Lab: SVFIR and Control-Flow Reachability

(Week 2)

Yulei Sui

School of Computer Science and Engineering

University of New South Wales, Australia

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

Task	Due	Relevant Topics	Focus
Lab-Quiz-1 (5 points)	Week-3 Wed 23:59	Programming practices; LLVM/SVF IR; graphs	Foundational knowledge
Lab-Exercise-1 (5 points)	Week-3 Wed 23:59	**Context-insensitive** reachability analysis; **Field-insensitive** Andersen's points-to analysis	Control- and data-flows on **self-constructed graphs**
Assignment-1 (20 points)	Week-4 Wed 23:59	**Context-sensitive** reachability analysis; **Field-sensitive** Andersen's points-to analysis	Control- and data-flows on **code graphs** (ICFG and PAG)

Understanding LLVM-IR and SVF-IR

- (1) Compile C programs under `SVFIR/src` into their LLVM IR and print their SVF IR (PAG, ICFG, Constraint Graph).
  - `https://github.com/SVF-tools/Software-Security-Analysis/wiki/SVFIR`
  - Understand the mapping from a C program to its corresponding LLVM IR
  - Understand the mapping from LLVM IR to its corresponding SVF IR
- (2) Generate and visualize the graph representation of SVF IR (e.g., `example.ll.pag.dot`, `example.ll.icfg.dot`, `consG.ll.dot`).
  - `https://github.com/SVF-tools/Software-Security-Analysis/wiki/SVFIR#4-visualize-icfg-constraint-graph-and-svfirpag-graph`
- (3) Write code to iterate SVFVars and print nodes/edges of PAG and ICFG.
  - `https://github.com/SVF-tools/Software-Security-Analysis/blob/main/SVFIR/SVFIR.cpp#L74-L98` (C++)
  - `https://github.com/SVF-tools/SVF-Python/tree/main/demo` (Python)
- (4) More about LLVM IR and SVF's graph representation
  - LLVM language manual `https://llvm.org/docs/LangRef.html`
  - SVF website `https://github.com/SVF-tools/SVF`

Control-Flow Reachability

(Week 2)

Yulei Sui, Mohamad Barbar, Hanyuan Li, Angela He

School of Computer Science and Engineering

University of New South Wales, Australia

Algorithm overview

- Problem: context-insensitive reachability generates *impossible* paths

Algorithm overview

- Problem: context-insensitive reachability generates *impossible* paths
- Observation 1: in ICFGs, each call node has corresponding return node

Algorithm overview

- Problem: context-insensitive reachability generates *impossible* paths
- Observation 1: in ICFGs, each call node has corresponding return node
- Observation 2: inner functions ”exit” before outer functions

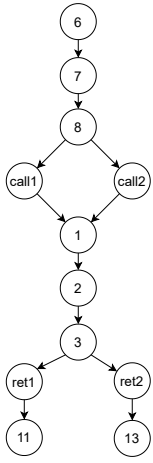
Algorithm overview

- Problem: context-insensitive reachability generates *impossible* paths
- Observation 1: in ICFGs, each call node has corresponding return node
- Observation 2: inner functions ”exit” before outer functions
- Solution: introduce **callstack** to track function calls/returns

Full algorithm

1. Represent each DFS state as: $\langle$current node, call stack$\rangle$
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring to avoid cycles.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

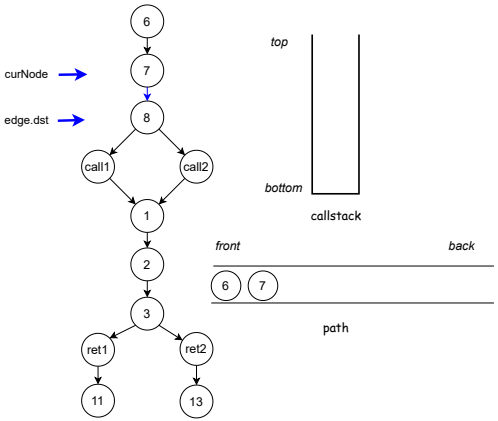
Example ICFG



The course repo can generate ICFG diagrams from LLVM IR files. To do so, follow the instructions at:
`https://github.com/SVF-tools/Software-Security-Analysis/wiki/SVFIR`.

