Acta Cybernetica **25** (2022) 781–795.

Report on the Differential Testing of Static Analyzers*

Gábor Horváth[ab], Réka Kovács[ac], and Péter Szécsi[ad]

Abstract

Program faults, best known as bugs, are practically unavoidable in today's ever growing software systems. One increasingly popular way of eliminating them, besides tests, dynamic analysis, and fuzzing, is using static analysis based bug-finding tools. Such tools are capable of finding surprisingly sophisticated bugs automatically by inspecting the source code. Their analysis is usually both unsound and incomplete, but still very useful in practice, as they can find non-trivial problems in a reasonable time (e.g. within hours, for an industrial project) without human intervention.

Because the problems that static analyzers try to solve are hard, usually intractable, they use various approximations that need to be fine-tuned in order to grant a good user experience (i.e. as many interesting bugs with as few distracting false alarms as possible). For each newly introduced heuristic, this normally happens by performing differential testing of the analyzer on a lot of widely used open source software projects that are known to use related language constructs extensively. In practice, this process is ad hoc, error-prone, poorly reproducible and its results are hard to share.

We present a set of tools that aim to support the work of static analyzer developers by making differential testing easier. Our framework includes tools for automatic test suite selection, automated differential experiments, coverage information of increased granularity, statistics collection, metric calculations, and visualizations, all resulting in a convenient, shareable HTML report.

Keywords: static analysis, symbolic execution, Clang, testing

*The project has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002)

[a]Department of Programming Languages and Compilers, Eötvös Loránd University, Budapest, Hungary

[b]E-mail: xazax@caesar.elte.hu, ORCID: 0000-0002-0834-0996

[c]E-mail: rekanikolett@caesar.elte.hu, ORCID: 0000-0001-6275-8552

[d]E-mail: ps95@caesar.elte.hu, ORCID: 0000-0001-9156-1337

DOI: 10.14232/actacyb.282831

1 Introduction

Any significant change to an open-source static analysis tool (also simply called *an analyzer*) is preceded by a discussion about its possible effects. The minimum typical requirement is the comparison of analysis results and performance on a few software projects before and after applying the changes.

Fulfilling this requirement is hard for various reasons. Developers often have a bias towards a set of projects they are familiar with, which might tempt them to avoid the challenge of finding a set of test projects that most effectively exercise the changed parts of the analyzer. In case of a long-lasting open-source review process [16] (developers often have to wait half a year before their contributions get accepted), changes need to be re-based on top of the latest version of a continuously evolving code base, and the analysis of all test projects needs to be re-run to ensure that the feature still works correctly. Analysis results also have to be processed and summarized to be easily understandable for the reviewers.

Note that, for our explanations throughout the paper, we use the term *author* to refer to the person who implements a change to the open-source application (this comes from being the author of the *patch*, a textual form of the set of actual modifications to the source code). This person has to justify the changes and prove to *reviewers* that there will be no unwanted regressions in the software's behavior. By *reviewers* we mean those fellow developers who audit and approve the changes.

Ideally, reproducing an analysis should be painless, and it should be possible to present results in an easily shareable and digestible format. This format should be simple to archive or embed in documentation, so that major design decisions can be easily re-evaluated later. This is important, since the decisions that make perfect sense today might be less adequate tomorrow.

In this paper, we present our toolchain that we call the *Clang Static Analyzer Testbench* (or *CSA Testbench*)[1], designed for the Clang Static Analyzer [1] and Clang-Tidy [2], two open-source static analysis tools built on top of the Clang [11] compiler for C family languages. The Clang Static Analyzer uses symbolic execution [10] to find bugs, while Clang-Tidy is a collection of syntactic checks. We are long-term contributors to these tools, and would like to share the principles of the differential testing infrastructure we have built with a wider community.

