

# Week 6 Assignment 2

Yulei Sui

University of Technology Sydney, Australia

# Assignment 2: Control-Dependence

## Context-Sensitive ICFG Traversal

- You will be using what you have learned about ICFG and context-sensitive graph traversal.
- **Goal:** implement a context-sensitive graph traversal on ICFG and print **feasible** paths from a source node to a sink node on the graph

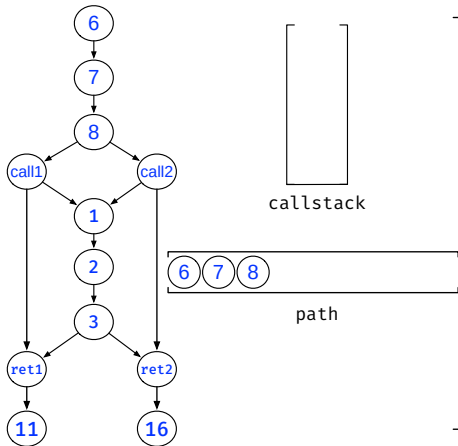
# Assignment 2: Control-Dependence

## Context-Sensitive ICFG Traversal

- You will be using what you have learned about ICFG and context-sensitive graph traversal.
- **Goal:** implement a context-sensitive graph traversal on ICFG and print **feasible** paths from a source node to a sink node on the graph
- **Specification and code template:**  
<https://github.com/SVF-tools/SVF-Teaching/wiki/Assignment-2>
- **SVF CPP API**  
<https://github.com/SVF-tools/SVF-Teaching/wiki/SVF-CPP-API>

# Context-Sensitive Control-Dependence

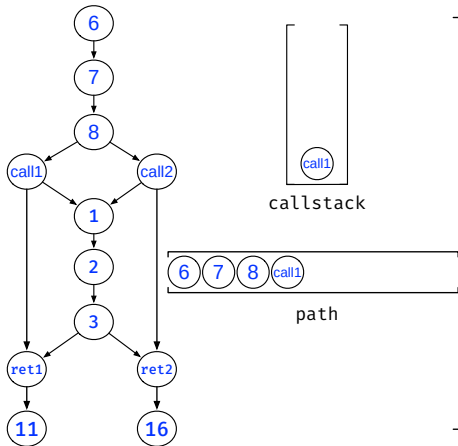
Obtaining a path from node 6 to node 11 on ICFG



```
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

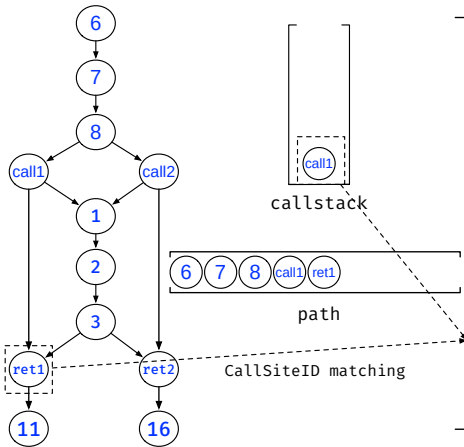
Obtaining a path from node 6 to node 11 on ICFG



```
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

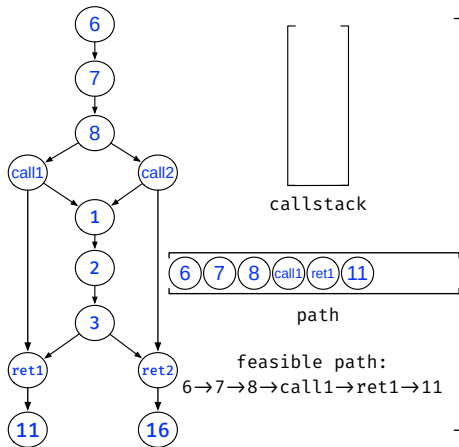
Obtaining a path from node 6 to node 11 on ICFG



```
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

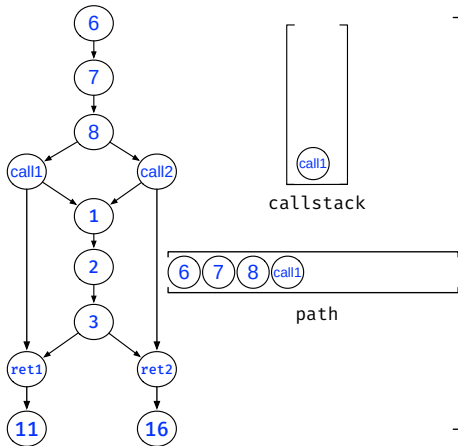
Obtaining a path from node 6 to node 11 on ICFG