Context-Sensitive Control-Flow Reachability

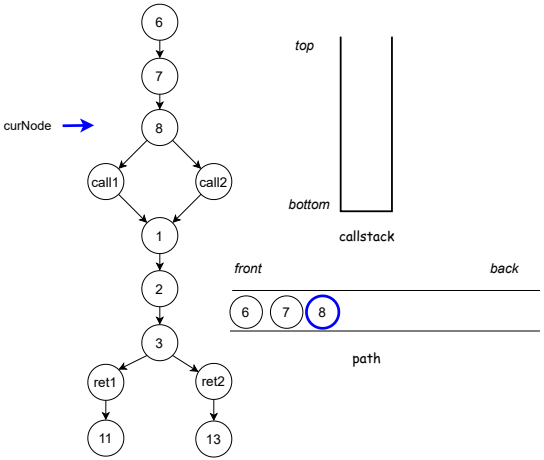
A feasible path from node 6 to node 11 on ICFG



1. Represent each DFS state as: ⟨current node, call stack⟩
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

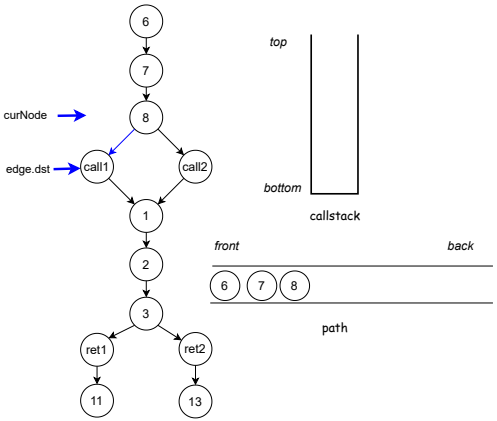
A feasible path from node 6 to node 11 on ICFG



1. Represent each DFS state as: ⟨current node, call stack⟩
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

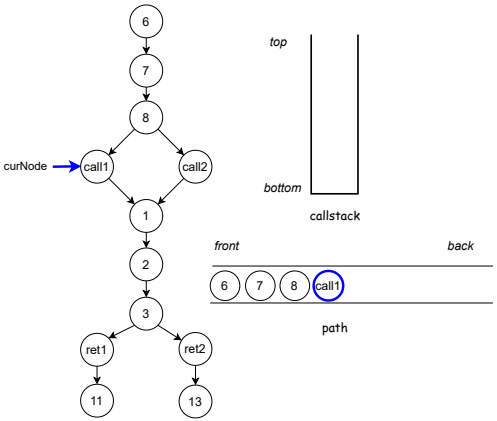
A feasible path from node 6 to node 11 on ICFG



1. Represent each DFS state as: ⟨current node, call stack⟩
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

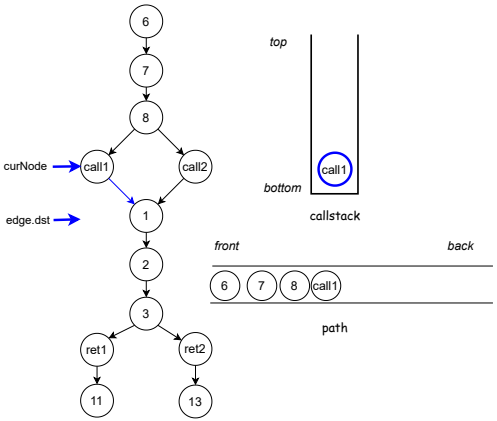
A feasible path from node 6 to node 11 on ICFG



1. Represent each DFS state as: ⟨current node, call stack⟩
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

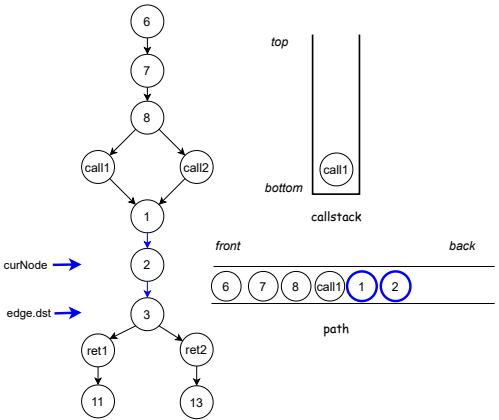
A feasible path from node 6 to node 11 on ICFG



1. Represent each DFS state as: $\langle$current node, call stack$\rangle$
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