Our framework aims to enhance the open-source review process by supporting *reviewers* and *authors* (as defined earlier) in the following ways:

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

[1] The code is open-source, licensed under the MIT License, and can be downloaded from https://github.com/Xazax-hun/csa-testbench.

- help reviewers reproduce the results and suggest changes to the test setup,
- help authors maintain the tests.

The input of the toolset is a single and easy-to-interpret configuration file in JSON format. Since the format is textual, reviewers can comment on the test setup using conventional review tools and it can also be embedded in documentation. Moreover, it is convenient to store such files in version control systems. The output is a customizable HTML report with useful information, various plots, and a record of the input configuration, including the version numbers, to ensure reproducibility. The goal is to store all the information required to repeat the experiment.

Our principles can be reused by developers of other static analyzers, and we also describe some alternative use cases for our framework.

The paper is structured as follows. Section 1 gives an overview of the difficulties faced during an open-source review process that requires differential testing. Section 2 introduces the principles behind the framework we built to tackle these problems.

2 The CSA Testbench Toolchain

2.1 Semi-automatic test suite generation

Problem After implementing a missing feature or tweaking an existing part of a static analyzer, testing the robustness of the change and checking whether a regression occured is a natural requirement towards the author. One conventional approach is to run the analyzer tool on a number of real-world software projects and artificial regression tests.

Finding a sufficient number of relevant real-world projects can be challenging. Ideal projects should be open-source and easy to set up, so that reviewers have a better chance of reproducing the results. Additionally, projects should exercise the right parts of the analyzer. For example, if the change is related to dynamic type information modeling, only projects using dynamic type information should be included.

One option is to use a trial-and-error approach and check a random sample of open-source projects, hoping to find enough that display the required traits. A slightly better approach is to use code searching and indexing services and look 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.

Solution We present a script that harvests the results of an existing code search service, and recommends projects to be included in the test suite based on the results. This script can spare a significant amount of development time and help authors find relevant projects on which their changes can be verified.

Our script uses the SeachCode [5] 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 syntax on Listing 1, we can specify the keywords to search for, the languages we are interested in, the desired number of projects and optionally a filename for the output:

```
$ ./gen_project_list.py 'pthread_mutex_t' 'C C++' 3 -o pthread.json
```

Listing 1: A sample invocation of the project list generator tool.

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

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

Listing 2: A fraction of a configuration file generated by the project list generator tool invocation showed on Listing 1.

This configuration file can be directly used as input to the main driver script of the testing infrastructure as detailed in Section 2.2.

Sometimes we only want to do a *stress test* to ensure that the analysis engine behaves gracefully for all projects and does not crash. We created an alternative tool to create a configuration file based on a Debian FTP mirror for package sources. The resulting file will contain more than 20 000 projects.

```
$ ./project_list_from_debian.py \
  --url ftp://ftp.se.debian.org/debian/ --output debian.json
```

Listing 3: Sample call of an alternative project list generator tool that lists all packages available at the specified Debian mirror.

2.2 Easy analysis reproduction and sharing

Problem A regular pattern is that the developer sharing text files that contain static analysis results on a set of projects. This makes evaluation considerably difficult for reviewers. First of all, they might not be familiar with the test projects at all. Text dumps of static analysis results are hard to interpret and the measurements are hard to reproduce. Further questions that might arise: How did the author compile the project? Which version of the analyzed project was used? How did the author invoke the analyzer? Which configuration options were used? Which revision (commit) of the analyzer was used?

Solution Our tools 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, with the help of our driver script. They can also easily suggest modifications to the conducted experiment.

