CS6340 Project Proposal: Reachability Analysis using SVF Framework

Aniruddha Mysore and Nidhi Manjunath (Group 9)

January 10, 2024

Abstract

SVF [6] represents a static tool designed for scalable and accurate interprocedural dependence analysis on LLVM intermediate representations (IRs). It facilitates the iterative execution of both value-flow construction and pointer analysis, enhancing precision in both techniques. This precision is valuable for supporting various forms of program analysis. In this project, we will leverage SVF to generate interprocedural control-flow graph, to obtain a more comprehensive view of the program's execution flow, and also consider how different functions interact with each other. The ICFG provides the control instructions from the program entry node to the program exit node and provides multiple control flows among the whole program. Analysis of an ICFG can be used to detect path reachability between two nodes. We will use this graph to determine reachability i.e if there exists a path between two given nodes.

1 Introduction

In this project, we focus on the problem of reachability analysis. We propose to examine whether certain states or conditions can be reached from one node (the source) to another (the sink) during execution, given some test code. Reachability analysis is useful because it can help verify the correctness of programs and identify design issues if certain paths are unreachable. At a higher level, reachability analysis enables tasks such as assessing system vulnerabilities, understanding system behavior, optimizing the system, and making informed decisions to improve system performance and safety. In this project, we will use static tools to perform flow analysis. Static tools allow us to detect bugs and security vulnerabilities at early stages of development, even before programs are compiled to machine code or executed. This improves software quality and maintainability in the long run.

1.1 Static Analysis

Static analysis stands as a crucial method in software testing, aiming to discover and mitigate code issues prior to execution. Within the realm of static analysis, a key research challenge involves resolving program dependencies and value flows. By advancing the precision and scalability of static value-flow analysis, we can markedly improve the overall efficacy of static program analysis in identifying bugs, memory leaks, and various vulnerabilities.

1.2 Reachability

Graph reachability analysis involves determining whether there exists a path between two nodes in a graph. In a directed graph, a vertex u (sink) is reachable from another vertex v (source) if

a directed path exists from v to u. If no such path exists, node u is said to be unreachable from v. SVF [6] analyzes a program by taking LLVM IR as input and transforms LLVM instructions into an Inter procedural Control Flow Graph (ICFG), which can be used for further analysis tasks. Since the ICFG is a graph representation of the sequence of decisions taken during the execution of the program, we can use it to correctly identify source nodes and sink nodes, find all traversal paths, and establish whether the sink is reachable from the given source. With this project, we aim to build a reachability analysis tool that works with the ICFG generated by SVF.

```c
void src() {
    return;
}

void g() {
    return;
}

void sink() {
    return;
}

int main () {
    src();

    g();

    src();

    sink();
}
```

Figure 1: An example C program

1.3 Sparse Value Flow

Unlike traditional flow analysis, sparse analysis allows for analyzing large programs without sacrificing performance while maintaining precision to varying degrees, depending on the technique used. To avoid expensive propagation of data-flow facts across a program's control flow graph, sparse analysis is usually conducted in stages: a pre-analysis is first applied to over-approximate a program's def-use chains, which are then refined by performing data flow analysis sparsely, i.e., only along such pre-computed def-use relations. SVF is one such tool that leverages sparse analysis to conduct value flow analysis on LLVM Intermediate Representation (LLVM IR).

1.4 Call Graphs for LLVM IR

In the LLVM toolchain [3], frontends compile various source languages to LLVM IR, serving as a language-agnostic abstraction for LLVM optimizers. The LLVM backend then compiles the optimized LLVM IR to architecture-specific machine code. SVF analyzes a program by taking LLVM IR as input and transforms LLVM instructions into an Interprocedural Control Flow Graph (ICFG), which can be used for further analysis tasks. Since the ICFG is a graph representation of the sequence of decisions taken during the execution of the program, we can use it to correctly identify source nodes and sink nodes, find all traversal paths, and establish whether the sink is reachable