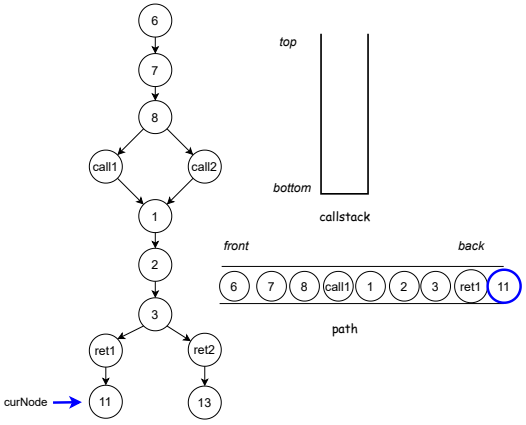
A feasible path from node 6 to node 11 on ICFG



1. Represent each DFS state as: ⟨current node, call stack⟩
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

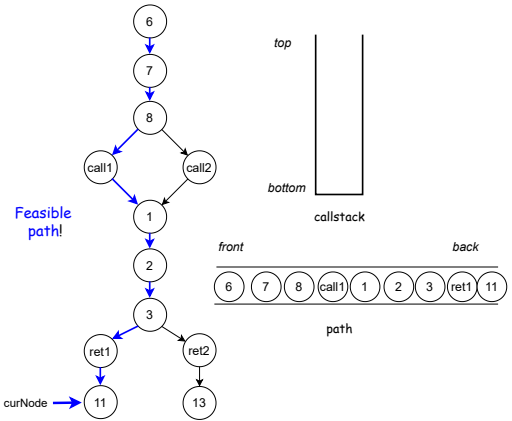
A feasible path from node 6 to node 11 on ICFG



1. Represent each DFS state as: ⟨current node, call stack⟩
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

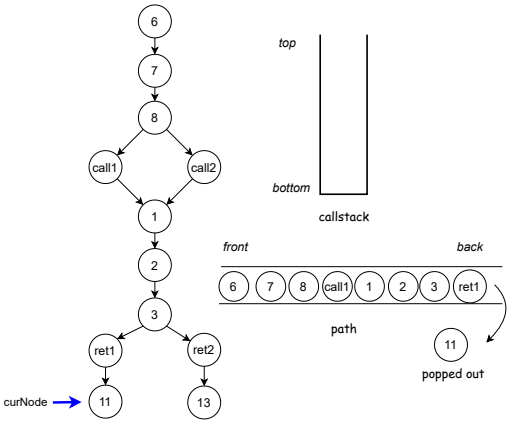
A feasible path from node 6 to node 11 on ICFG



1. Represent each DFS state as: ⟨current node, call stack⟩
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

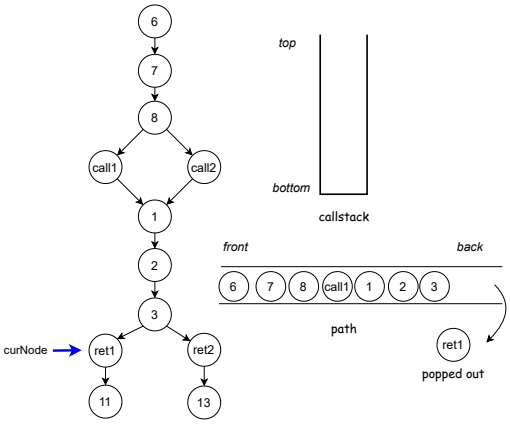
A feasible path from node 6 to node 11 on ICFG



1. Represent each DFS state as: ⟨current node, call stack⟩
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

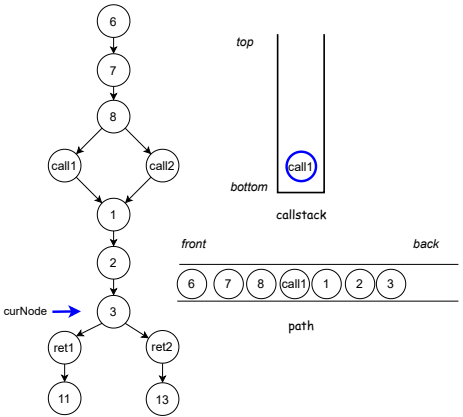
A feasible path from node 6 to node 11 on ICFG