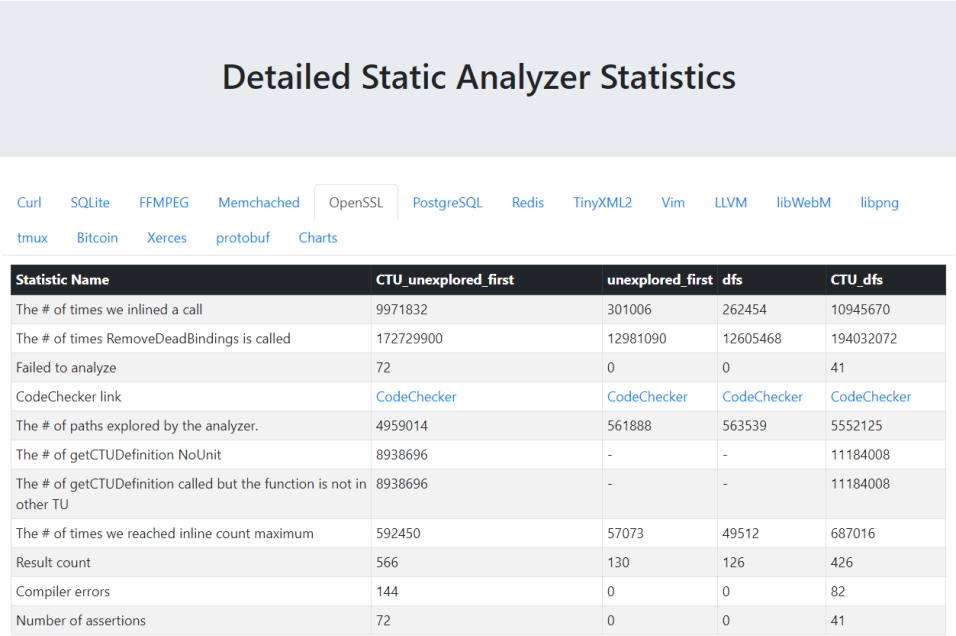
The scripts aggregate useful information about the analysis into an easy-to-share HTML format (as seen in Figure 1). Analysis results are not mere text dumps anymore, but are presented on a convenient web user interface that also displays the path associated with the report (showed in Figure 2). Other information such as the number of code lines in the project, version of the analyzer, analysis time, analysis coverage (Figures 3 and 4), and statistics from the analysis engine is recorded and charts are generated automatically (Figure 5). The web user interface also has permanent links to each individual error report in order to make it easier to refer to them in discussions. These pages are hosted by the person sharing the results, code reviewers do not need to install anything to browse the results.

The configuration file showed in Section 2.1 is almost enough to run the analysis on its own. The only extra information needed to be specified is the URL of the *CodeChecker* server where analysis results are intended to be stored for later inspection (Listing 4).

```
1  {
2    "projects" : ...
3    "CodeChecker": {
4      "url" : "localhost:15010/Default",
5    }
6  }
```

Listing 4: A segment of the configuration file specifying the address of the CodeChecker server.

CodeChecker [3] is a tool we 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 commenting on static analysis reports and suppressing false positives. It has a convenient user interface to visualize path-sensitive bug reports (see Figure 2) and to support differential analysis. We can compare two analysis



Detailed Static Analyzer Statistics

Curl SQLite FFMPEG Memchached OpenSSL PostgreSQL Redis TinyXML2 Vim LLVM libWebM libpng tmux Bitcoin Xerces protobuf Charts

Statistic Name	CTU_unexplored_first	unexplored_first	dfs	CTU_dfs
The # of times we inlined a call	9971832	301006	262454	10945670
The # of times RemoveDeadBindings is called	172729900	12981090	12605468	194032072
Failed to analyze	72	0	0	41
CodeChecker link	CodeChecker	CodeChecker	CodeChecker	CodeChecker
The # of paths explored by the analyzer.	4959014	561888	563539	5552125
The # of getCTUDefinition NoUnit	8938696	-	-	11184008
The # of getCTUDefinition called but the function is not in other TU	8938696	-	-	11184008
The # of times we reached inline count maximum	592450	57073	49512	687016
Result count	566	130	126	426
Compiler errors	144	0	0	82
Number of assertions	72	0	0	41

Figure 1: A section of the automatically produced HTML report containing information about analysis runs with different analyzer configurations. The table contains links to the corresponding analysis runs in a web user interface (see Figure 2), and links to detailed line-based coverage reports (Figures 3 and 4). A similar table for each analyzed project can be found under appropriately labeled tabs in the header of the report. The *Charts* tab hides a number of interactive charts generated from the results (for an example, see Figure 5).

