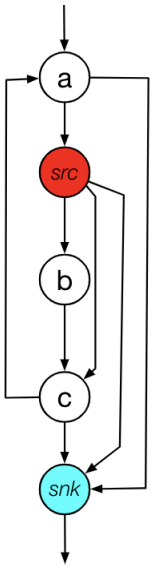
CS 6340: Software Analysis and Test
SVF Project

1 Project Description

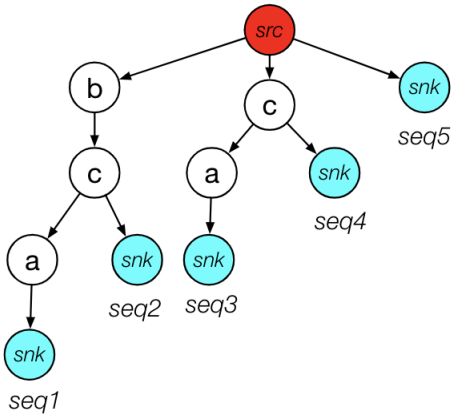
In this project, you will get familiar with the SVF framework and build a reachability analysis tool for LLVM IR. SVF is a static tool that enables scalable and precise interprocedural dependence analysis on LLVM IRs. Your submitted analysis tool should read LLVM IR files and decide (statically) whether the `sink`() function entry block is reachable from the `src`() function entry block.

1.1 Task Description

An interprocedural control-flow graph (ICFG) describes the control flow of a target program. The ICFG represents 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.



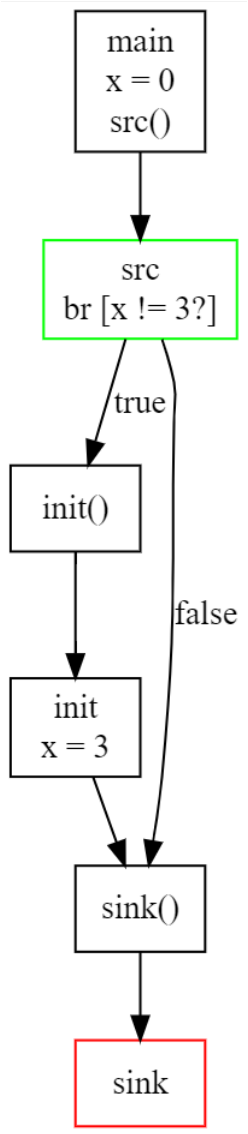
(a) Partial ICFG



(b) Traversal procedure

Your program should perform the previously mentioned reachability analysis. Some caveats:

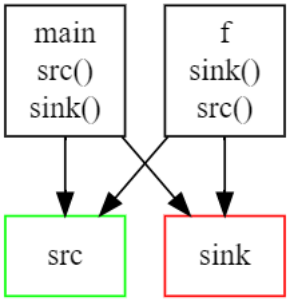
1. The analysis should be **Path-Insensitive & Context-Sensitive**. **Be sure to consider how Context-Sensitivity affects your results.**
2. No data flow analysis is necessary. As such, it is okay to traverse paths that are technically infeasible:



(a) $src \xrightarrow{calls} sink$ (infeasible)

(b) $src \xrightarrow{calls} init \xrightarrow{returns} src \xrightarrow{calls} sink$

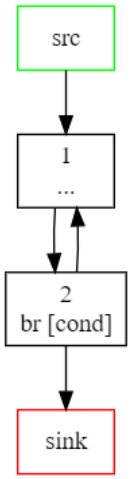
3. There may be multiple call sites of src and sink.



   (a) $src \xrightarrow{returns} main \xrightarrow{calls} sink$

4. Your analysis should report all cycles within the traversal paths. The connections into the cycle subpath must be correct (the element before the cycle must connect to the first element of the cycle and the element after the cycle must connect to the last element of the cycle). Reporting any of the multiple cyclic representations is fine as long as each cycle is represented in the output. Some additional notes:

   - There will be no nested cycles.
   - Each cycle will have a defining br instruction that will not be shared with another cycle.



   (a) $src \rightarrow 1 \rightarrow 2 \rightarrow sink$ (acyclic path)

   (b) $src \rightarrow Cycle[1 \rightarrow 2] \rightarrow sink$ (cycle path)

   (c) $src \rightarrow 1 \rightarrow Cycle[2 \rightarrow 1] \rightarrow 2 \rightarrow sink$ (alternate cycle path)

5. You do not need to handle indirect function calls

6. You do not need to handle recursion in callstacks (A function will not call itself anywhere in its body or its sub-functions).

7. You may end analysis at the `sink`() entry node. Paths going through this and eventually reaching the `sink`() entry node again do not need to be reported.

The ICFG can be generated by the SVF framework; SVF also provides the functionality to identify instructions represented by the nodes in the ICFG.

One goal in this project is also to learn how to utilize a real-world static analysis framework to implement analysis tools. Please start early to get familiar with the SVF framework in this project! Reference for the SVF framework is available at https://github.com/SVF-tools/SVF/wiki.

