Operating Systems II

SoSe 2016

— Lab Assignment 4 —

Verification

— Submission Email —

os-lab@deeds.informatik.tu-darmstadt.de

— Course Website —

https://www.deeds.informatik.tu-darmstadt.de/teaching/courses/ss2016/operating-systems-ii

Publication:	Mon, June 27, 2016
Submission:	Son, July 17, 2016 before 23:59
Testing:	Fri, July 22, 2016 (room A126)

Preface

In this lab, you will gain experience in software verification using state of the art tools. The lab is based on content from the lecture and extends some discussed concepts and techniques. You will learn how to verify programs written in the *C programming language* using an existing symbolic model checker before you implement your own simplified verifier.

Lab Environment

You should do the lab on your own machine. Make sure that you use a modern Linux distribution that includes recent (but not necessarily the latest) versions of the Linux kernel, the common command line and build tools. Most common distributions allow for easy installation and upgrade/downgrade of the required tools and software. Common choices for desktop Linux systems include Debian, Ubuntu, and Fedora.

You should have installed stable versions of the following basic development tools: GCC (4.9 or 5.0), Make (3.8 or 4.0), Python 2.7 and a sensible editor (e.g. Vim or Emacs).

Documentations and Finding Help

Most tools that have to be used in the lab come with a documentation. Make use of it! There are many forms of documentation, from simple plain text files to websites. In Linux systems, a common way to access documentations, especially for widely used standard tools and libraries, are the online reference manuals, the so-called man pages. Make extensive use of this feature. It is always a good idea to check the man pages if you are not sure how to use a certain tool or library. You can access the man pages with the following command:

```
man [section] <command-name>
```

Section 3 refers to C library functions and section 2 to system calls. The usage of section numbers is optional, but may be necessary when identical command names exist in different sections. For more details, try `man man` to access the man page for the man pages system ☺.

For general problems related to how tools work, how to accomplish a certain task using shell scripts, how to write Makefiles, etc., Stack Overflow and related online resources are a good place to start investigating: https://stackoverflow.com

Also, do not forget that online search engines and books from the library are your best friends!

Hints & Best Practice

We expect that you not only provide a *working and robust solution* for the tasks, but also adhere to *a consistent, structured, clean, and documented working and coding style*. Read the tasks thoroughly in order not to miss important details. Make sure that your solution matches the stated tasks. Keep your solution/code simple and do not lose sight of the actual tasks.

For tasks that require programming or scripting, you have to *implement proper error handling* mechanisms in your solution. For instance, function calls and program invocations may fail and their failure must be handled properly. Consult appropriate documentations to learn how error handling for the tools and libraries you use works. Often, problems with function calls and tools are indicated by magic return values (e.g. `-1` or `NULL`) or exit codes (e.g. `126`).

For programming tasks, make use of available build tools, compiler features, and linters. Make sure that your code compiles without any warnings by using the `-Wall` compiler switch during compilation (e.g., `gcc -Wall -o executable source.c`). If your code produces warnings during the testing session, we expect that you are able to explain each and every warning and justify why you did not fix it.

For building your solution, you should use the Make tool and write appropriate Makefiles instead of directly invoking the compiler. Automate your setup by writing shell scripts, e.g., for invoking a sequence of tools or gathering log files etc.

We expect each and every group member to know all the details of the tasks and your submitted group solution! During the testing session, every group member has to be able to explain, build, and execute the group solution. Moreover, each group member has to be able to answer questions related to the tasks and your solution.

Plagiarism & Fraud

We only accept *original solutions* of each individual group. Members of a group should work together very closely to solve the lab tasks. However, we do not support inter-group collaboration. If we notice that groups share the same solution, all involved groups will not get bonus points. Moreover, do not copy and paste code or other information from the Internet or other resources.

Be aware that there are a lot of tools that are able to automatically identify text and code that has been copied and pasted. We will make use of such tools to check whether your submitted solution is your original work.

Submission

Make sure to send your lab solution with all relevant files as tar archive via email to the address stated on the cover page before the deadline. You have to work in groups of 3 to 4 students. Do not submit individual solutions but send one email per group! ***In your submission email, include the names, matriculation numbers and email addresses of ALL group members.*** The testing date (see cover page) is mandatory for all group members. Therefore, ***indicate collisions with other TU courses on the testing date*** so that we can assign you a suitable time slot. Assigned time slots are fixed and cannot be changed. Students that do not appear for the assigned testing time will get no bonus points for that lab. During the test, we discuss details of your solutions with you to verify that you are the original authors and have a good understanding of your solution. The amount of bonus points you achieve depends on your performance in the testing session.

Miscellaneous

Please regularly check the course website as we publish up-to-date information and further updates there.

Questions regarding the lab can be sent to: os-lab@deeds.informatik.tu-darmstadt.de

Have Fun & Good Luck!