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 needing to repeat the experiment. It can be used to share not only bug reports, but also classifications and comments explaining why some findings are considered false positives or true positives.

After adding this detail to the configuration file, we are ready to run the analysis on the previously selected set of projects (Listing 5).

```
1 $ ./run_experiments.py --config pthread.json
```

Listing 5: Sample invocation of the main driver script of the experiment.

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

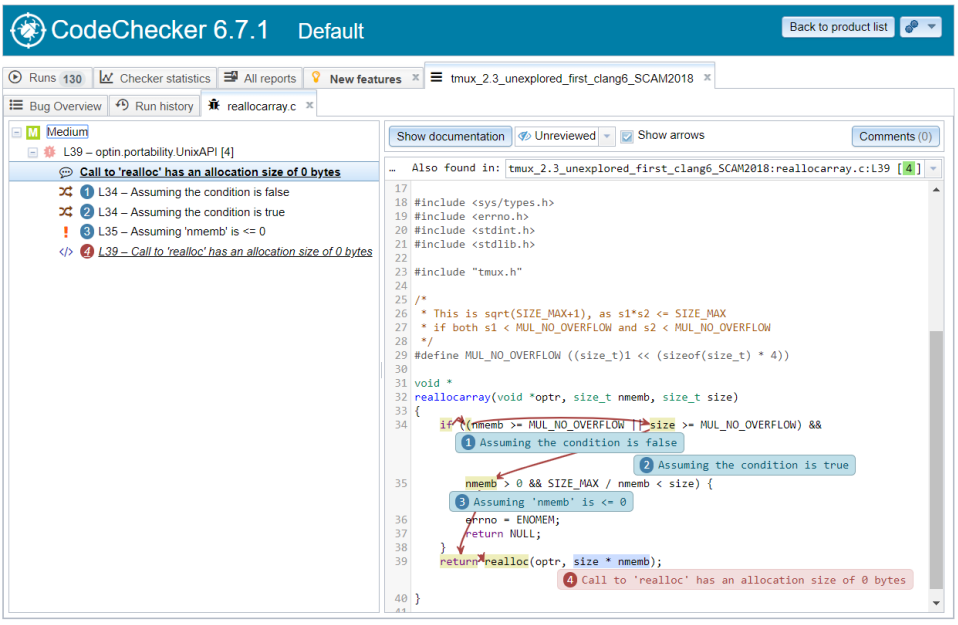


Figure 2: The CodeChecker web user interface. Path-sensitive reports guide the user along the execution path leading to the bug. On the web user interface, different runs can be compared against each other, and bug reports can be filtered by many criteria, e.g. by severity, by review status, by detection status, by detection date, by checker name, by checker message, etc. Bug reports can also be marked false positive, with the possibility of leaving an explaining note for the record.

However, the script will not download and install all the dependencies required to compile the projects. It is the user's responsibility to ensure that the host machine is able to compile the projects, which turned out to be a big burden for the authors. For this reason, we introduced support for the two emerging C++ package managers, Conan [4] and Vcpkg [6]. Relying on these package managers instead of repository URLs ensures that the analysis will not fail due to a missing dependency. In Listing 6, we can see how easy it is to test on a project which is available in one of the package managers.

```
{
  "projects": [
    {
      "name": "zlibconan",
      "package": "zlib/1.2.11@conan/stable",
      "package_type": "conan",
    },
    {
      "name": "zlibvcpkg",
      "package": "zlib",
```

```
      "package_type": "vcpkg",
    }
  ]
}
```

Listing 6: A sample configuration file that will instruct the framework to download projects using the *Conan* and *Vcpkg* package managers.