1.2 Project Setup

1.2.1 Docker

To avoid configuration issues, we will use Docker in this project. We provide a guide to use Docker and our grading environment will be the same as described in the guide. Here are the instructions to set up, build, and test your project.

First, you need to install Docker on your local machine. Here is the link to get started with Docker: https://www.docker.com/get-started.

Once you have Docker installed on your machine, you can pull the provided Docker image from Docker Hub and then work on it. If you want to know more about Docker please consult the official documentation.

1. First you can pull the image from Docker Hub.

   ```
   docker pull mdavis438/svfproject:latest
   ```

2. Then you can start a Docker container from the provided Docker image using the following command.

   ```
   docker run -it --name svfproject mdavis438/svfproject:latest /bin/bash
   ```

   If you want to come back later you can exit the bash session and start the stopped container again using the following command.

   ```
   docker start -ai svfproject
   ```

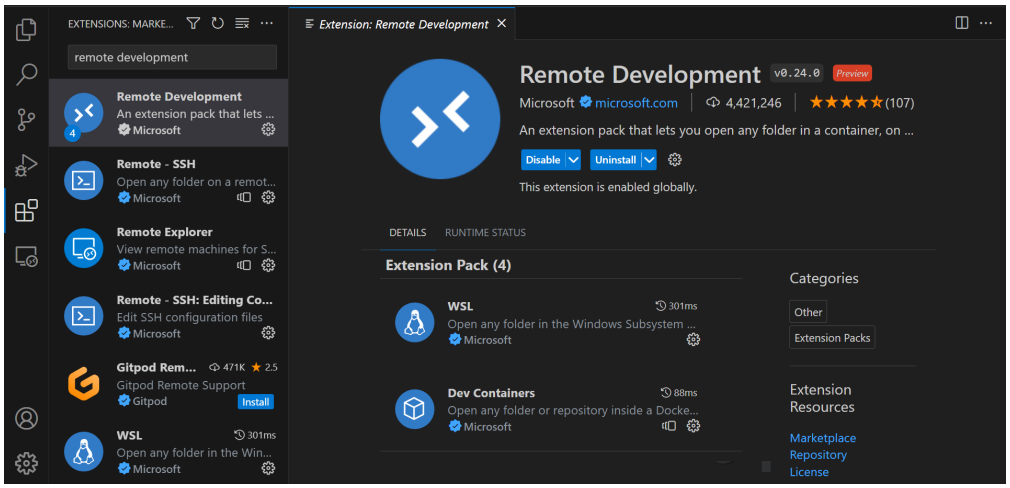
Please keep in mind that docker containers will not save changes when closed! Either:

1. **Save the container to a docker image**
2. **Mount the project directory to your local computer**
   (add `-v host/directory/path:/home/project/src` to your docker run command)
3. **Save the file copy to your local machine before closing the container**
4. **Save your code on a private repository**

1.2.2 VSCode (optional)

This section provides a guide for if you want to use VScode dev container connections.

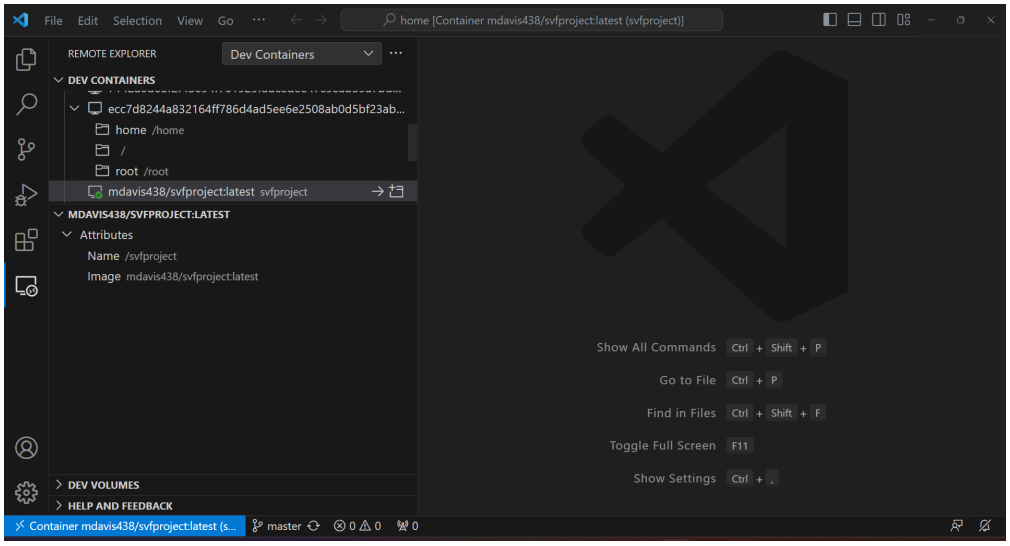
1. Install VSCode Editor: https://code.visualstudio.com/download
2. Install Remote Development Extension in VSCode



3. This alternative of the docker start command will start the container in the background without an interactive terminal:
   `docker start svfproject`
4. Attach to the docker container.