1. Represent each DFS state as: $\langle$current node, call stack$\rangle$
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

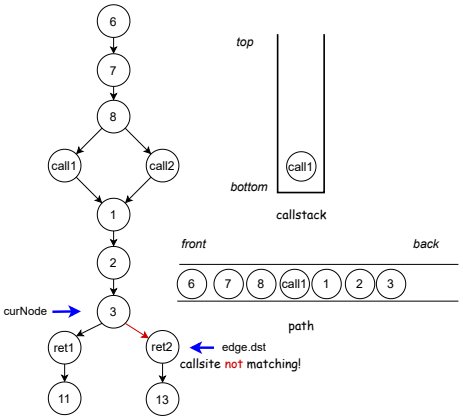
A feasible path from node 6 to node 11 on ICFG



1. Represent each DFS state as: ⟨current node, call stack⟩
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

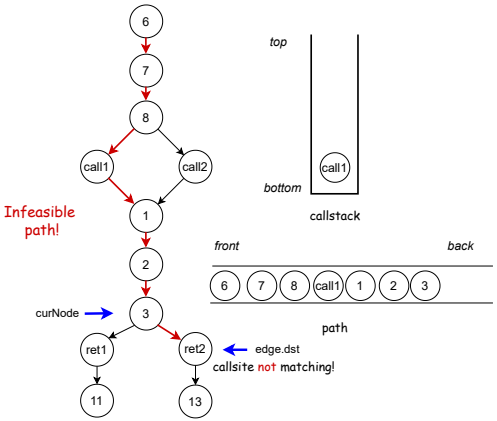
A feasible path from node 6 to node 11 on ICFG



1. Represent each DFS state as: ⟨current node, call stack⟩
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

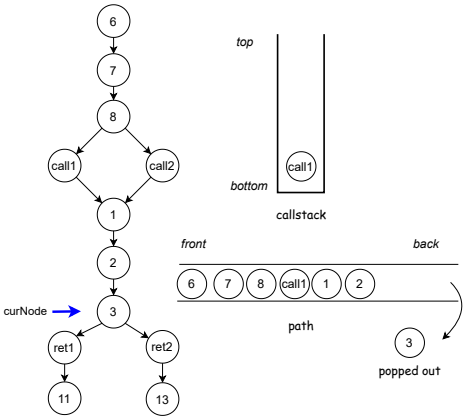
A feasible path from node 6 to node 11 on ICFG



1. Represent each DFS state as: ⟨current node, call stack⟩
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

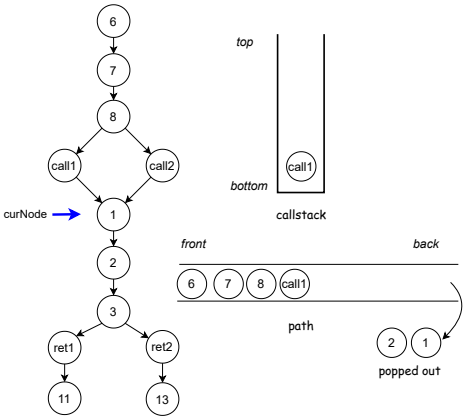
A feasible path from node 6 to node 11 on ICFG



1. Represent each DFS state as: ⟨current node, call stack⟩
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

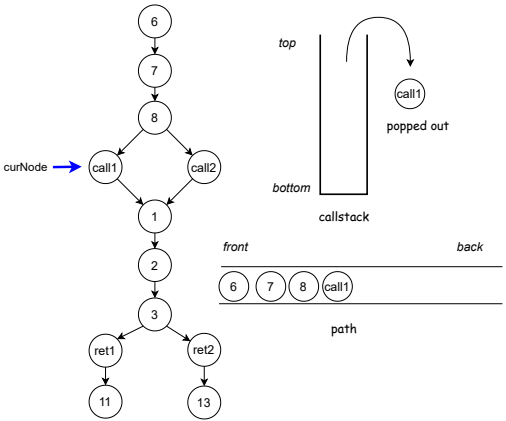
A feasible path from node 6 to node 11 on ICFG



1. Represent each DFS state as: $\langle$current node, call stack$\rangle$
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