In case a special build command is required, or the build system is not yet supported, the user can specify the build command and the configuration command. Building a specific version of the project determined by a tag, a commit hash, or a URL to a source tarball instead of top of tree is also possible and highly encouraged, in order to get consistent results in 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 (Listing 7).

```
{
  "projects": [
    {
      "url": "github.com/itkovian/torque.git",
      "name": "torque",
      "tag": "tag name",
      "build_command": "special build command"
    }, ...
  ]
  "configurations": [
    {
      "name": "original",
      "clang_sa_args": "",
    },
    {
      "name": "variant A",
      "clang_sa_args": "argument to enable feature A",
      "clang_path": "path to clang variant"
    }
  ], ...
}
```

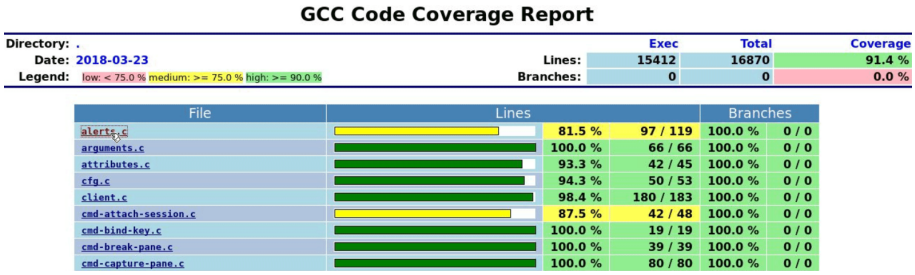
Listing 7: Differential testing can be achieved by running many analysises on the same projects with different options passed to the analyzer.

2.3 A more precise differential analysis

Problem 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.

Solution We implemented line-based coverage measurement based on the `gcov` [7] 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, and to do differential analysis on the coverage itself. Our toolset includes scripts to aid that kind of analysis.



GCC Code Coverage Report

		Exec	Total	Coverage
Directory: .				
Date: 2018-03-23	Lines:	15412	16870	91.4 %
Legend: low: < 75.0 % medium: >= 75.0 % high: >= 90.0 %	Branches:	0	0	0.0 %

File	Lines		Branches	
alerts.c	81.5 %	97 / 119	100.0 %	0 / 0
arguments.c	100.0 %	66 / 66	100.0 %	0 / 0
attributes.c	93.3 %	42 / 45	100.0 %	0 / 0
cfg.c	94.3 %	50 / 53	100.0 %	0 / 0
client.c	98.4 %	180 / 183	100.0 %	0 / 0
cmd-attach-session.c	87.5 %	42 / 48	100.0 %	0 / 0
cmd-bind-key.c	100.0 %	19 / 19	100.0 %	0 / 0
cmd-break-pane.c	100.0 %	39 / 39	100.0 %	0 / 0
cmd-capture-pane.c	100.0 %	80 / 80	100.0 %	0 / 0

Figure 3: A sample report summarizing coverage percentages over the analyzed files. Line-based coverage information can be browsed by clicking on filenames.

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 text mining to process the output of the analyzer, in which we are able to automatically detect newly added custom statistics without any additional configuration, and aggregate them over translation units.

As mentioned in Section 2.2, the final report of our toolset includes figures like charts and histograms. The list of figures can be set in the configuration file. 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. One sample use-case is producing a histogram of analysis times per translation unit. This can help us track down performance regressions in outliers.

We cannot emphasize the importance of automatically generated figures enough. The statistics about a run of the symbolic execution engine is not easy to interpret. For example, an increase in the number of generated symbolic states can be both a good and bad news depending on how the rest of the statistics are changed. Not requiring the author to create the figures from the numbers manually is a great productivity boost.

