Control-Flow and Reachability Analysis

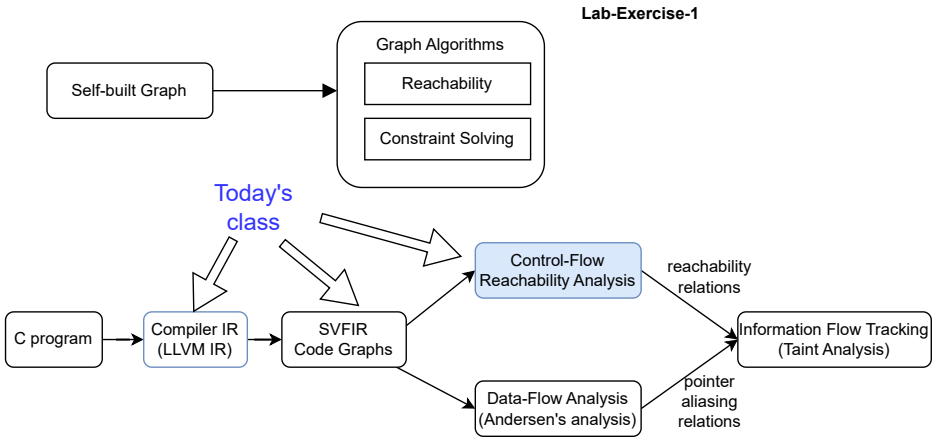
(Week 2)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Today's Class



Control-Flow and Data-Flow

What are control-flow and data-flow?

- **Control-control or 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-data or 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 SVFIR of a program
  - Combining SVFIR with ICFG to conduct symbolic execution (mimic the runtime path-based execution) of a program.

Program 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.

Program 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.

Program 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.
  - ...

Program 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.

Program 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 v1 and v2 are aliases (e.g., representing the same memory location), if v1 is tainted by user inputs, then v2 is also tainted.
  - ...

Control-Flow

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-Flow

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-Flow

```c
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);
    }
}
```

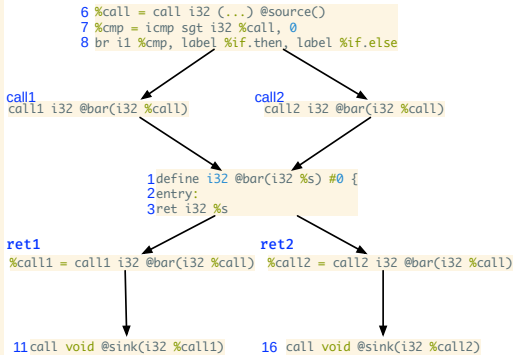
https://github.com/SVF-tools/Software-Security-Analysis/blob/main/SVFIR/src/control-flow.c

Control-Flow

```
define i32 @bar(i32 %s) #0 {
entry:
ret i32 %s
}

define i32 @main() #0 {
entry:
%call = call i32 (...) @source()
%cmp = icmp sgt i32 %call, 0
br i1 %cmp, label %if.then, label %if.else

if.then:          ; preds = %entry
%call1 = call i32 @bar(i32 %call)
call void @sink(i32 %call1)
br label %if.end

if.else:          ; preds = %entry
%call2 = call i32 @bar(i32 %call)
call void @sink(i32 %call2)
br label %if.end

if.end:       ; preds = %if.else, %if.then
ret i32 0
}
```

Control-Flow

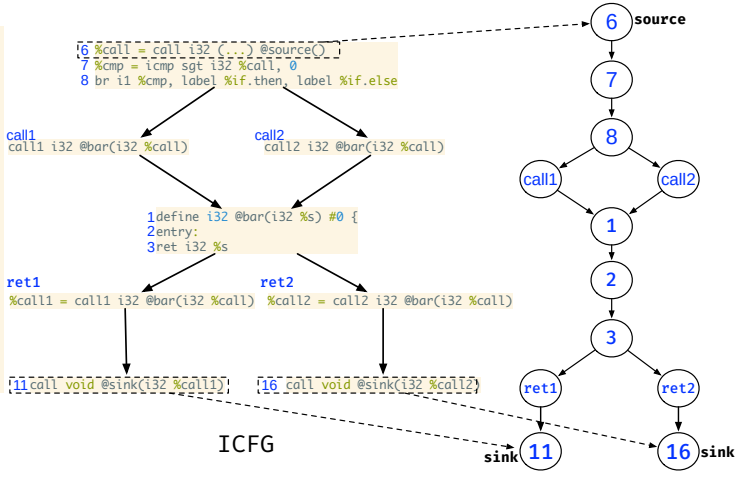
```
1  define i32 @bar(i32 %s) #0 {
2  entry:
3  ret i32 %s
   }

   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

   if.then:          ; preds = %entry
9  %call1 = call i32 @bar(i32 %call)
10 call void @sink(i32 %call1)
11 br label %if.end

12 
   if.else:          ; preds = %entry
13 %call2 = call i32 @bar(i32 %call)
14 call void @sink(i32 %call2)
15 br label %if.end

16 
   if.end:       ; preds = %if.else, %if.then
17 ret i32 0
18 }
```



