CS510 Project 2: Memory Leak Detection

October 6, 2023

Project Description

Memory Leak is memory allocations without releases. In this project, we conduct a static analysis to detect memory leak bugs in program. For this, we use LLVM-IRs which are popularly used in program analysis.

Memory Leak Albeit there are a myriad of interfaces to (de-)allocate memory, we stick to `malloc()` and `free()`. For clarity, we define that the program has a memory leak if and only if there is a path that incurs any of heap space remaining when it exits.

```
  ...
1 for (...){
2   x = malloc();
3   ...
4   if (...)
5     break;
6   ...
7   free(x);
8 }
  ...
```

This example contains a memory leak along with path; 1 → 2 → 4 → 5 → 8.

To Perform A Static Analysis

A static analysis algorithm must satisfy two properties; soundness and termination. That is, the tool must complete and not incur false positives/negatives.

Your tool will be broken into two phases; (i) path extraction and (ii) leak analysis.

Path Extraction. In this phase, you will be extracting all paths along the program flow. Here, the paths are not limited to be feasible, that is, you do not need to analyze the branch conditions (which may entail complicated sat solvers, if being considered). Let assume, for any conditional jumps (i.e., if-else, loop), the program can flow toward any directions. For termination of your analysis, you need to limit the number of expansions for loop. Due to characteristic of leak analysis, we can limit the loop executions at most a couple of times in our program semantics (please read assumptions). Consider the following example;

```
1  x = malloc();
2  while (d < 5){
3     d += 1
4     if (c > 0)
5         free(x);
6     y = malloc()
7     if (d > 0)
8         free(y);
9     c += 1
10  }
11  free(y);
```

Followings are paths to extract;

loop skip:

1: 1 → 2 → 10 → 11

loop once:

2: 1 → 2 → 3 → 4 → 6 → 7 → 9 → 10 → 11

3: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 9 → 10 → 11

4: 1 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → 10 → 11

5: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11

loop twice:

6: 1 → 2 → 3 → 4 → 6 → 7 → 9 → 10 → 2 → 3 → 4 → 5 → 6 → 7 → 9 → 10 → 11

7: 1 → 2 → 3 → 4 → 6 → 7 → 9 → 10 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → 10 → 11

8: 1 → 2 → 3 → 4 → 6 → 7 → 9 → 10 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11

9: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 9 → 10 → 2 → 3 → 4 → 6 → 7 → 9 → 10 → 11

10: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 9 → 10→ 2 → 3 → 4 → 6 → 7 → 8 → 9 → 10 → 11

11: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 9 → 10 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11

12: 1 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → 10 → 2 → 3 → 4 → 6 → 7 → 9 → 10 → 11

13: 1 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → 10 → 2 → 3 → 4 → 5 → 6 → 7 → 9 → 10 → 11

14: 1 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → 10 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11

15: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 2 → 3 → 4 → 6 → 7 → 9 → 10 → 11

16: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 2 → 3 → 4 → 5 → 6 → 7 → 9 → 10 → 11

17: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → 10 → 11

18: 1 → 2 → 3 → 4 → 6 → 7 → 9 → 10 → 2 → 3 → 4 → 6 → 7 → 9 → 10 → 11

19: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 9 → 10 → 2 → 3 → 4 → 5 → 6 → 7 → 9 → 10 → 11

20: 1 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → 10 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → 10 → 11

21: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11

If there are multiple loops, intuitively we can use a set-multiplication. Look at the following example;

```
1 x = malloc();
2 y = malloc();
3 while (d < 5){
4     free(x);
5 }
6  while (c < 5){
7     free(y);
8 }
```

We have $3 \times 3 = 9$ paths;

1: $1 \to 2 \to 3 \to 5 \to 6 \to 8$

2: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 8$

3: $1 \to 2 \to 3 \to 4 \to 5 \to 3 \to 4 \to 5 \to 6 \to 8$

4: $1 \to 2 \to 3 \to 5 \to 6 \to 7 \to 8$

5: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8$

6: $1 \to 2 \to 3 \to 4 \to 5 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8$

7: $1 \to 2 \to 3 \to 5 \to 6 \to 7 \to 8 \to 6 \to 7 \to 8$

8: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 6 \to 7 \to 8$

9: $1 \to 2 \to 3 \to 4 \to 5 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 6 \to 7 \to 8$

Leak Analysis. Per each path, we find memory allocations without free. The tool should raise memory leak alert, if exists, at the end of program (i.e., by each end of path). You may need to use tainting-analysis technique upon load/store as Project 1 (no worries, it will be easier than Project 1 thanks to simplicity of LLVM-IRs). You may also need to trace function calls (i.e., `malloc()`s and `free()`s). Note that a memory leak can also occur due to overwriting (`x = malloc()` $\to$ `x = malloc()` $\to$ `free(x)`). Double `free()`-ing upon a memory is out of our scope. You also need to trace struct type variable. For detailed scope, refer to the given testcases.