   Open the Remote Explorer view from the left panel. Select "Dev Containers" in the drop-down at the top of the sidebar. This should list all of the docker containers available.

   Choose your project container ('svfproject'). VSCode will take some time to setup the connection and then open the container environment.



More info on dev containers can be found here. VSCode provides this tutorial but be sure to replace the example container instructions with our docker container.

1.2.3 Project Setup

1. To complete this project, you only need to modify the following file inside the container.

   `~/project/src/project.cpp`

2. After completing the code in this file you can build the project by first going to the directory "`~/project/`" and then running the following commands

   ```
   source ./env.sh
   ./build.sh
   ```

3. This will generate the executable `~/project/bin/svf-project`. The project can be run using the command

   ```
   ./test.sh [path/to/test.bc]
   ```

   and will generate the `out.txt` output file

2 Provided Code

The provided code and test case information is in the Docker container on Docker Hub. In the Docker container we have provided the framework to build the project based on the dependence of SVF libraries. In the placeholder `project.cpp` file, you need to perform the reachability analysis and print the result to `stdout`. There are also many test cases provided in `~/project/test_cases`.

3 Grading

There are in total 100 points for this project.

3.1 Correct Implementations (60 Points)

Several test cases are given to you to test the correctness of your project under the `~/project/test_cases` directory. You can run the script `test.sh` and compare your output to the README file in the `~/project/test_cases` directory to check the correctness of your implementation. For grading, there will be hidden test cases for the project. **The provided example test cases DO NOT fully cover the possibilities of the problem which may appear in the hidden test cases. It is highly recommended to thoroughly test your implementation by writing your own tests.** We will use the build.sh script to build your project, and there will be no optimizations or transformations performed on the IR you will be graded on (no loop unrolling etc.)

For each input test case, your program is expected to print out the following outputs:

- **First line - 40 Points:** The result whether the function entry block of `src`() can reach the function entry block of `sink`(). If it is reachable, your program should print "Reachable", otherwise print "Unreachable". Print the EXACT text of "Reachable" or "Unreachable". You will receive the proportion of 40 Points corresponding to the proportion of matching reachability results in the hidden testcases.
- **Following lines - 20 Points:** If there are n traversal paths between the `src` and `sink`, your project should also print n more lines. Each line is a path from `src`() to `sink`(). The path should report the ICFG Node IDs in order of the path, starting with the func entry node of `src`() and ending with the func entry node of `sink`() (specifically the nodes returned via `FunEntryICFGNode* getFunEntryICFGNode(const SVFFunction* fun);`).

  Supposing that node 1 is the `src` node and node 5 is the `sink` node, a path could look like: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$.

  In the instance that the path or a portion is cyclic, annotate the cyclic region in the format described in 1.1. Example: $1 \rightarrow Cycle[2 \rightarrow 3 \rightarrow 4] \rightarrow 5$. The entry/exit nodes from the cycle must be correct, regardless if this duplicates nodes i.e. $1 \rightarrow Cycle[2 \rightarrow 3 \rightarrow 4 \rightarrow 2] \rightarrow 5$. In general, if there is a cycle, make sure it is reported in at least one path. Some cycles can have multiple representations, but as long as one of the variations is reported that is sufficient.

  Correct program traversal paths will award 20 Points.

3.2 Performance Evaluation (10 Points)

If your program can finish running each test case in 30 seconds, you will get 10 Points. The test cases are incredibly small (max of ≈300 nodes). If your analysis takes over 30 seconds for a test this is probably an extreme time complexity problem with your implementation.

3.3 Design (30 Points)

In your final report, please briefly describe the following:

1. Details about how you perform the reachability analysis.
2. Details about how you perform the cycle detection.
3. Details about how you made the analysis context-sensitive and how this affects the results.
4. Any known outstanding bugs or deficiencies that were unable to be resolved before submission.
5. "Future work"-style ways that the analysis could be improved or extended.

4 Submission

As stated before, to facilitate the grading, you should not modify other files besides `project.cpp` in the project directory! On Canvas, submit a single ZIP file that contains:

- The `project.cpp` file
- The `design.pdf` file described in Section 3.3.
- Any test cases you added (these will not be graded).

We will only use the `project.cpp` file to test the correctness of your implementation. If you modify other files, even if your code runs correctly on your local machine, you will still get points deducted if it cannot run properly in our grading environment.

5 Collaboration

We will award identical grades to each member of a given project team, unless members of the team directly register a formal complaint. We assume that the work submitted by each team is their work solely. Any questions about design and implementation should conducted solely with the instructors through office hours, email, etc. Under no condition is it acceptable to use code written by another team, or obtained from any other source. As part of the standard grading process, each submitted solution will automatically be checked for similarity with other submitted solutions and with other known implementations.