Towards Proper Differential Analysis of Static Analysis Engine Changes

Gábor Horváth, Réka Kovács, Péter Szécsi

Abstract: The design and implementation of heuristics for static analysis engines require detailed knowledge about the code to be analyzed. Extensive testing is therefore required to validate whether a change to the analysis engine is beneficial for real-world software projects.

The current practice of testing an analyzer on a fixed set of projects is not sufficient. Changes in the engine might affect language features that are utilized by only a fraction of the projects in the test suite. We explore a direction to ease the design of differential analysis experiments on a dynamic set of projects. This involves semi-automatic generation of the test set and evaluation of analysis results before and after applying changes to the engine. As the presented framework includes tools that aid the interpretation, reproduction, and sharing of analysis results, it might be valuable for a wide range of developers in the community.

Keywords: static analysis, symbolic execution, Clang, testing

Introduction

The proposition of a new patch to a static analysis engine involves disclosing information about the possible effects of the change. This normally includes analysis results on a few software projects before and after applying the patch.

Several problems arise during this process. Finding a set of test projects that truly show the effects of the patch can be a challenging task. Furthermore, a reviewer's request to extend the number of test projects might result in a significant amount of extra work for the patch author. This extra work is the result of the different components. With the continuous evolution of the static analysis engine, the author also needs to rerun the analysis on the old and the requested projets after a rebase. The results need to be processed so it can be easily digested by the reviewers. Ideally, the reproduction and extension of an analysis should be painless, and it should be possible to present results in an easily shareable and digestible format.

A related project, Corvig [1] is a tool to run dynamic and static analysis on projects and aggregate the results. Its emphasis is on collecting metrics about the analyzed projects and not on collecting metrics about the analyzers.

Overview

In this paper, we present a toolchain for the Clang Static Analyzer [2], a static analysis tool built on top of the Clang compiler for C family languages. The presented toolchain [3] aims to improve the situation by supporting both reviewers and authors in the following ways:

- help authors select a set of relevant projects for testing and run static analysis on them,
- aggregate statistics about the analysis (e.g.: how often a cut heuristic is triggered when building the symbolic execution graph),
- aggregate the results of the analysis,
- help authors and reviewers evaluate and share the results,
- help reviewers reproduce the results and maintain the tests.

The input of the toolset is a single, easy to interpret configuration file. The output is a HTML report with useful information and figures. The output also contains the input configuration for easier reproducibility.

Towards automatic test suite generation The conventional approach to testing is to run the analysis tool on a number of projects. Finding a sufficient amount of relevant real-world projects can be challenging. Ideal projects should be open-source for reproducibility and should exercise the right parts of the analyzer. For example, if the change is related to the modeling of dynamic type information, only projects using dynamic type information should be included. One possible option is to check a random sample of open-source projects, hoping to find enough that display all of the required traits. A slightly better approach is to use code searching and indexing services and for projects with interesting code snippets. These services, however, are optimized to present the individual snippets and suboptimal to retrieve the most relevant projects according to some criteria.

To mitigate this problem we present a script to harvest the results from an existing code search service and to recommend projects to be included in the test suite based on the results.

Towards easy reproduction and sharing The next problem is sharing results with reviewers. A regular pattern we see is the patch author sharing text files containing the results of the static analysis on certain projects. Text dumps of static analysis results are hard to interpret and the measurements are hard to reproduce. How did the author compile the project? Which version of the analyzed project was used? How did the author invoke the analyzer? What configuration options were used? What revision (commit) of the analyzer was used?

Our scripts use a concise configuration format that contains all the relevant information about the analyzed projects: repository, tag/commit, configuration options for the analysis, etc. Obtaining this configuration file enables reviewers to reproduce the exact same measurements at their convenience. They can also easily suggest modifications to the conducted experiment. Moreover, the results are not mere text dumps anymore but are presented on a convenient web user interface that also displays the path associated with the report. Other information such as the number of code lines of the project, version of the analyzer, analysis time, analysis coverage, and statistics from the analysis engine is recorded and figures like charts are generated automatically.

Towards more precise differential analysis Finally, the number of tools available to support differential analysis on a project is scarce. In case of the Clang Static Analyzer, we can only compare the number of bugs found, analyzer engine statistics, and the coverage percentage measured in basic blocks. All of these are aggregated scalar values missing positional information, with the statistics and the coverage being displayed individually for each translation unit.

We implemented analysis coverage measurement right in the heart of the analyzer. Using the extra information provided by the modified engine we can perform differential analysis on the coverage itself instead of settling for the study of coverage percentage values alone. Reviewers can check which new lines became covered after the change and which lines left the scope for some reason. We also support differential analysis of the analyzer reports.

Recommended workflow Using our toolset the recommended workflow is the following. The author of the patch provides reviewers with a link to the test results. Reviewers can choose to either merely look at the results or repeat the whole experiment based on the configuration, depending on the verification effort required for the change. They can also suggest changes to the configuration to gather more insight about the changes.

The Proposed Toolchain

Semi-automatic test suite generation Our script addressing the collection of relevant test project candidates for an engine change uses the `searchcode.com` service for its backend. For example, in order to test a new static analysis check written to detect `pthread_mutex_t` abuse, we might be interested in projects that use `pthread` extensively.