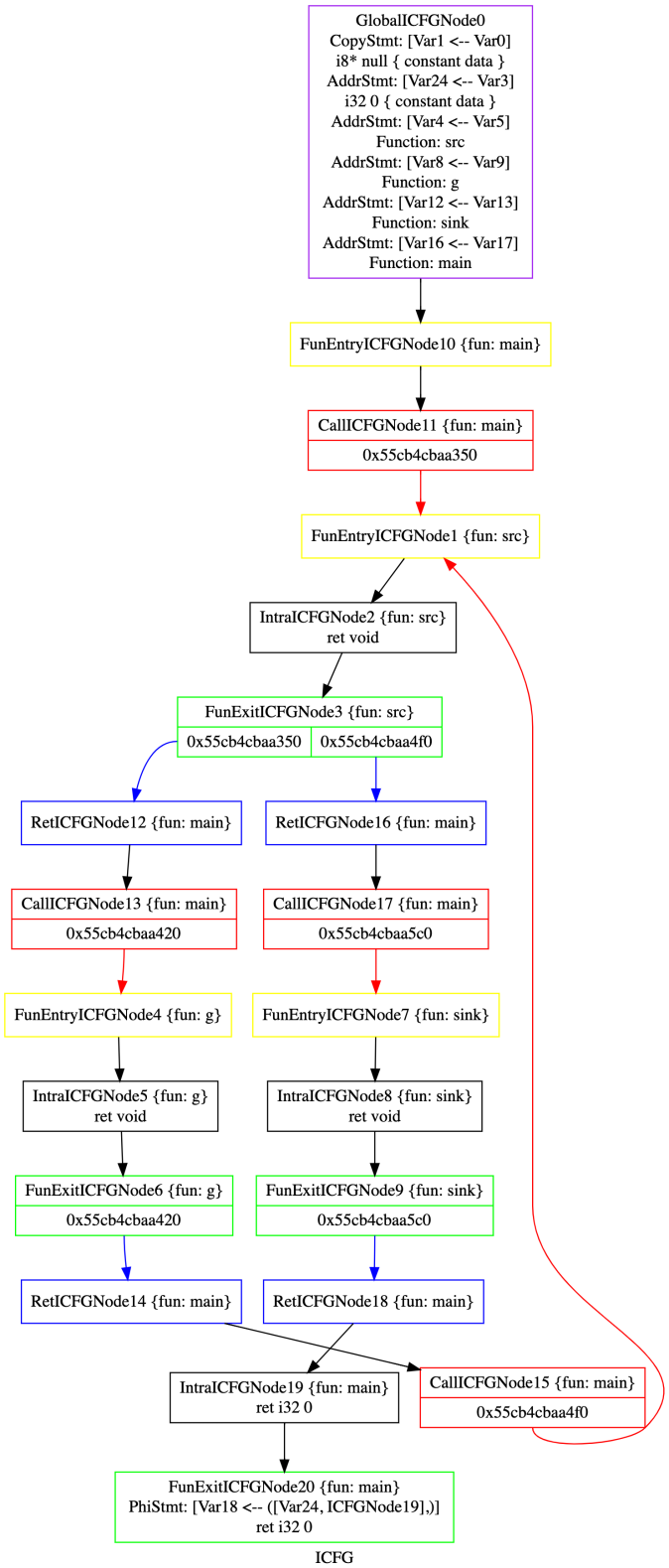

Figure 2: ICFG of the example C program

from the given source. Figure 2 depicts an ICFG that SVF has generated for the example C program in Figure 1 . With this project, we aim to build a reachability analysis tool that works with the ICFG generated by SVF.

2 Related Work

We use the concepts defined in [4] to understand generation of control flow graphs. In CFG, each function call statement is represented using two nodes: a call node representing the connection from the caller to the entry of f, and an after-call node where execution resumes after returning from the exit of f. CFGs can be constructed such that there is always a unique entry node and a unique exit node for each function i.e source and sink functions in our project. The connection between the call node and its after-call node is represented by a call edge, needed for propagating local variables of the caller. We can now various graph traversal algorithms to perform flow analysis on the ICFG.