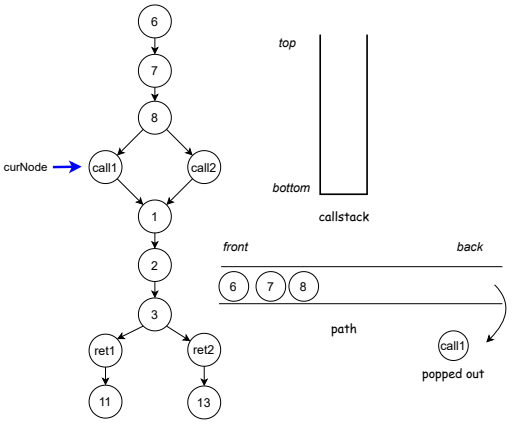
A feasible path from node 6 to node 11 on ICFG



1. Represent each DFS state as: ⟨current node, call stack⟩
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

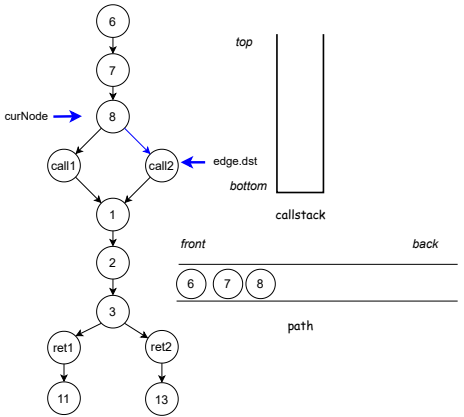
A feasible path from node 6 to node 11 on ICFG



1. Represent each DFS state as: ⟨current node, call stack⟩
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

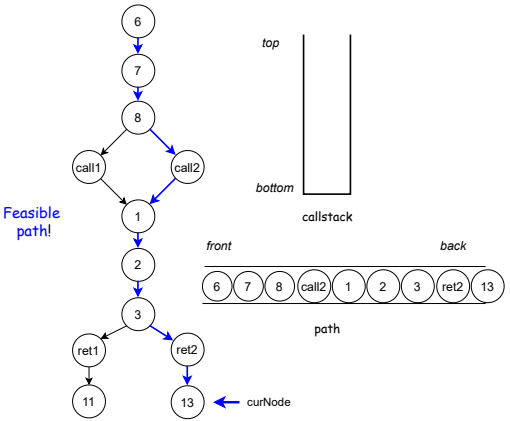
A feasible path from node 6 to node 13 on ICFG



1. Represent each DFS state as: ⟨current node, call stack⟩
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

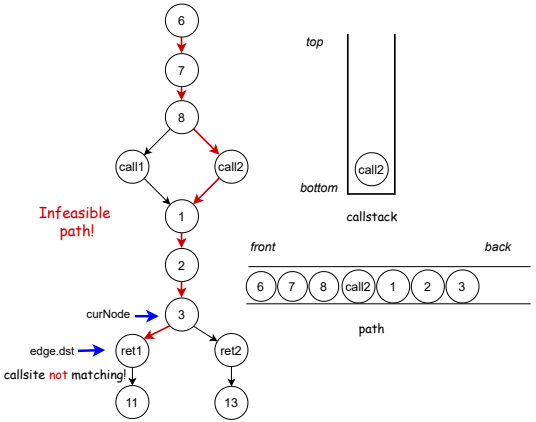
A feasible path from node 6 to node 13 on ICFG



1. Represent each DFS state as: $\langle$current node, call stack$\rangle$
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability

A feasible path from node 6 to node 13 on ICFG



1. Represent each DFS state as: ⟨current node, call stack⟩
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

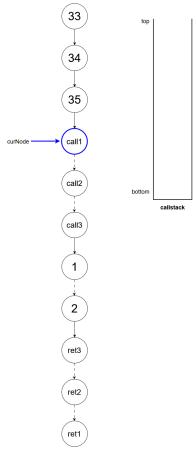
Nested functions

```
1  extern int source();
2  extern int sink(int);
3
4  int third(int x) {
5      return x + 1;
6  }
7
8  int second(int x) {
9      return third(x) + 1;
10 }
11
12 int first(int x) {
13     return second(x) + 1;
14 }
15
16 int main() {
17     int x = source();
18     int result = first(x);
19     sink(result);
20
21     return 0;
22 }
23
```

Nested functions

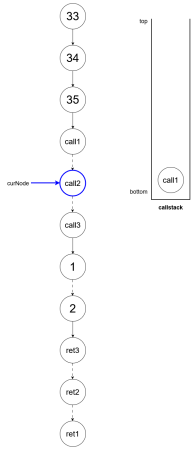
```c
extern int source();
extern int sink(int);

int third(int x) {
    return x + 1;
}

int second(int x) {
    return third(x) + 1;
}

int first(int x) {
    return second(x) + 1;
}

int main() {
    int x = source();
    int result = first(x);
    sink(result);

    return 0;
}

```

Context-Sensitive Control-Flow Reachability (2)



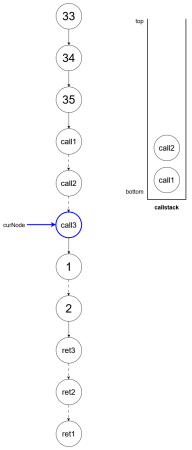
1. Represent each DFS state as: $\langle$current node, call stack$\rangle$
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability (2)



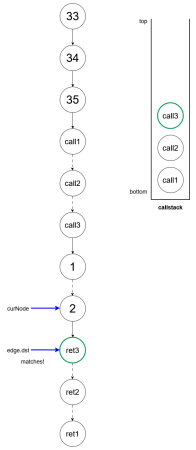
1. Represent each DFS state as: $\langle$current node, call stack$\rangle$
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability (2)



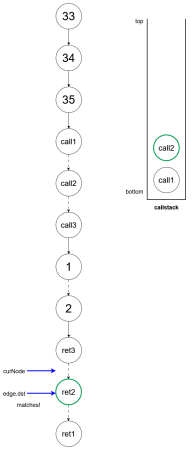
1. Represent each DFS state as: $\langle$current node, call stack$\rangle$
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability (2)



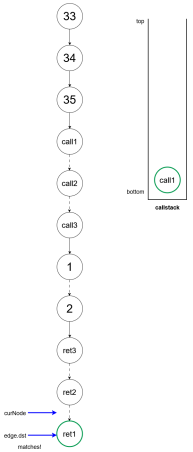
1. Represent each DFS state as: $\langle$current node, call stack$\rangle$
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability (2)



1. Represent each DFS state as: $\langle$current node, call stack$\rangle$
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability (2)



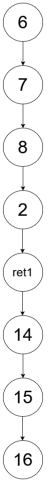
1. Represent each DFS state as: ⟨current node, call stack⟩
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Edgecase: source in middle of function

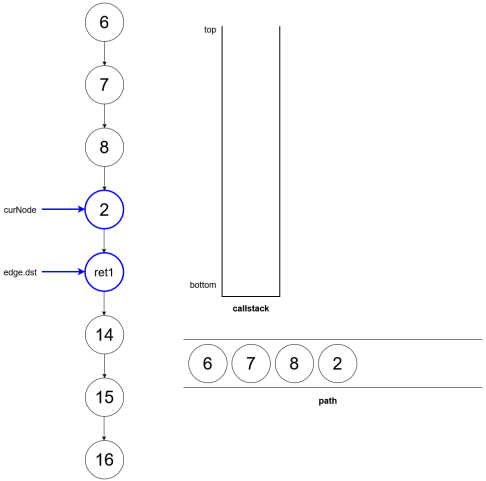
```
1   extern int source();
2   extern int sink(int);
3
4   int other() {
5       return source() + 1;
6   }
7
8   int main() {
9       int x = other();
10      sink(x);
11
12      return 0;
13  }
14
```

Edgecase: source in middle of function

```
1   extern int source();
2   extern int sink(int);
3
4   int other() {
5       return source() + 1;
6   }
7
8   int main() {
9       int x = other();
10      sink(x);
11
12      return 0;
13  }
14
```

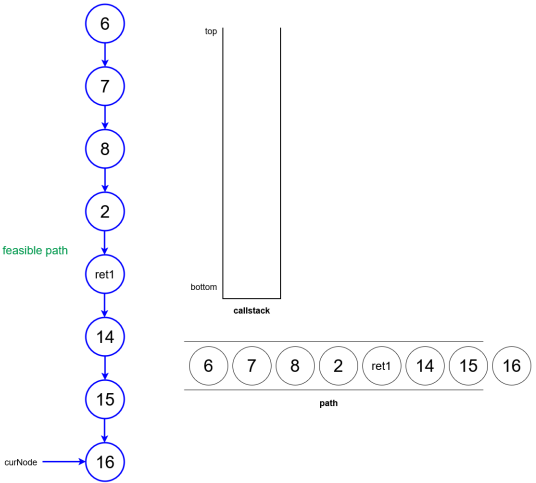


Context-Sensitive Control-Flow Reachability (3)



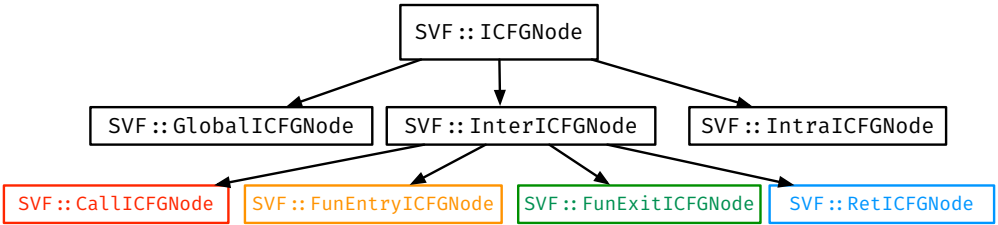
1. Represent each DFS state as: $\langle$current node, call stack$\rangle$
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

Context-Sensitive Control-Flow Reachability (3)



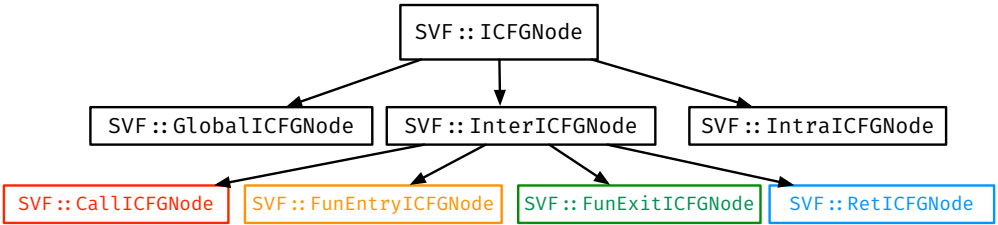
1. Represent each DFS state as: ⟨current node, call stack⟩
2. Start DFS from the current ICFG node.
3. If the current state is already on the DFS path, stop exploring.
4. Add the current node to the path.
5. If the current node is the sink, record the current path.
6. For each outgoing edge:
   - **Intra-procedural edge:** directly visit the target node.
   - **Call edge:** push the call site onto the stack and visit the callee.
   - **Return edge:**
     - **Non-empty-stack:** allow traversal only if it matches top callsite; then pop.
     - **Empty-stack:** allow traversal back to its caller return site.
7. Backtrack by restoring the path, call stack, and path-local visited state.

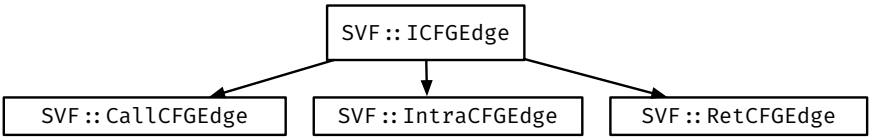
ICFG Node and Edge Classes



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.
  - `RetICFGNode* retNode = SVFUtil::cast<RetICFGNode>(ICFGNode);`
  - `CallCFGEdge* callEdge = SVFUtil::dyn_cast<CallCFGEdge>(ICFGEdge);`

“Casting” from Parent to Child in Python

- Python is a **dynamically typed language**:
  - No cast is required to call a method or access an attribute of an object.
  - A method can be called without declaring the object’s type in code, as long as the object is of the correct type at runtime.
- **Dynamic Typing $\neq$ No Discipline**:
  - **Adding type hints** is highly recommended to improve **readability** and **tooling support**.
  - This enables code completion, bug detection, and better maintainability.
- `isinstance()` and `typing.cast`:
  - Use `isinstance()` to narrow the type at runtime and assist IDE.
  - Use `typing.cast()` to explicitly annotate the expected type.
- Example: narrowing and annotating types for better IDE inference.
  - `if isinstance(node, RetICFGNode):` # IDE infers node is RetICFGNode
  - `call_edge = typing.cast(CallCFGEdge, edge)` # IDE infers call_edge is CallCFGEdge