Using the following syntax we can specify the keywords to search for, the languages we are interested in, the desired number of projects:

```
1 python gen_project_list.py 'pthread_mutex_t' 'C C++' 5 ——output pthread.json
```

The above call creates a configuration file with the suggested projects in the following format:

```
{ "projects": [ { "url":"github.com/itkovian/torque.git", "name": "torque" }, ... ]
  }
```

Easy analysis reproduction and sharing The file above is almost enough to run the analysis on its own. The only extra information needed to be specified is the CodeChecker [4] server where analysis results are intended to be stored for later inspection.

```
{ "projects": ... , "CodeChecker": { "url" : "localhost:15010/Default" } }
```

CodeChecker is a tool designed to integrate the Clang Static Analyzer and Clang Tidy into C/C++ build systems. It also acts as a mature bug management system that supports the commenting on static analysis reports and the suppression of false positives. It has a convenient user interface to visualize the path of the path-sensitive bug reports and to support differential analysis. We can compare two analysis runs using CodeChecker to differentiate between common reports and those present only in a specific analysis run. CodeChecker's web GUI allows sharing the results with the rest of the world without the need of repeating the experiment. It can be used not only to share the bug reports, but the corresponding classifications and comments why some findings were false positives or true positives according to the author of the patch.

After this we are ready to run the analysis on the previously selected set of projects:

```
python run_experiments.py ––config pthread.json
```

The script checks out each of the projects, attempts to infer their build system and build them, runs the analysis and finally collects the results. At the time of writing this paper `autotools`, `CMake`, and `make` are supported as build systems.

In case a special build command is required or the build system is not yet supported, the user can specify the build command. Building a special version of the project characterized by a tag or a commit hash instead of top of tree is also possible and highly encouraged to get consistent results with subsequent experiments. Finally, differential analysis can currently be conducted by running the same projects multiple times with different options passed to the analyzer or using different versions of the analyzer. An example can be seen below.

```
  { "projects": [ ... ],
    "configurations": [
      { "name": "original", "clang_sa_args": "", },
      { "name": "variant A",
        "clang_sa_args": "argument to enable feature A",
        "clang_path": "path to clang variant" }
    ],
  ... }
```

A more precise differential analysis Currently, coverage measurements provided by the Clang Static Analyzer are limited. The engine can only record the percentage of basic blocks reached during the analysis of a translation unit, which is not sufficiently precise for multiple reasons. First, the analysis can stop in the middle of a basic block due to running out of the analysis budget for that specific execution path. Secondly, there is no precise way of merging information from different translation units. Finally, inline functions or templates in header files might appear in multiple translation units and their contribution will be counted multiple times upon attempting to aggregate information over translation units.

We implemented line-based coverage measurement based on the `gcov` format. We do not calculate coverage as an overall percentage value but record it separately for each line. This makes it possible to precisely aggregate coverage information over translation units. This also makes it possible to do differential analysis on the coverage itself. Our toolset includes scripts to aid that kind of analysis.

In some cases, we are interested in the reason behind a specific bug report disappearing when running the analysis with different parameters. Performing differential analysis on the coverage, we are able to determine whether the analyzer actually examined the code in question during both runs.

The Clang Static Analyzer can output different kinds of statistics such as the number of paths examined, the number of times a specific cut heuristic was used etc. Instead of having a fixed set of statistics

to collect we used some heuristics to process the output of the analyzer, in which we are able to automatically detect statistics and aggregate them over translation units.

We create an HTML summary of the statistics collected during the analysis. This report includes charts and histograms. After adding a new statistic to the analyzer engine the author only needs to add a single entry in the configuration file to make the toolset generate a figure based on that statistic.

Conclusions

We find the traditional practice of static analysis tool testing insufficient. One of the greatest problems is that a fixed set of test projects might not stress the newly introduced code paths of the analysis engine. The other concern is reproducibility, which is not only essential for reviewers tinkering with the measurements but also for any subsequent re-evaluation of the changes. Finally, the current practice of presenting the measurement results does not aid the interpretation of the raw data. Using an easier to digest presentation of the measurements could reduce the effort needed to evaluate the changes for the reviewers.

In order to mitigate these issues, we suggested a particular analysis workflow and developed a supporting toolchain the Clang Static Analyzer. These tools not only help to collect relevant candidate projects for testing but also perform a proper differential analysis on the test projects and generate easy to interpret figures for the reviewers. We also added a new line-based coverage measurement mechanism to the Clang Static Analyzer engine to improve the precision of the analysis.

Acknowledgement

Supported by ELTE Eötvös Loránd University in the frame of the ÚNKP-17-3 New National Excellence Program of the Ministry of Human Capacities.

References

[1] P. Marinescu, P. Hosek, and C. Cadar, "Covrig: A framework for the analysis of code, test, and coverage evolution in real software," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 93–104. [Online]. Available:

[2] "Clang static analyzer," 2018. [Online]. Available: https://clang-analyzer.llvm.org/

[3] G. Horvath, R. Kovacs, and P. Szecsi, "Clang static analyzer testbench," 2018. [Online]. Available: https://github.com/Xazax-hun/csa-testbench

[4] "Codechecker," 2018. [Online]. Available: https://github.com/Ericsson/codechecker