Figure 4: Our amended version of the Clang Static Analyzer can provide coverage information for each executed line of each analyzed file. In Figure 3, lines covered by the analysis are shown in a green color, while lines not covered are shown in red. White lines contain no executable code.

C-Reduce [15] is a tool that takes a large C, C++, or OpenCL file that has a property of interest (such as triggering a compiler bug) and automatically produces a much smaller C/C++ file that has the same property. We also use C-Reduce to get minimal examples that showcase differences between two versions of the static analysis engine. First, we need a file on which analysis engine versions produce different results. This can be a different set of warnings or other statistics emitted by the engine. These minimal examples can greatly aid our understanding of the effects of a change. The main shortcoming of C-Reduce is the lack of support for reducing multiple translation units at once. We do plan to add this feature in the future.

2.4 Recommended workflow, how to use the toolchain

Using our toolset the recommended workflow is shown in Figure 6. The author of the patch uses some of our scripts to select the project to test the changes on. After running the experiments she makes sure all the data support the hypothesis. Then she uploads tha patch for review and provides reviewers with a link to the test results which includes the configuration. 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

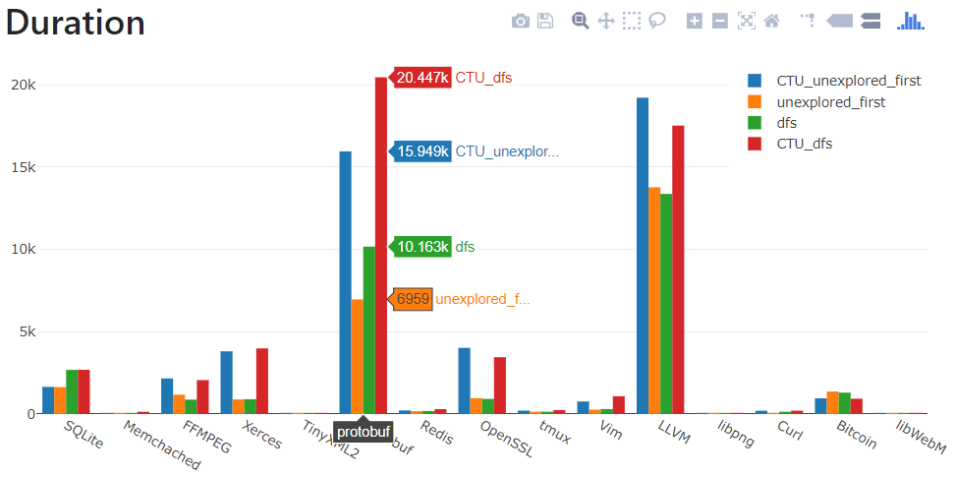


Figure 5: One of the many interactive charts generated based on statistics collected during analysis, this figure shows duration times for different analyzer configurations for different open-source projects. Precise numbers are shown when hovering over the block of columns corresponding to a project.

changes to the configuration to gather more insight about the changes. After such suggestions it is as easy to re-run the whole experiment as pushing a button.

2.5 Alternative applications

The tools we introduced in the previous section can be generalized beyond supporting only static analysis engines. First, obtaining a set of projects with certain properties (e.g. projects using runtime type information) can be valuable for the testing of any language tool. Secondly, the ability to check out and analyze any number of past tags of a project and perform differential analysis on them enables the collection of historical data about the evolution of the project's coding conventions. We can also track the number of findings over time for a certain project.

We also found that these scripts are great to build CI loops. Running the analysis on a set of projects for each commit is a great way to find regressions. We introduced a flag to break the CI loop each time the analysis of a project fails for some reason. The reported HTML will contain useful information about the analysis failures as well as assertion messages.

Finally, one of the most interesting applications of our scripts is automatic parameter tuning. Some static analysis engines have a great number of adjustable parameters. Our tools are not only suitable for running the analysis, but also for setting its parameters and measuring time, coverage, engine statistics, and the number of reported bugs. Using this information, a machine learning algorithm can attempt to optimize the parameters in order to improve the quality of the analysis.