2.0.1 Interprocedural Dataflow analysis framework

In the past we have seen data flow analysis methods by converting them into graph reachability problems using special polynomial time algorithms. This includes problems in which the set of dataflow facts D is a finite set and the dataflow functions distribute over the meet operator also called as IDFS problems [5]. However, for the simplicity of this project we will use DFS to implement reachability.

As we know, Depth-First Search (DFS) is a well known algorithm used for traversing or searching through graphs. The algorithm starts at a source node and explores as far as possible along each branch before backtracking. This DFS implementation uses recursion to explore all reachable nodes from a given starting node. The visited array is used to keep track of nodes that have already been visited to avoid infinite loops in the case of cyclic graphs [2].

3 Solution Design

At a high-level, our solution consists of the following series of steps:

1. Read the LLVM bitcode and generate the ICFG - This step relies completely on a sequence of SVF library procedure calls
2. Identify sources and sinks in the ICFG - this is again largely handled by SVF library functions
3. Invoke the traversal algorithm which uses a depth-first-search (DFS) to find reachability paths
4. Print the paths found, if any - including any cycles

Of these, the traversal algorithm merits further discussion and is the subject of the next sections of this report.

4 Vanilla DFS and its challenges

Algorithm 1 Context Sensitive Path Analysis using Depth-First-Search

```
1: function DFS(visited, path, callstack, src, dst)
2:     visited.insert(src)
3:     path.push_back(src)
4:     if src == sink then
5:         print_path()
6:     for edge e ∈ src.outEdges() do
7:         if e ∉ visited then
8:             if e.isIntraCFGEdge() then
9:                 DFS(visited, path, callstack, e.dst, dst)
10:            else if e.isCallCFGEdge() then
11:                callstack.push(e.getCallSite())
12:                DFS(visited, path, callstack, e.dst, dst)
13:            else if e.isRetCFGEdge() then
14:                if ! callstack.empty() && callstack.top() == e.getCallSite() then
15:                    callstack.pop()
16:                    DFS(visited, path, callstack, e.dst, dst)
17:    visited.erase(src)
18:    path.pop_back()
19: function TRAVERSE
20:    visited: set⟨ICFGNode⟩
21:    path: vector⟨ICFGNode⟩
22:    callstack: stack⟨SVFInstruction⟩
23:    DFS(visited, path, callstack, src, dst)
       return path
```

Algorithm 2 Depth-First-Search with Stored Cycles

```
   paths: set⟨vector⟨ICFGNode⟩⟩
   cycle_found: set⟨vector⟨ICFGNode⟩⟩
1: function DFS(visited, path, callstack, src, dst)
2:     visited.insert(src)
3:     path.push_back(src)
4:     if src == sink then
5:         AddToPathSet(paths, path)
6:     for edge e ∈ src.outEdges() do
7:         if e ∉ visited then
8:             ...                                   ▷ Same as original DFS algorithm
9:         else                                      ▷ We have detected a cycle
10:            cycle = [e.destNode()...path.end()]
11:            AddToPathSet(cycles_found, cycle)
12:    visited.erase(src)
13:    path.pop_back()
14: function TRAVERSE
15:    visited: set⟨ICFGNode⟩
16:    callstack: stack⟨SVFInstruction⟩
17:    for src ∈ sources do
18:        for dst ∈ sinks do
19:            dfs_path: vector⟨ICFGNode⟩
20:            DFS(visited, dfs_path, callstack, src, dst)
       return paths
```