Output Format Output all paths that may incur memory leak. Default file descriptor is provided with the starter code. Use line numbers to represent paths. You may neglect tivial lines (e.g., ' {' or '}'). As in the previous homework, if represented path is reasonably recognizable, it will be OK. We will manually grade your output tolerating minor whitespaces/newline differences. Orders of paths do not count.

Here's example of execution;

```
1 #include <stdioh.h>
2
3 char c, d;
4 char * x;
5 char * y;
6
7 int main(){
8      c =getc(stdin);
9      d =getc(stdin);
10
11     while (d < 50){
12        x = malloc();
13        if (c > 15)
14           break;
15
16        free(x)
17        y = malloc();
18        }
19  free(y);
20 }
```

We expect the output as;
1: 8 → 9 → 11 → 12 → 13 → 14 → 19
2: 8 → 9 → 11 → 12 → 13 → 16 → 17 → 11 → 12 → 13 → 16 → 17 → 19

Based on your implementation, the second path can be;
2: 8 → 9 → 11 → 12 → 13 → 16 → 17 → 11 → 12 → 13 → 16 → 17 → **11** → 19
It is also OK. Also, you may or may not include 1 → 2 → 3 → 4 → 5 → 6 → 7, 10, 15, 18, 20 since they are trivial. Also you can print '->' instead of '→' due to ascii characters short. Each path is placed in new line. As we will manual-grade submissions, minor format differences will be tolerated. For this project, you can print results to console, using `outs()` (please refer to the starter code). Also, please comment out or remove all print functions for any debug messages when submitting your work.

Assumptions

- The target program is written in C.
- The target program includes '`main()`' and that is the portion of being analyzed.
- The target program will only use global variables.
- The target program may use '`char`', '`char *`', '`int`', '`int *`', '`struct`', '`struct *`'.

- The target program will not use ’`array`’, ’`union`’.
- The target program will not perform arithmetic operations onto pointer variables (e.g., you do not expect **char *p = malloc(); p++;**).
- The target program may use `for` or `while` loop and `if-else` statements but not include `switch` and `cases`.
- The target program may use ’`continue`’ or ’`break`’ in loop.
- The target program wiil not use function calls (except for ’`malloc()`’ and ’`free()`’).

LLVM guidance The workplace environment (i.e., the mc18 server) includes a preinstalled LLVM. First, your LeakDetector.cpp must be compiled and ran as a module pass with an LLVM. Use following command;

```
gcc -o LeakDetector.so -shared -fPIC LeakDetector.cpp
```

If you successfully build a LeakDetector.so, you can start analyzing the target source code. In order to perform static analysis, tranfrom a `.c` file to LLVM LRs;

```
clang -O0 -g -S -emit-llvm xxx.c -o xxx.ll
```

Note that xxx should be replaced with your file names (e.g., `test1` - `test10` in testcases). Also, in order to extract line numbers from the source code, we need to use `-g` option. Finally, once you get `xxx.ll` file, you can run your LLVM Module Pass (i.e., `LeakDetector`).

```
opt -enable-new-pm=0 -load LeakDetector.so -ldetector < xxx.ll
```

We use the legacy `ModulePass` APIs which have been updated in the recent versions, please use `-enable-new-pm=0` option to avoid conflicts.

The starter code includes your `ModulePass` registration. Therefore, your `LeakDetector` Module will execute automatically. Here, `fun.front()` represents the first basic block in a target code. Iterating over the basic block will let you visit each instruction from that BB. Also, from the last instruction in a BB (i.e., branch instruction), you can extract successors (i.e., possible movable BBs) using LLVM APIs (i.e., `getNumSuccessors()`, `getSuccessor()`). Therefore, your path extraction algorithm resembles tree-travesal algorithm, however you need to deal with cycle handlings (i.e., how many times you allow visiting the same BB and how?). You can represent a path as list of `Instruction`s or `BasicBlock`s.

Data provenance analysis is similar to that of the project 1. First, follow `LoadInst` and `StoreInst`. You may also need to follow `GetElementPtrInst` type for pointer operations (i.e., `struct *`). As any other analysis, the designing step it that you look at its IR translation per each different `c` statement. You can also access each operational arguments in a single IR using `getOperand(0)`, `getOperand(1)`.

Grading

Due Date 11/3, 11:59PM EST

Submit your `LeakDetector.cpp` on Brightspace by 11/3, 11:59PM. Your grade will be based on the correctness (i.e., termination and soundness) of your implementation. If your program does not run or goes to an infinite loop (1 minute cap per each run), no score is given. False positives/negatives will impose point deductions.