Assignment 2

Morten Poulsen[1] Lars Skovlund[2]

Department of Computing, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
{mrtn,lskovlun}@diku.dk

1. Introduction

In this assignment we will discuss tools to analyze program code. No program were given, so finding tools is a part of the assignment.

The basic problem is that many programmers make errors when programming. Finding and removing these bugs is tedious, and takes time and resources, so an automation of the process is interesting, even if it is only a partial automation. The most commonly used tool used by programmers to find errors in code is the compiler. However, most compilers only find syntax errors, not logical errors.

One strategy of debugging a program is to make a "white-box test". In this test all loops in all methods in the program are tested to make sure that all code is reachable and behave as expected. Identifying the different paths through a method can be a difficult task, especially if the method is complex. So a tool to calculate a flow graph from the code would be a great tool.

In order to make programming, debugging and maintenence less error-prone, one could also analyze the code to find places where it is complicated, where it is lacking in comments, and where parts of the code could be placed in methods outside the method in question. Tools to calculate these statistics ("Software metrics") for the code and its methods could be very useful as well.

There are also a host of regular programming errors, as well as methods that usually gives problems. Keeping close tabs on these can increase correctness of the code. Analyzing the compiled code can also give some insights, such as analyzing memory leaks and many other security problems.

In the rest of this essay, we will discuss and compare some of the different tools to analyze program code.

2. Valgrind

Valgrind [6] is an open-source tool for performing various profiling and debugging tasks on compiled code. At its core is an x86 emulator, which permits it to monitor all memory reads and writes. On top of this core are built

Written 3 May 2007.

several tools which perform cache effect simulation (`cachegrind`), call profiling (`callgrind`) and memory allocation checking (`memcheck`). `helgrind`, which tries to identify possible race conditions in a program, is not operational in the newest version of Valgrind.

Valgrind is an example of a *dynamic checking tool*, in that it performs its magic by watching the debuggee as it runs. The opposite, *static checking*, is most popular with functional languages and other language paradigms, in which there exists a well-defined mathematical theory that can be used to perform the checking. Languages like `C++` suffer in this regard because of the widespread use of side effects, which can be quite unpredictable just by looking at the source code. The tools exist (see, e.g., [1]), but with all tools of this type the golden rule is that it is better to try more than one product, because they work in different ways and find different problems.

Some Valgrind tools, like `memcheck`, generate output directly on the console. Others, like `callgrind` generate a data file that requires an external program to interpret meaningfully. This external program could be `KCachegrind`, which is a KDE-based front end to Valgrind. Another option might be to extend `IDA Pro` using a plug-in. We describe IDA Pro next.

3. IDA Pro

IDA Pro [5] is a tool which is very popular in the IT security business. It is a commercial disassembler product (free trial available without registration) with the special quirk that it is interactive. It starts out by loading your program into a database. It then runs an initial analysis to establish the control flow graph and variable use of the program. Finally, it loads library definitions corresponding to the target CPU, operating system and compiler, and "applies" them, labelling arguments as they are passed between routines in the code. After this, IDA Pro allows you to browse the program output, defining names and adding comments, struct types and enumerations. Everything is saved in the database to facilitate the analysis of larger programs.

In addition to the above features, IDA Pro supplies an extensive plugin interface which allows you to add support for more CPUs (or virtual machines) and binary formats, as well as writing interactive or non-interactive plugins. The plugins are written in C++ and have access to a substantial API. You can also write small programs in a built-in scripting language called IDC. IDA Pro also includes a debugger.

Given the control flow representation which is at the core of IDA Pro's internal representation, the product could serve as a solid base for writing various binary analysis tools. In particular, coverage tests which may be difficult to perform at the source level, can be performed easily in IDA Pro: We are interested in knowing which instruction paths lead to particular basic blocks. This is a simple matter of backtracking the instruction flow graph from the nodes of interest, analysing the branching constructs along the way.

4. Resource standard metrics

Resource Standard Metrics for C/C++, Java and C# (henceforth RSM) is developed by M Squared Technologies [2]. The central functionality of RSM is to calculate the software metrics of source code. It can calculate lines of code, lines of comments, McCabe Cyclomatic complexity (that is, number of logical paths through the code), and a few other metrics. This can be done for the sourcefile as well as for all methods in the sourcefile.

Additionaly, RSM can analyze the program for potential problems ("notices"), such as lines longer than 80 characters, usage of '=' in if-loops, or parentesis that does not balance. Users can also define notices for RSM to look for by using regular expressions. The output of RSM is written to a file, and the user can chose the format of the output-file. RMS supports regular formats such as txt or html. The output from RSM easily becomes very cluttered, which is a real problem in using the program. RSM comes with an (optional) gui that, while being better than using the command-line, isn't very intuitive.

Unlike the previous programs RSM is a static code analyzer. RSM can be downloaded in a trial version from [2] that is almost fully functional: It can only evaluate up to 10 source files, but it has no time-limit. There is both a Windows and an Unix version. The online documentation is fairly decent.

5. Purify

IBMs Rational Purify is a dynamic checking tool that run on compiled code. It's primarily used to find memory leaks and similar non-fatal memory errors in programs. These errors can be difficult to find and track, as they often do not cause fatal errors.

According to [3], if Purify has access to the source files it can pinpoint where the error happens. A test run of Purify on a C++ project with access to source code found some memory leaks, but were unable to detect where the problems arose in the source code. Most errors were in third-party libraries, but the location of the usage of these libraries in the project were not evident from the reports. Getting told that a memory leak happens in `_CRTDLL_INIT [f:\rtm\vctools\crt_bld\self_x86\crt\src\crtlib.c:434]` simply isn't very enlightening, and the fact that my system doesn't even have an F-drive is not an improvement! However, the total memory leakage were 12 KB, compared to that the program were using 20 MB, so it may very well be because there were no leaks in the project, but that there were errors in the third-party libraries, such as OpenGL.

Purify is able to monitor and report a number of Windows-specific phenomena which the alternatives we have examined don't report. These are things such as structured exception handling and OS-based debug output.

Purify exists in versions for Linux/Unix and Windows , and a 15 day trial can be downloaded from [4].

6. Conclusion

We have evaluated one open source and three commercial tools for analyzing code. Of the four, only RSM is a static checking tool. It would have been interesting to evaluate other static tools, but most were commercial and it was complicated to get an evaluation or trial version (one vendor actually wanted our phone number, so they could call us before we could download the trial!).

References

[1] www.coverity.com
[2] http://msquaredtechnologies.com
[3] http://en.wikipedia.org/wiki/IBM_Rational_Purify
[4] www.ibm.com/software/awdtools/purify/
[5] www.datarescue.com/idabase/
[6] www.valgrind.org