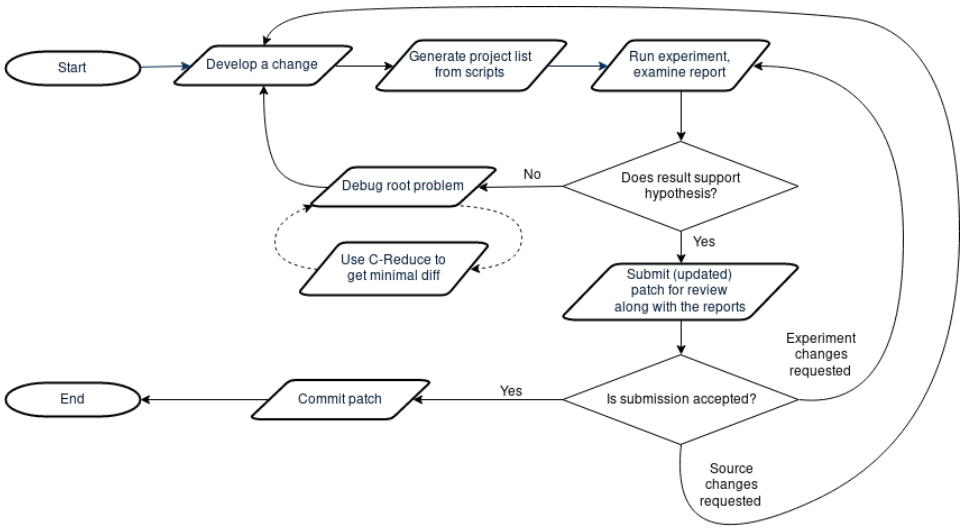


Figure 6: A flowchart describing the recommended workflow when using the CSA Testbench to do differential testing of an analyzer change.

2.6 Future work

Unfortunately, using textual queries to get a set of interesting projects is not sufficient. There are certain language constructs that are hard to query this way, such as implicit casts or structured bindings. Likewise, using code search services is also an imperfect solution, a semantic indexer would probably be more suitable.

We intend to introduce more (optional) measurements into the scripts such as memory profiling during analysis. We also plan to perform a more detailed analysis of how the proposed process can improve the quality of the static analysis engine.

These set of tools are the result of optimizing the productivity of our team while working on some static analysis tools. While each added feature helped to improve our work-flow, it is hard to quantify the improvements. We plan to conduct some surveys in the future to verify the usefulness of our framework among a wider community of researchers and developers.

3 Related Work

The difficulty of performing static analysis varies among programming languages, due to differences in the number and maturity of tools written for them. Two languages on the worse end of the spectrum are C and C++, as no widely used build system or package repository exists in their fragmented ecosystem. Having tools to deal with software repositories directly can be a step towards overcoming this problem and helping researchers perform more rigorous evaluations for their tools targeting these languages. Since C++ is a language of enormous size, most

projects use a relatively small subset of it. For this reason, finding a good set of test projects is even more critical.

This problem is less likely to surface during the analysis of other languages. Some of them, like Java, armed with Maven repositories, are in a convenient position for experimentation. Software packages can be easily downloaded, built and analyzed. Fortunately, the C++ community realized the value of having package managers, and now two of them named Conan [4] and vcpkg [6] started to gain popularity, but have not reached wide adoption yet.

In the following paragraphs, we describe tools that play a similar role for other programming languages than our framework for C++.

VISUFLOW [9] is a tool to help debug static analysis software. While it is great for debugging problems on small reproducers, it is not suitable to debug problems that only manifest on large projects, such as cut heuristics and exploration strategy related issues in symbolic execution. The same author conducted a survey with 115 analysis writers [8]. They concluded that the state-of-the-art tools were not sufficient to fulfil the needs of static analysis software authors. The participants of the survey identified graph visuals, access to the intermediate representation and intermediate result count as very important features, and our framework excels at visualizing intermediate counters (statistics) over a large corpus of test projects.

