School of Innovation,
Design and Engineering

Module 4: Dynamic Program Analysis

This document provides the laboratory instructions for the part of Module 4 that deals with dynamic program analysis. The objective of the lab is to give hands-on experience with the much used tool environment **Valgrind**, and the tool **memcheck**, for dynamic analysis.

VALGRIND/MEMCHECK HANDS-ON EXERCISE

Also this exercise will take place at MDH, during one of the campus days. We have pre-installed Valgrind with memcheck on the lab computers. In case you want to try out Valgrind yourself, and possibly do some of the assignments by yourself ahead of the campus day, you can download Valgrind and install on your own machine provided that you run linux or some other unix version. See Section 6 below.

1. Starting a console window in linux to run Valgrind

Since Valgrind does not run under Windows, we have installed a virtual machine (Virtualbox) running Ubuntu linux. Start it by double-clicking the "Ubuntu linux" icon on the desktop. Ubuntu linux will start. To the left, you will have a bar with icons. Click on the rectangular one displaying ">_" (telling "Terminal" when hovering over it with the cursor). This will open a terminal console, with a command line, positioned at the directory `/home/mdh`.

2. Preparing and analyzing a first example

Go to subdirectory "code-examples" (type the command "`cd code-examples`" in the console window.) This directory is a copy of the folder with the same name under Windows, containing the same example C codes.

Compile insertsort.c with the command "`gcc -g -O0 insertsort.c`". This produces an executable `a.out`. (The flag "`-g`" tells the compiler `gcc` to generate debug info for the compiled code: this debug info is used by Valgrind to produce sensible error messages that can be interpreted relative to the source code. The flag `-O0` ("Oh Zero") disables compiler optimizations: this helps valgrind relate the errors detected in the executable to the proper places in the source code.)

Now execute `a.out` under Valgrind by issuing the command "`valgrind ./a.out`". By default, the `memcheck` tool will be invoked.

The run will produce printout (the "Commentary"). Study it. Did `memcheck` find the planted bug in the code? If not, what could have prevented it from finding the bug?

3. Analyzing a second example

Next, consider the example program `main.c` (from the Valgrind manual). This program performs two flagrant memory errors: it makes an access outside a heap block, and it does not deallocate the allocated heap block (a memory leak). Compile `main.c` in the same way as the previous program (resulting in a new

`a.out`), and analyze the new `a.out` with `valgrind/memcheck` as before. What does the Commentary look like now? Did `memcheck` find the errors?

4. Analyzing a third example

Now consider `dynmem.c`. This is a slighty more complex program that allocates a list, and then deallocates it and prints its contents at the same time. The program contains two errors: try to spot them with `valgrind/memcheck`! Then fix the errors, and run the program with valgrind/memcheck again to see that no memory problems remain.

(You can edit `dynmem.c` with the text editor "`gedit`" in one of the following two ways:

- Open the file manager (double-click its icon to the upper left), open the "code-examples" folder, double-click on `dynmem.c`.
- In the terminal console, make sure that you are in the "code-examples" folder, then type "`gedit dynmem.c &`" at the prompt.)

5. Analyzing own code (optional)

If you still have the time, and some suitable C code of your own, then load it onto the machine, and analyze with `valgrind/memcheck`!

6. Continuing on your own

Valgrind is free software. You can obtain the sources from `www.valgrind.org`, and build it yourself. Also most linux distributions carry Valgrind, which then provides a convenient way to install the tool using the software manager of the distribution.

7. Assignment

The assignment for this lab is to write a short report that gives an account for your work and your findings in Sections 3, 4, and optionally 5 above. Describe which errors you found in the code, how memcheck reported them, and (when applicable) how you fixed the errors. If you analysed some own code, then it is very interesting to know whether you managed to uncover any previously unknown bug or weakness in the code. Send your report (pdf format) to bjorn.lisper@mdh.se.

If you want to, you can merge the reports for this exercise and the exercise on static analysis into a single report. Then make sure that the report is clearly structured into two distinct parts: one for each exercise.