ICFG

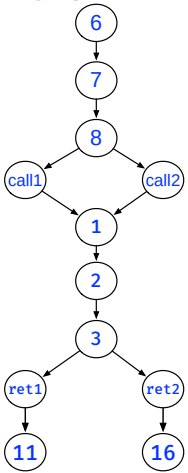
Control-Flow

```
1  define i32 @bar(i32 %s) #0 {
2  entry:
3  ret i32 %s
   }

   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

   if.then:          ; preds = %entry
9  %call1 = call i32 @bar(i32 %call)
10 call void @sink(i32 %call1)
11 br label %if.end

12 if.else:          ; preds = %entry
13 %call2 = call i32 @bar(i32 %call)
14 call void @sink(i32 %call2)
15 br label %if.end

16 if.end:        ; preds = %if.else, %if.then
17 ret i32 0
18 }
```



Context-Insensitive Control-Flow

Obtaining a path from source to sink on ICFG

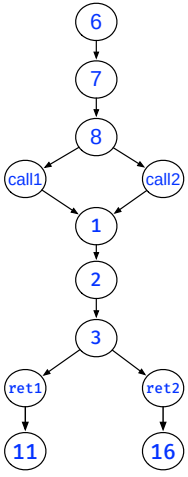


Basic DFS on ICFG: source → sink

```
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();
```

Context-Insensitive Control-Flow

Obtaining paths from node 6 to node 11 on the ICFG



Basic DFS on ICFG: source → sink

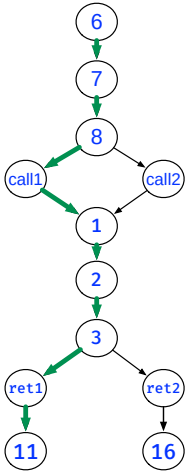
```
visited: set<NodeID>
path: vector<NodeID>

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 11

Path 1:
6→7→8→**call1**→1→2→3→**ret1**→11
Path 2:
6→7→8→**call2**→1→2→3→**ret1**→11

Context-Insensitive Control-Flow

Feasible paths from node 6 to node 11



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 11

Path 1: **feasible path**

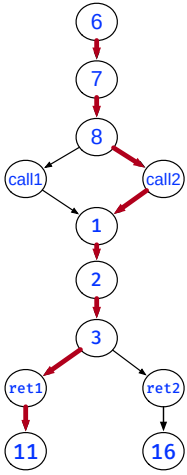
6→7→8→**call1**→1→2→3→**ret1**→11

Path 2:

6→7→8→**call2**→1→2→3→**ret1**→11

Context-Insensitive Control-Flow

Infeasible path from node 6 to node 11



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 11

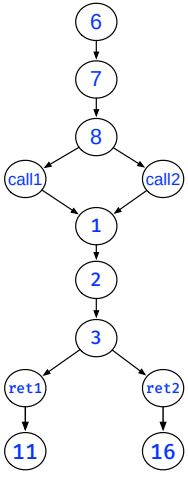
Path 1:
6→7→8→**call1**→1→2→3→**ret1**→11

Path 2:
6→7→8→**call2**→1→2→3→**ret1**→11

spurious path

Context-Insensitive Control-Flow

Obtaining paths from node 6 to node 16 on ICFG



Basic DFS on ICFG: source → sink

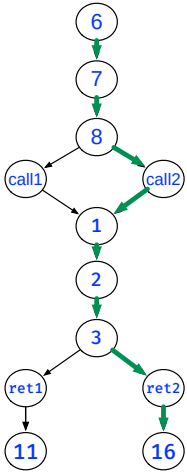
```
visited: set<NodeID>
path: vector<NodeID>

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

Context-Insensitive Control-Flow

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**

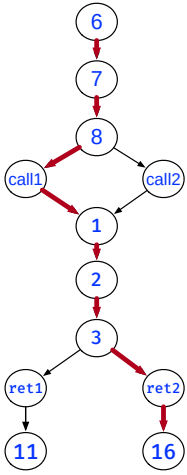
6→7→8→**call2**→1→2→3→**ret2**→16

Path 4:

6→7→8→**call1**→1→2→3→**ret2**→16

Context-Insensitive Control-Flow

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-Flow (Algorithm)

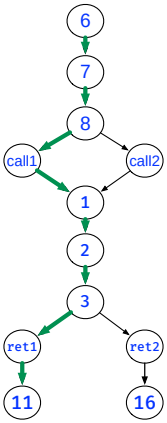
Algorithm 1: 1 Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩ callstack : vector⟨SVFInstruction⟩
visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4        return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8        collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11         dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13         callstack.push_back(edge.getCallSite());
14         dfs(edge.dst, snk);
15         callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17         if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18             callstack.pop_back();
19             dfs(edge.dst, snk);
20             callstack.push_back(edge.getCallSite());
21         else if callstack == ∅ then
22             dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence (Example)

call1 matches with ret1



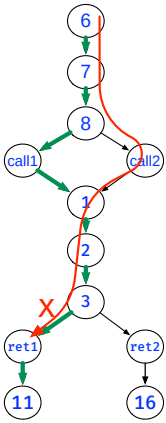
Algorithm 2: 1 Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence (Example)

call2 does not match with ret1



Algorithm 3: 1 Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

What's next?