A simple, "vanilla" DFS algorithm is stated in Algorithm 1 - this was provided by authors of SVF in their Teaching-SVF course material [1]. This algorithm can determine if there is at-least one

reachable path between a source and sink node. However, in this project we aim to find ALL the reachability paths present between the source and sink, including cycles, if any. We found that with some simple additions, the vanilla algorithm can be extended to find cycles in the reachability path. Here are the results of our algorithm on the given test cases:

```
test1.bc
Reachable
1-->2-->3-->17-->20-->22-->24-->25-->13-->14-->15-->18-->4
Cycle[1-->2-->3-->17-->20-->22-->24-->25-->13-->14-->15-->16-->1]
-->2-->3-->17-->20-->22-->24-->25-->13-->14-->15-->18-->4

test2.bc
Reachable
4-->5-->1

test3.bc
Reachable
4-->5-->6-->7-->8-->9-->17-->18-->19-->21-->1

intertest.bc
Reachable
1-->2-->3-->4-->9-->11
Cycle[1-->2-->3-->4-->5-->7-->17-->18-->1]-->2-->3-->4-->9-->11
```

5 Cycle-storing DFS

Algorithm 2 is our modification of vanilla DFS to include cycle determination. The cycle detection logic is quite simple - in essence, we know that during DFS if we encounter a node twice we are in a cycle. In this case, we save the portion of the traversal path that is a cycle in a candidate cycles list. The portion of the path between visit 1 and visit 2 of the same node is a cyclical path.

5.1 Algorithm Design Progression

We developed our method incrementally, to solve for reachability on all the ICFG examples we observed from the provided tests. As mentioned before, vanilla DFS was able to determine whether at-least one path existed between source and sinks.

5.1.1 Multiple sources and sinks

The test cases included in the docker container all used single invocations of the source function leading to our initial implementation incorrectly assuming there would be a single source and sink to invoke DFS on. The ICFG in Fig 2 was a simple example where this assumption did not hold, and we changed the traversal logic to invoke DFS on every source-sink pair (lines 17 - 20 in Alg. 2).

5.1.2 Cycles

There were several cycle determination options we explored. One strategy was to run DFS a second time from edges that did not yield a path to sink on the first run. The idea was to check if there was a path to any node in the reachability paths found in the first pass. We chose to not implement this strategy since we realized a simpler method (albeit more expensive in storage costs) would be

to just store cycles we encounter. We chose this trade-off since the project's performance evaluation only measures time. Our algorithm is poor in space-utilization while maintaining $O(log(N))$ time complexity of DFS.

5.1.3 Nested cycles

We have not evaluated our implementation on ICFGs with nested cycles - none of the provided tests include nested cycles and we were limited by time constraints on ensuring the algorithm performed reasonably well on the tests already at hand.

6 Conclusion

In this project we have implemented a time-efficient DFS algorithm to perform reachability analysis on a given ICFG graph. Given a set of source and sink sources, we determine all possible paths from a source to a sink including the detection of cycles. Considering the limitations of the time constraints we hope to extend our analysis to nested cycles and improve space utilization in the future.

References

[1] *DFS algorithm for context-sensitive reachability*. Github. URL: https://github.com/SVF-tools/Teaching-Software-Analysis/blob/slides/slides/Assign-2.pdf.

[2] S. Even and G. Even. *Graph Algorithms*. Cambridge University Press, 2011. ISBN: 9781139504157. URL: https://books.google.com/books?id=m3QTSMYm5rkC.

[3] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: San Jose, CA, USA, 2004, pp. 75–88.

[4] Anders Møller and Michael I. Schwartzbach. "Static Program Analysis". In: *Encyclopedia of Cryptography and Security*. 2011. URL: https://api.semanticscholar.org/CorpusID:46011248.

[5] Thomas Reps, Susan Horwitz, and Mooly Sagiv. "Precise Interprocedural Dataflow Analysis via Graph Reachability". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 49–61. ISBN: 0897916921. DOI: 10.1145/199448.199462. URL: https://doi.org/10.1145/199448.199462.

[6] Yulei Sui and Jingling Xue. "SVF: interprocedural static value-flow analysis in LLVM". In: *Proceedings of the 25th international conference on compiler construction*. ACM. 2016, pp. 265–266.