1) Verification Basics with CBMC

Throughout this lab, you will work with CBMC, a bounded model checker for the C programming language. You can obtain the tool on most Linux distributions by installing the cbmc package:

```
# apt-get install cbmc (Debian/Ubuntu)
# yum install cbmc (Fedora)
```

Once you have completed the installation, check that you have the right version:

```
# cbmc –-version
4.5
```

At this point, you should be able to verify c programs using the cbmc command as follows:

```
# cbmc <arguments*> file.c
```

Issuing the last command verifies the program assuming the main function as an entry point. You can specify an alternative function by using the “function” argument:

```
# cbmc --function foo file.c
```

To solve the tasks and for more information on how to use CBMC refer to the official manual on *http://www.cprover.org/cbmc/doc/manual.pdf*

In this task, you will verify the correctness of a program that calculates the sum of consecutive integers 1 + 2 + … + n:

```
int sum(int *arr, int len)
{
    int i, res=0;
    for(i = 0; i <= len; i++)
    {
          res += arr[i];
    }
    return res;
}

void start(int n)
{
    int arr[n];
    int i;
    for(i = 0; i < n; i++)
    {
          arr[i] = i+1;
    }
}
```

For this task, if the model checker takes too long (e.g., more than 15 minutes) to terminate, set the unwinding limit to 150.

```
# cbmc --unwind 150 file.c
```

Task 1.1

> Extend the provided program with assertions to verify the correctness of the sum function.
>
> The program is correct if the sum is equal to n * (n + 1) / 2.

Task 1.2

> Suppose that the parameter *n* of the start function is always smaller than 10. Modify the program to account for that.

Task 1.3

> Run CBMC on the code provided above and verify its correctness.

Task 1.4

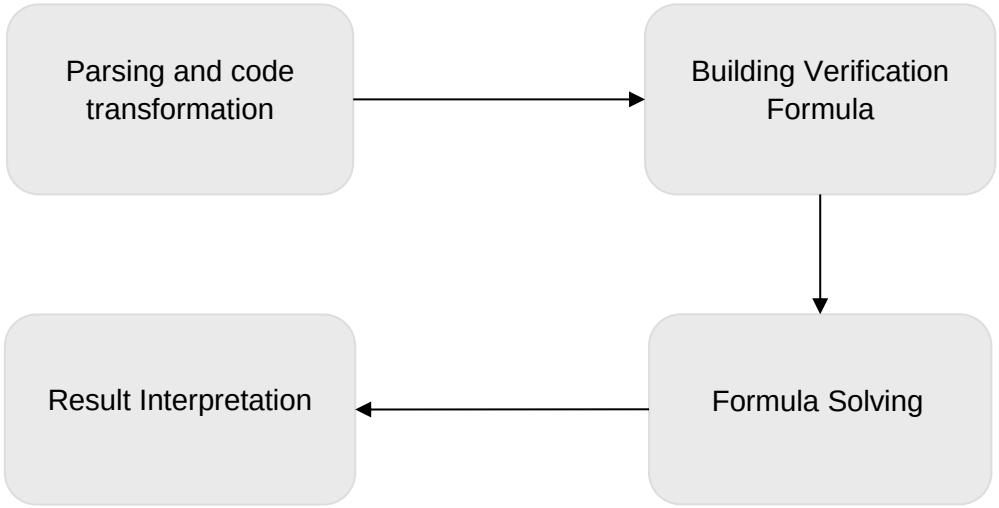
> Is the program correct? If not explain the problem, fix it and re-verify the program. What happens now?

Task 1.5

> Modify the program to fix the last issue and verify the program accordingly.

2) Symbolic Model Checking with the Z3 SMT solver

In this task you will use an SMT solver to verify the correctness of a program. Tools like CBMC use solvers in the backend to check the correctness of programs. The next figure shows the usual workflow of such a tool.



First, the program is parsed and transformed to an intermediate language. By doing so, model checkers can support any programming language for which a compiler to the intermediate language exists. Then, the verification formula is built and passed to an SMT/SAT solver. Finally, the satisfying assignment, if any, is interpreted and is presented to the user.

In this task you are required to encode the program from the previous task into a formula. You will then pass the formula to an SMT solver and interpret the result.

We will use the Z3 SMT solver. You can download the source from https://github.com/Z3Prover/z3. On the same page, you will find instructions for installing the solver. Make sure that you also install the Python binding.

You can verify that the installation was successful by importing the z3 Python module as follows:

```
from z3 import *
print “Successfully imported z3!”
```

A useful tutorial on how to use the z3 Python module can be found at *http://ericpony.github.io/z3py-tutorial/guide-examples.htm*. You can also check the API documentation available at: ***http://z3prover.github.io/api/html/namespacez3py.html***.

Task 2.1

> Install z3 by following the instructions on the page provided above and check that the z3 Python module was successfully installed.

Task 2.2

> Suppose that we set the unwinding of loops to 10 iterations. Transform manually the sum function from the last task accordingly.

Task 2.3

> Write a Python script to encode the body of the sum function from the previous task as shown in the Lecture. Automatically parsing the program is not required. Your script should create the necessary variables and build the verification formula using the z3 Python API. Assume that the array *arr* is already initialized with 10 elements in it as in the start function and that the *len* parameter is equal to 10.

Task 2.4

> Extend your script to solve the verification formula and interpret the results.