- Understand control-flow reachability in this slides
- Debug and work with the code under the `SVFIR` and `CodeGraph` folders
- If you finished `Quiz-1` and `Lab-Exercise-1`, you could have a look at the spec of `Assignment-1`. Once the data flow is taught in Week 3, you could start coding `Assignment-1`
  - `Assignment-1`'s specification: `https://github.com/SVF-tools/Software-Security-Analysis/wiki/Assignment-1`

Lab: Control-Dependence and Reachability

(Week 2)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

Quiz-1 + Lab-Exercise-1 + Assignment-1

- A set of quizzes on WebCMS (5 points) due on **Week 3 Wednesday 23:59**
  - LLVM compiler and its intermediate representation
  - Code graphs (including ICFG and PAG)
- Lab-Exercise-1 (5 points) due on **Week 3 Wednesday 23:59**
  - Implement a graph traversal on a general graph
- Assignment-1 (20 points) due on **Week 4 Wednesday 23:59**
  - **Control-flow**: Implement a context-sensitive graph traversal on a CodeGraph (i.e., ICFG) and collect **feasible** paths from a source to a sink node on the ICFG.
  - **Data-flow**: Implement Andersen's inclusion-based constraint solving for points-to analysis
  - Implement a taint checker **using control-flow and data-flow analysis**.

Quiz-1 + Lab-Exercise-1 + Assignment-1

- A set of quizzes on WebCMS (5 points) due on **Week 3 Wednesday 23:59**
  - LLVM compiler and its intermediate representation
  - Code graphs (including ICFG and PAG)
- Lab-Exercise-1 (5 points) due on **Week 3 Wednesday 23:59**
  - Implement a graph traversal on a general graph
- Assignment-1 (20 points) due on **Week 4 Wednesday 23:59**
  - **Control-flow**: Implement a context-sensitive graph traversal on a CodeGraph (i.e., ICFG) and collect **feasible** paths from a source to a sink node on the ICFG.
  - **Data-flow**: Implement Andersen's inclusion-based constraint solving for points-to analysis
  - Implement a taint checker **using control-flow and data-flow analysis**.
  - **Specification and code template**: `https://github.com/SVF-tools/Software-Security-Analysis/wiki/Assignment-1`
  - **SVF APIs for control- and data-flow analysis** `https://github.com/SVF-tools/Software-Security-Analysis/wiki/SVF-CPP-API`

Context-Sensitive Control-Flow (Algorithm)

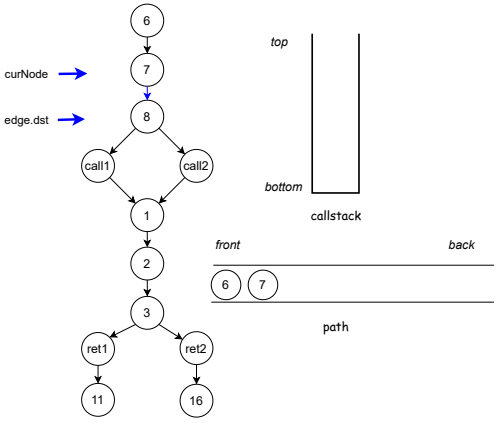
Algorithm 4: 1 Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4        return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8        collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11         dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13         callstack.push_back(edge.getCallSite());
14         dfs(edge.dst, snk);
15         callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17         if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18             callstack.pop_back();
19             dfs(edge.dst, snk);
20             callstack.push_back(edge.getCallSite());
21         else if callstack == ∅ then
22             dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG



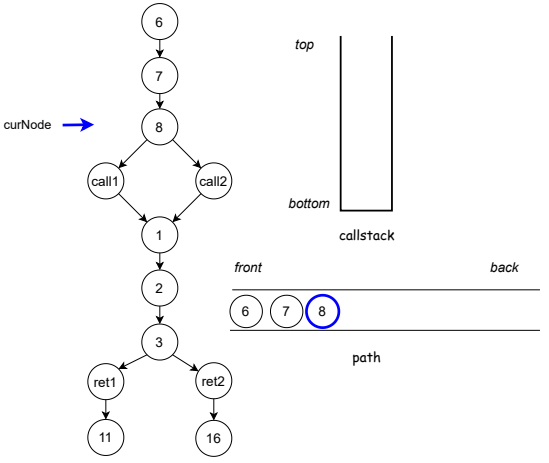
Algorithm 5: 1 Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG

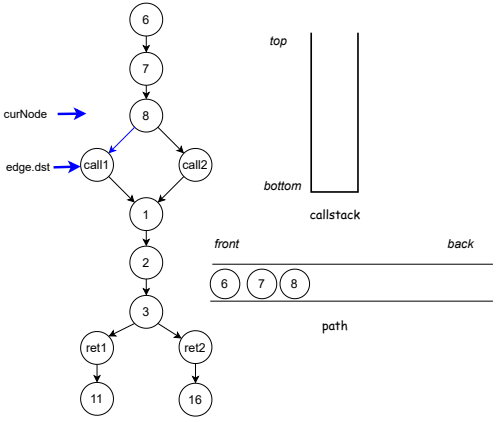


Algorithm 6: 1 Context sensitive control-flow reachability

```
Input : curNode : ICFGNode  snk : ICFGNode  path : vector⟨ICFGNode⟩
        callstack : vector⟨SVFInstruction⟩  visited : set⟨ICFGNode, callstack⟩;
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4        return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8        collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10       if edge.isIntraCFGEdge() then
11           dfs(edge.dst, snk);
12       else if edge.isCallCFGEdge() then
13           callstack.push_back(edge.getCallSite());
14           dfs(edge.dst, snk);
15           callstack.pop_back();
16       else if edge.isRetCFGEdge() then
17           if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18               callstack.pop_back();
19               dfs(edge.dst, snk);
20               callstack.push_back(edge.getCallSite());
21           else if callstack == ∅ then
22               dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG



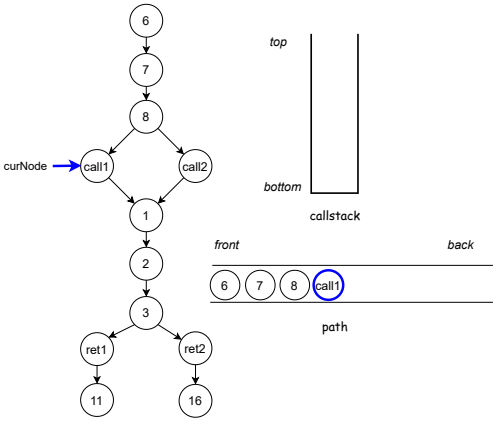
Algorithm 7: 1 Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG

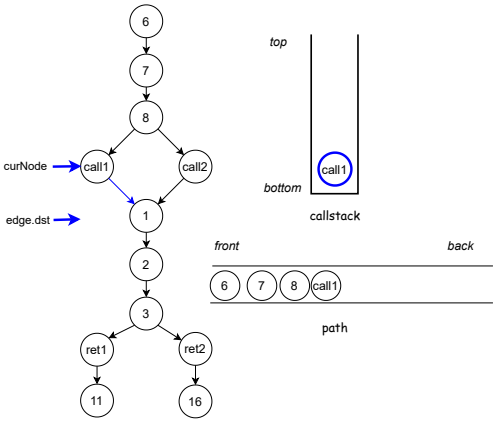


Algorithm 8: 1 Context sensitive control-flow reachability

```
Input : curNode : ICFGNode  snk : ICFGNode  path : vector⟨ICFGNode⟩
        callstack : vector⟨SVFInstruction⟩  visited : set⟨ICFGNode, callstack⟩;
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG



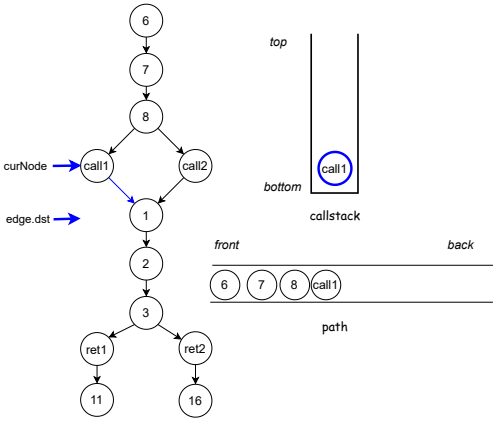
Algorithm 9: 1 Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG



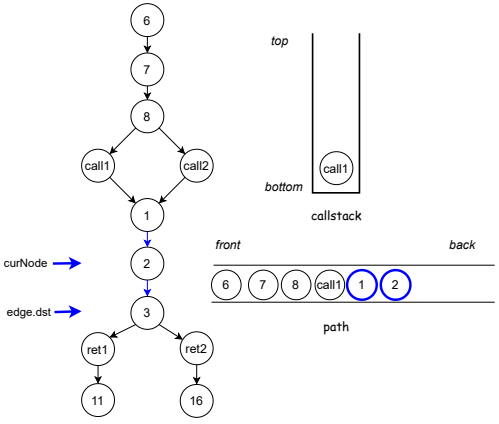
Algorithm 10: 1 Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG



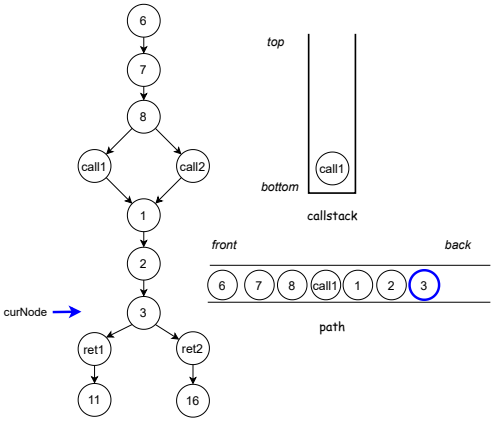
Algorithm 11: 1 Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG



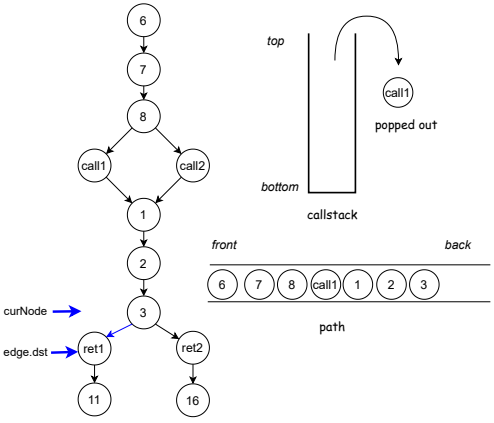
Algorithm 12: 1 Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG



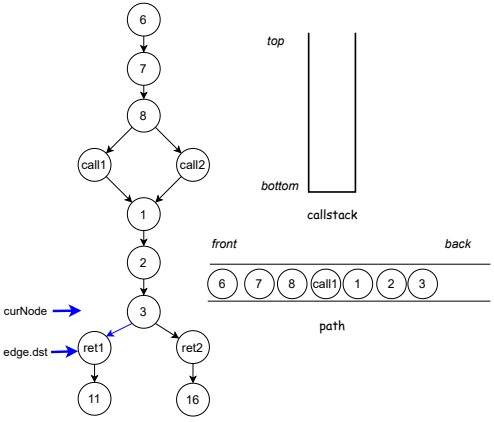
Algorithm 13: 1 Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG



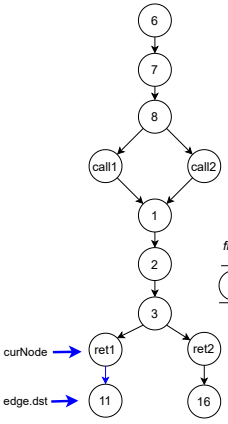
Algorithm 14: 1 Context sensitive control-flow reachability

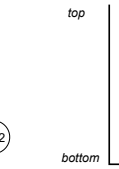
Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

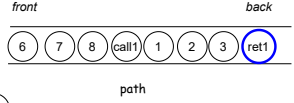
```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG





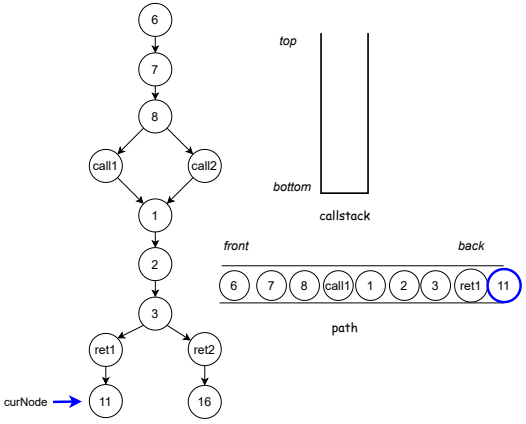


Algorithm 15: 1 Context sensitive control-flow reachability

```
Input : curNode : ICFGNode  snk : ICFGNode  path : vector⟨ICFGNode⟩
        callstack : vector⟨SVFInstruction⟩  visited : set⟨ICFGNode, callstack⟩;
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG

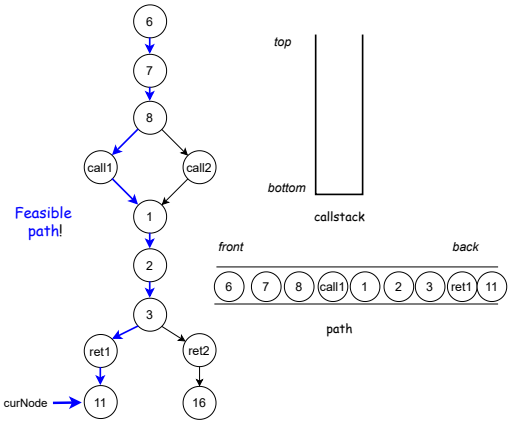


Algorithm 16: 1 Context sensitive control-flow reachability

```
Input : curNode : ICFGNode  snk : ICFGNode  path : vector⟨ICFGNode⟩
        callstack : vector⟨SVFInstruction⟩  visited : set⟨ICFGNode, callstack⟩;
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG



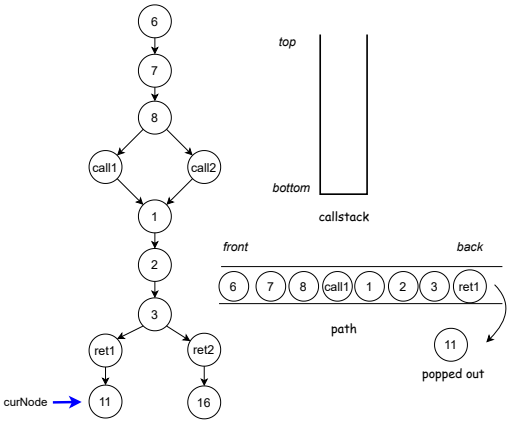
Algorithm 17: 1 Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG

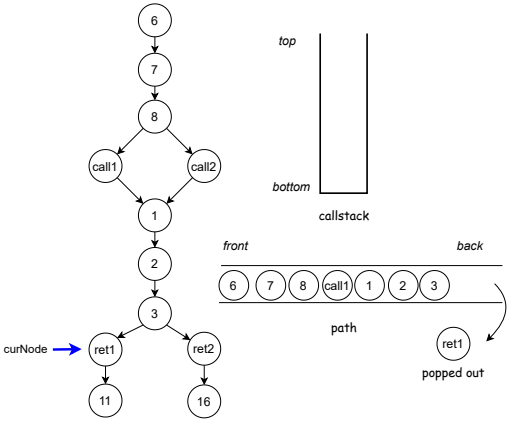


Algorithm 18: 1 Context sensitive control-flow reachability

```
Input : curNode : ICFGNode  snk : ICFGNode  path : vector⟨ICFGNode⟩
        callstack : vector⟨SVFInstruction⟩  visited : set⟨ICFGNode, callstack⟩;
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG

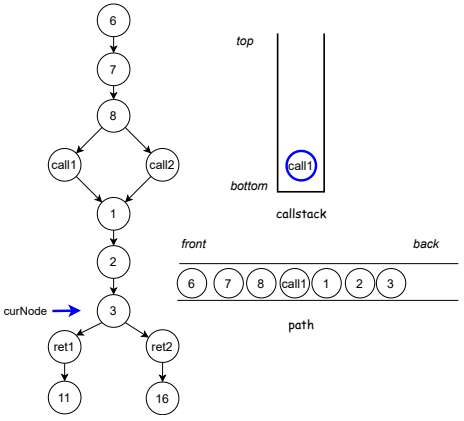


Algorithm 19: 1 Context sensitive control-flow reachability

```
Input : curNode : ICFGNode  snk : ICFGNode  path : vector⟨ICFGNode⟩
        callstack : vector⟨SVFInstruction⟩  visited : set⟨ICFGNode, callstack⟩;
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG



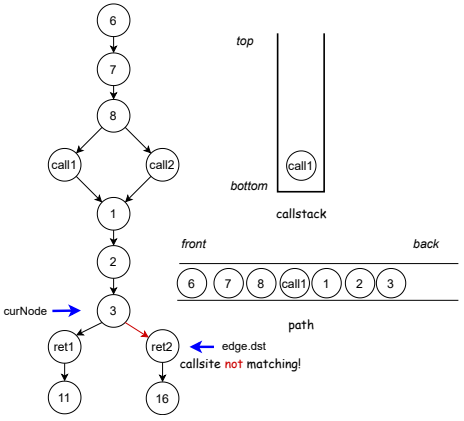
Algorithm 20: 1 Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG

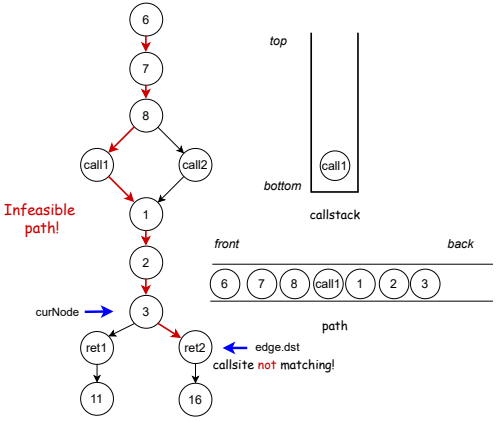


Algorithm 21: 1 Context sensitive control-flow reachability

```
Input : curNode : ICFGNode  snk : ICFGNode  path : vector⟨ICFGNode⟩
        callstack : vector⟨SVFInstruction⟩  visited : set⟨ICFGNode, callstack⟩;
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG



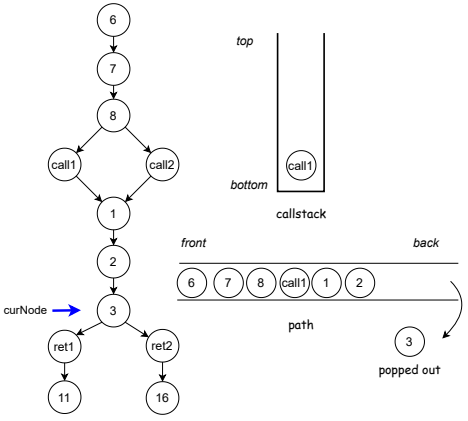
Algorithm 22: 1 Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG

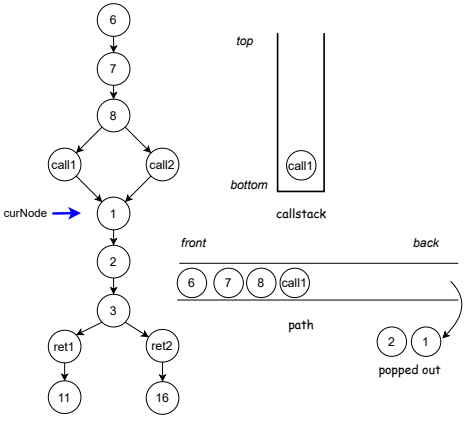


Algorithm 23: 1 Context sensitive control-flow reachability

```
Input : curNode : ICFGNode  snk : ICFGNode  path : vector⟨ICFGNode⟩
        callstack : vector⟨SVFInstruction⟩  visited : set⟨ICFGNode, callstack⟩;
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG

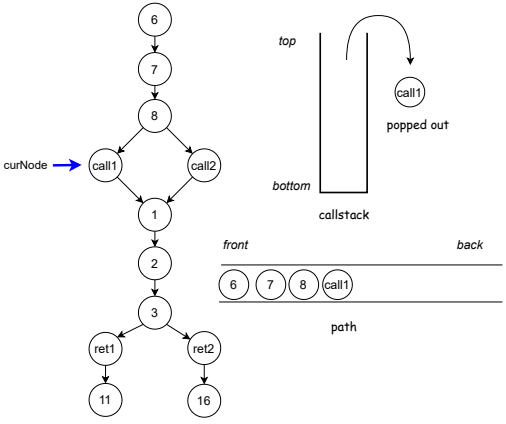


Algorithm 24: 1 Context sensitive control-flow reachability

```
Input : curNode : ICFGNode  snk : ICFGNode  path : vector⟨ICFGNode⟩
        callstack : vector⟨SVFInstruction⟩  visited : set⟨ICFGNode, callstack⟩;
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG



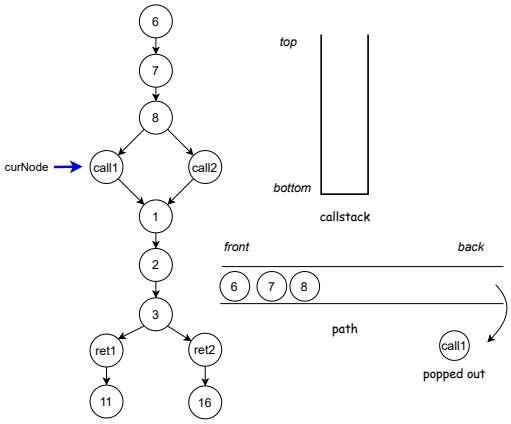
Algorithm 25: 1 Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG



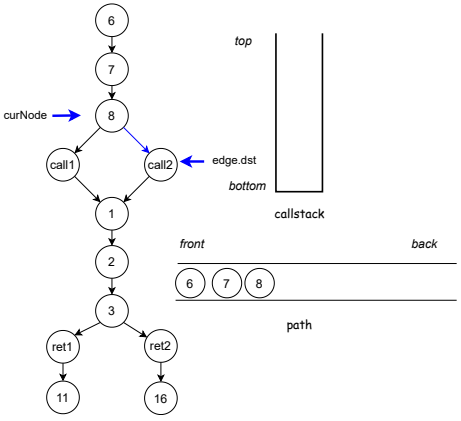
Algorithm 26: 1 Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG



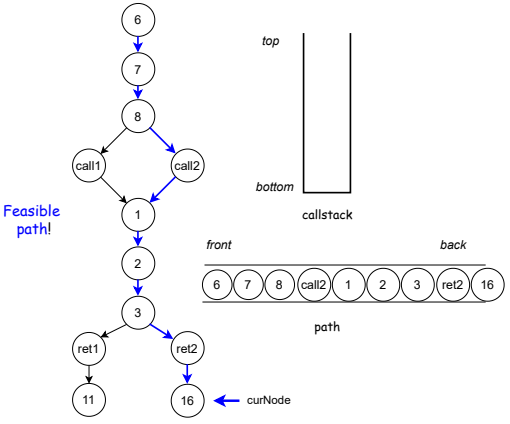
Algorithm 27: 1 Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG



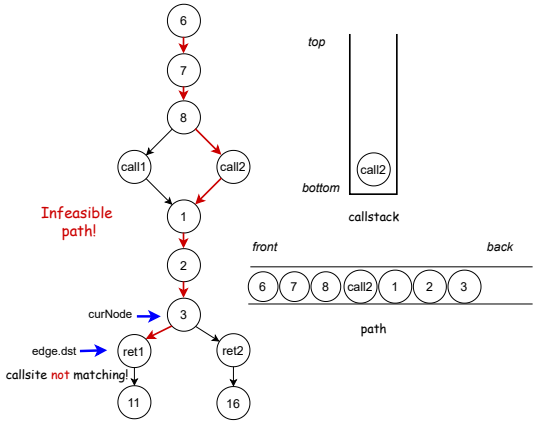
Algorithm 28: 1 Context sensitive control-flow reachability

input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

Context-Sensitive Control-Dependence

A feasible path from node 6 to node 11 on ICFG

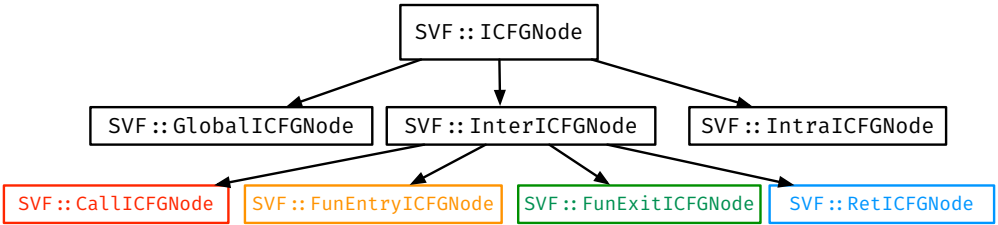


Algorithm 29: 1 Context sensitive control-flow reachability

Input : curNode : ICFGNode snk : ICFGNode path : vector⟨ICFGNode⟩
callstack : vector⟨SVFInstruction⟩ visited : set⟨ICFGNode, callstack⟩;

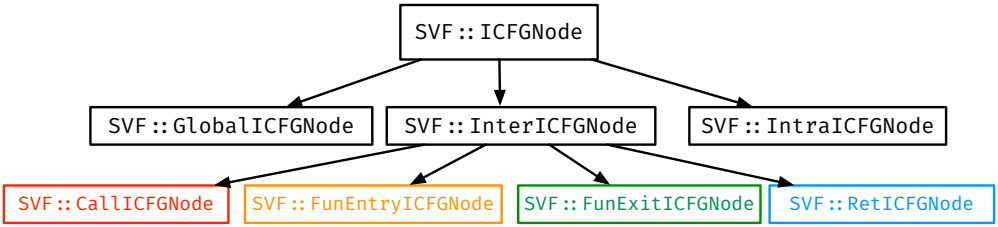
```
1  dfs(curNode, dst)
2    pair = ⟨curNode, callstack⟩;
3    if pair ∈ visited then
4      return;
5    visited.insert(pair);
6    path.push_back(curNode);
7    if src == snk then
8      collectICFGPath(path);
9    foreach edge ∈ curNode.getOutEdges() do
10     if edge.isIntraCFGEdge() then
11       dfs(edge.dst, snk);
12     else if edge.isCallCFGEdge() then
13       callstack.push_back(edge.getCallSite());
14       dfs(edge.dst, snk);
15       callstack.pop_back();
16     else if edge.isRetCFGEdge() then
17       if callstack ≠ ∅ && callstack.back() == edge.getCallSite() then
18         callstack.pop_back();
19         dfs(edge.dst, snk);
20         callstack.push_back(edge.getCallSite());
21       else if callstack == ∅ then
22         dfs(edge.dst, snk);
23   visited.erase(pair);
24   path.pop_back();
```

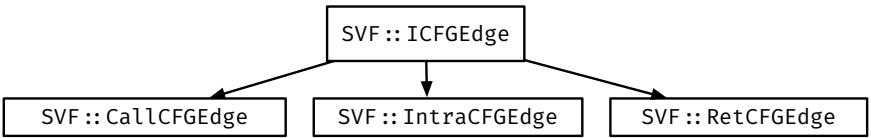
ICFG Node and Edge Classes



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/ICFGEdge.h

SVFUtil::cast and SVFUtil::dyn_cast

- C++ Inheritance: see slides in Week 1 Lab.
- Casting a **parent** class pointer to pointer of a **Child** type:
  - SVFUtil::cast
    - Casts a pointer or reference to an instance of a specified class. This cast fails and aborts the program if the object or reference is not the specified class at runtime.
  - SVFUtil::dyn_cast
    - "Checked cast" operation. 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);`
  - `CallCFGEdge* callEdge = SVFUtil::dyn_cast<CallCFGEdge>(ICFGEdge);`