Using static analysis together with mining is not a new idea. Macedo et. al. [12] used the mining of malware and static analysis together to extract behavioral patterns aiming to identify malware. The difference from our work is that we are mining repositories in order to improve the quality of a static analysis tool.

Covrig [13] is a tool to run dynamic and static analysis on several projects and aggregate the results. It is supporting a different use-case than our tool. Its emphasis is on collecting metrics about the analyzed projects and not on collecting metrics about the analyzers.

Ray et. al. [14] used entropy as a measure for comparing static analysis findings in order to correct code. They found that search-based bug-fixing methods may benefit from using entropy both for fault-localization and for the searching for fixes. Our presented toolset might help conduct similar studies in the future for C family languages, as it supports comparing a patched and unpatched (or differently configured) version of the static analysis engine.

4 Conclusions

We find the traditional practice of static analysis tool testing cumbersome and 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, but for any subsequent re-evaluation of the changes. As the analyzer evolves, some of its distinct parts interact with each other. Consequently, some of the changes that seemed sensible in the past might become irrational in the future. Having a record of experiments from the past facilitates the re-evaluation of those decisions in the light of new circum-

stances. Finally, the current practice of presenting the measurement results does not aid the interpretation of the raw data. Using an easier-to-digest representation of measurements would reduce the effort needed to evaluate the changes.

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

References

[1] Clang Static Analyzer, a source code analysis tool for C, C++, and Objective-C programs. URL: https://clang-analyzer.llvm.org/ (Retrieved: 23/03/2019).

[2] Clang-Tidy, a static analysis and code refactoring tool. URL: http://clang.llvm.org/extra/clang-tidy/ (Retrieved: 23/03/2019).

[3] CodeChecker, a defect database and viewer extension for Clang-Tidy and the Clang Static Analyzer. URL: https://github.com/Ericsson/codechecker (Retrieved: 23/03/2019).

[4] Conan, an open-source C/C++ package manager. URL: https://conan.io/ (Retrieved: 23/03/2019).

[5] SearchCode, a free source code search engine. URL: https://searchcode.com/ (Retrieved: 23/03/2019).

[6] Vcpkg, a C/C++ library manager for Windows, Linux, and MacOS. URL: https://docs.microsoft.com/en-us/cpp/vcpkg (Retrieved: 23/03/2019).

[7] Bhushan, Ram Chandra and Yadav, Dharmendra Kumar. Number of test cases required in achieving statement, branch and path coverage using 'gcov': An analysis. In *2017 the 7th International Workshop on Computer Science and Engineering*, pages 176–180, 2017. DOI: 10.18178/wcse.2017.06.031.

[8] Do, Lisa Nguyen Quang, Krüger, Stefan, Hill, Patrick, Ali, Karim, and Bodden, Eric. Debugging static analysis. *CoRR*, abs/1801.04894, 2018.

[9] Do, Lisa Nguyen Quang, Krüger, Stefan, Hill, Patrick, Ali, Karim, and Bodden, Eric. Visuflow: a debugging environment for static analyses. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 89–92. ACM, 2018. DOI: 10.1145/3183440.3183470.

[10] Hampapuram, Hari, Yang, Yue, and Das, Manuvir. Symbolic path simulation in path-sensitive dataflow analysis. *SIGSOFT Softw. Eng. Notes*, 31(1):52–58, September 2005. DOI: 10.1145/1108768.1108808.

[11] Lattner, Chris. LLVM and Clang: Next generation compiler technology. Lecture at BSD Conference 2008, 2008.

[12] Macedo, Hugo Daniel and Touili, Tayssir. Mining malware specifications through static reachability analysis. In Crampton, Jason, