  $\notin$  visited)
      DFS(visited, path, e.dst, dst);
  visited.erase(src);
  path.pop_back();
```

ICFG paths: node 6 \rightarrow node 11

Path 1:

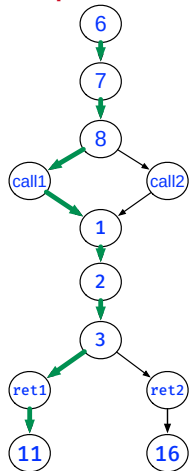
6 \rightarrow 7 \rightarrow 8 \rightarrow **call1** \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow **ret1** \rightarrow 11

Path 2:

6 \rightarrow 7 \rightarrow 8 \rightarrow **call2** \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow **ret1** \rightarrow 11

Context-Insensitive Control-Dependence

Feasible paths from node 6 to node 11



Basic DFS on ICFG: source \rightarrow sink

```
visited: set<NodeID>
path: vector<NodeID>

DFS(visited, path, src, dst)
  visited.insert(src);
  path.push_back(src);
  if src == dst then
    Print path;
  foreach edge e  $\in$  outEdges(src) do
    if (e.dst  $\notin$  visited)
      DFS(visited, path, e.dst, dst);
  visited.erase(src);
  path.pop_back();
```

ICFG paths: node 6 \rightarrow node 11

Path 1: **feasible path**

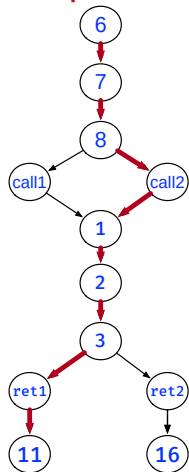
6 \rightarrow 7 \rightarrow 8 \rightarrow **call1** \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow **ret1** \rightarrow 11

Path 2:

6 \rightarrow 7 \rightarrow 8 \rightarrow **call2** \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow **ret1** \rightarrow 11

Context-Insensitive Control-Dependence

Infeasible path from node 6 to node 11



Basic DFS on ICFG: source \rightarrow sink

```
visited: set<NodeID>
path: vector<NodeID>

DFS(visited, path, src, dst)
  visited.insert(src);
  path.push_back(src);
  if src == dst then
    Print path;
  foreach edge e  $\in$  outEdges(src) do
    if (e.dst  $\notin$  visited)
      DFS(visited, path, e.dst, dst);
  visited.erase(src);
  path.pop_back();
```

ICFG paths: node 6 \rightarrow node 11

Path 1:

6 \rightarrow 7 \rightarrow 8 \rightarrow **call1** \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow **ret1** \rightarrow 11

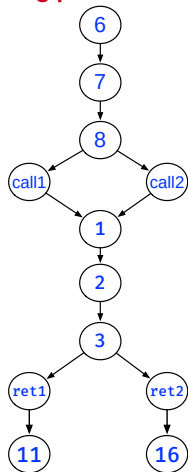
Path 2:

6 \rightarrow 7 \rightarrow 8 \rightarrow **call2** \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow **ret1** \rightarrow 11

spurious path

Context-Insensitive Control-Dependence

Obtaining paths from node 6 to node 16 on ICFG



Basic DFS on ICFG: source \rightarrow sink

```
visited: set<NodeID>
path: vector<NodeID>

DFS(visited, path, src, dst)
  visited.insert(src);
  path.push_back(src);
  if src == dst then
    Print path;
  foreach edge e  $\in$  outEdges(src) do
    if (e.dst  $\notin$  visited)
      DFS(visited, path, e.dst, dst);
  visited.erase(src);
  path.pop_back();
```

ICFG paths: node 6 \rightarrow node 16

Path 3:

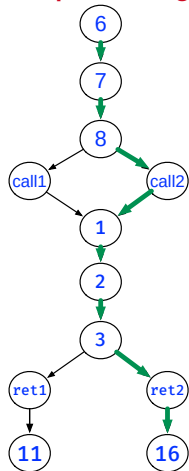
6 \rightarrow 7 \rightarrow 8 \rightarrow **call2** \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow **ret2** \rightarrow 16

Path 4:

6 \rightarrow 7 \rightarrow 8 \rightarrow **call1** \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow **ret2** \rightarrow 16

Context-Insensitive Control-Dependence

Feasible paths using from node 6 to node 16 on the ICFG



Basic DFS on ICFG: source → sink

```
visited: set<NodeID>
path: vector<NodeID>

DFS(visited, path, src, dst)
  visited.insert(src);
  path.push_back(src);
  if src == dst then
    Print path;
  foreach edge e ∈ outEdges(src) do
    if (e.dst ∉ visited)
      DFS(visited, path, e.dst, dst);
  visited.erase(src);
  path.pop_back();
```

ICFG paths: node 6 → node 16

Path 3: **feasible path**

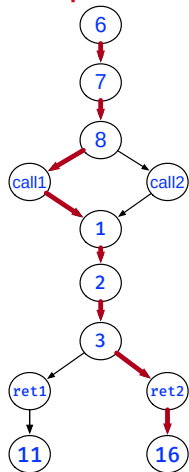
6 → 7 → 8 → **call2** → 1 → 2 → 3 → **ret2** → 16

Path 4:

6 → 7 → 8 → **call1** → 1 → 2 → 3 → **ret2** → 16

Context-Insensitive Control-Dependence

Infeasible paths using from node 6 to node 16 on the ICFG



Basic DFS on ICFG: source → sink

```
visited: set<NodeID>
path: vector<NodeID>

DFS(visited, path, src, dst)
  visited.insert(src);
  path.push_back(src);
  if src == dst then
    Print path;
  foreach edge e ∈ outEdges(src) do
    if (e.dst ∉ visited)
      DFS(visited, path, e.dst, dst);
  visited.erase(src);
  path.pop_back();
```

ICFG paths: node 6 → node 16

Path 3:

6 → 7 → 8 → **call2** → 1 → 2 → 3 → **ret2** → 16

Path 4:

6 → 7 → 8 → **call1** → 1 → 2 → 3 → **ret2** → 16

spurious path

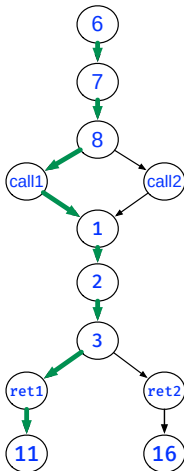
Context-Sensitive Control-Dependence

An extension of the context-insensitive algorithm by matching calls and returns.

- Get only feasible interprocedural paths and exclude infeasible ones
- Requires an extra callstack to store and mimic the runtime calling relations.

Context-Sensitive Control-Dependence

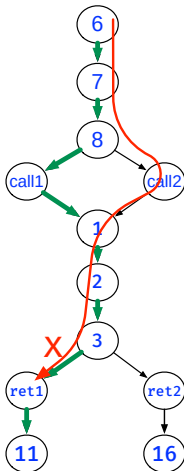
call1 matches with ret1



```
visited: set<NodeID>
path: vector<NodeID>
callstack: stack<callsite> //A stack of LLVM call instructions
DFS(visited, path, callstack, src, dst)
1  visited.insert(src)
2  path.push_back(src)
3  if src == dst then
4    Print path
5  foreach edge e ∈ outEdges(src) do
6    if e.dst ∉ visited then
7      if e.isIntraCFGEde() then
8        DFS(visited, path, callstack, e.dst, dst)
9      else if e.isCallCFGEde() then
10         callstack.push(e.getCallsite())
11         DFS(visited, path, callstack, e.dst, dst)
12      else if e.isRetCFGEde() then
13         if !callstack.empty() && callstack.top() == e.getCallsite() then
14             callstack.pop()
15             DFS(visited, path, callstack, e.dst, dst)
16  visited.erase(src);
17  path.pop_back();
```

Context-Sensitive Control-Dependence

call2 does not match with ret1



```
visited: set<NodeID>
path: vector<NodeID>
callstack: stack<callsite> //A stack of LLVM call instructions
DFS(visited, path, callstack, src, dst)
1  visited.insert(src)
2  path.push_back(src)
3  if src == dst then
4    Print path
5  foreach edge e ∈ outEdges(src) do
6    if e.dst ∉ visited then
7      if e.isIntraCFGEde() then
8        DFS(visited, path, callstack, e.dst, dst)
9      else if e.isCallCFGEde() then
10         callstack.push(e.getCallsite())
11         DFS(visited, path, callstack, e.dst, dst)
12      else if e.isRetCFGEde() then
13         if !callstack.empty() && callstack.top() == e.getCallsite() then
14             callstack.pop()
15             DFS(visited, path, callstack, e.dst, dst)
16  visited.erase(src);
17  path.pop_back();
```

What's next?

- (1) Understand control-dependence in this slides
- (2) Finish the quizzes of Assignment 2 on Canvas
- (3) Implement a context-sensitive ICFG traversal, i.e., coding task in Assignment 2
 - Refer to 'Assignment-2.pdf' on Canvas to know more.