```
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

Obtaining a path from node 6 to node 11 on ICFG

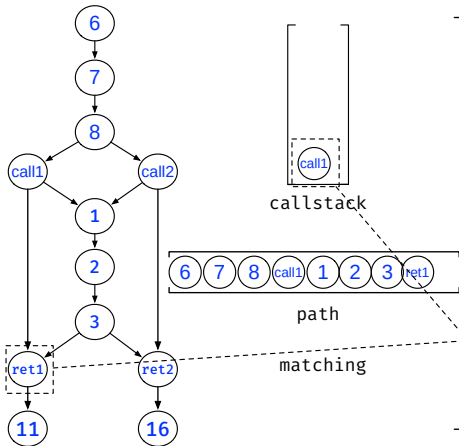


```
visited: set<NodeID>
path: vector<NodeID>
callstack: stack<callsite> //A stack of LLVM call instructions
DFS(visited, path, callstack, src, dst)
1 visited.insert(src)
2 path.push_back(src)
3 if src == dst then
4   Print path
5 foreach edge e ∈ outEdges(src) do
6   if e.dst ∉ visited then
7     if 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

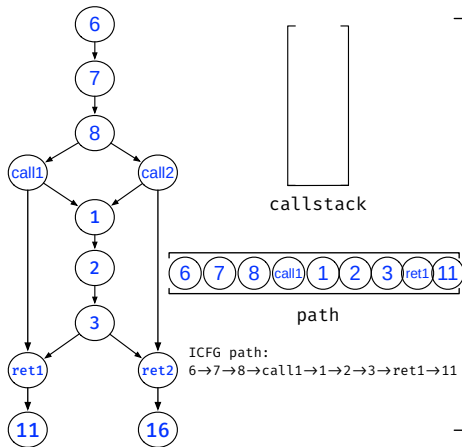
Obtaining a path from node 6 to node 11 on ICFG



```
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

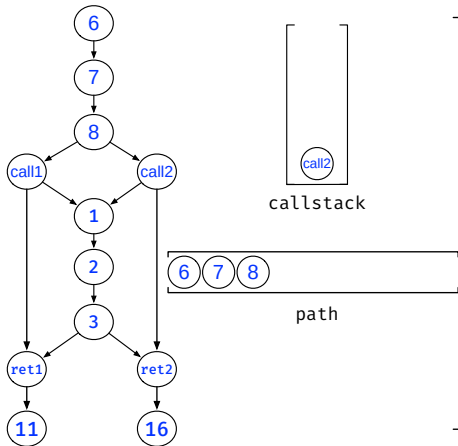
Obtaining a path from node 6 to node 11 on ICFG



```
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

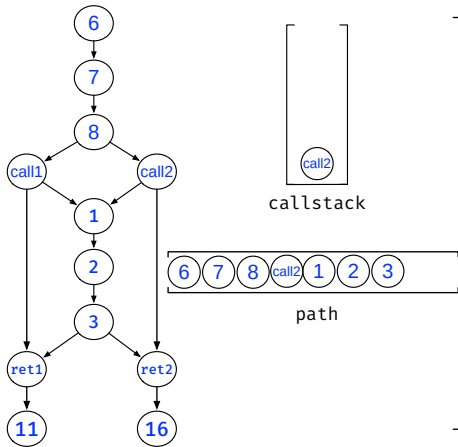
Obtaining a path from node 6 to node 11 on ICFG



```
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

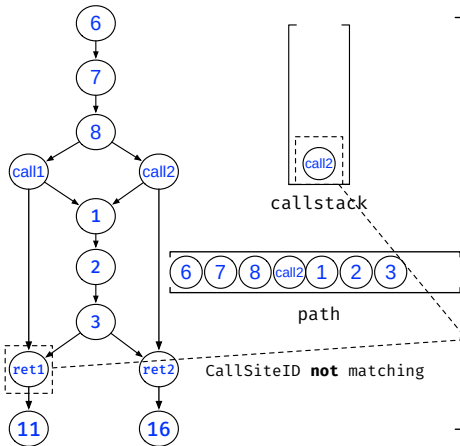
Obtaining a path from node 6 to node 11 on ICFG



```
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();
```

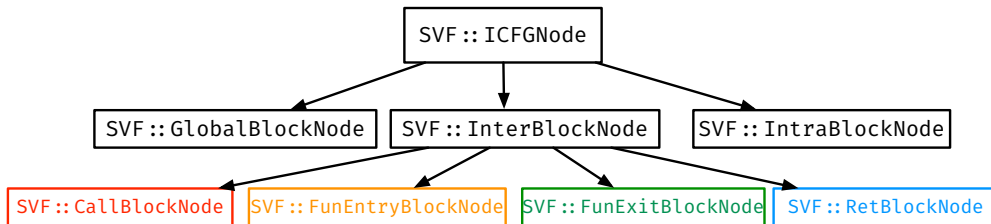
# Context-Sensitive Control-Dependence

Obtaining a path from node 6 to node 11 on ICFG



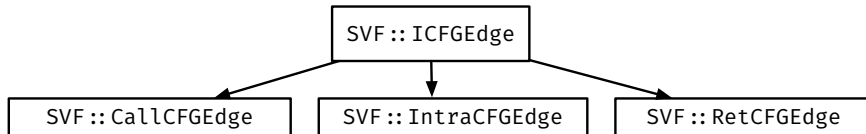
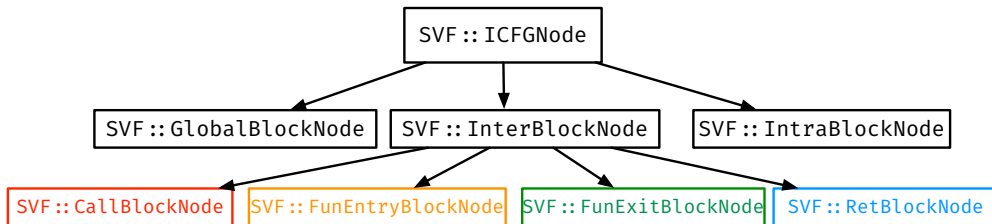
```
visited: set<NodeID>
path: vector<NodeID>
callstack: stack<callsite> //A stack of LLVM call instructions
DFS(visited, path, callstack, src, dst)
1 visited.insert(src)
2 path.push_back(src)
3 if src == dst then
4   Print path
5 foreach edge e ∈ outEdges(src) do
6   if e.dst ∉ visited then
7     if 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();
```

# ICFG Node and Edge Classes



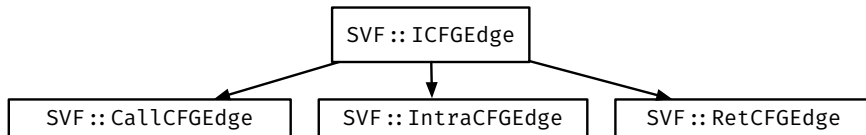
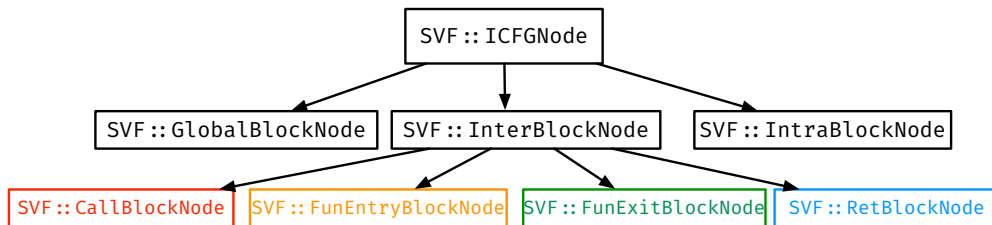
<https://github.com/SVF-tools/SVF/blob/master/include/Graphs/ICFGNode.h>

# ICFG Node and Edge Classes



<https://github.com/SVF-tools/SVF/blob/master/include/Graphs/ICFGEde.h>

# ICFG Node and Edge Classes



<https://github.com/svf-tools/SVF/wiki/Analyze-a-Simple-C-Program#4-interprocedural-control-flow-graph>



## cast and dyn\_cast

- C++ Inheritance: see slides in Week 2:.
- Casting a **parent** class pointer to pointer of a **Child** type:
  - `SVFUtil::cast`
    - Casting a pointer or reference to an instance of a specified class. This casting fails and abort the program if the object or reference is not the specified class at runtime.
  - `SVFUtil::dyn_cast`
    - "checking cast" operation. It checks to see if the operand is of the specified type, and if so, returns a pointer to it (this operator does not work with references). If the operand is not of the correct type, a null pointer is returned.
    - works very much like the `dynamic_cast<>` operator in C++, and should be used in the same circumstances.
- Example: Accessing the attributes of the child class via casting.
  - `RetBlockNode* retNode = SVFUtil::cast<RetBlockNode>(ICFGNode);`
  - `CallCFGEde* callEdge = SVFUtil::dyn_cast<CallCFGEde>(ICFGEde);`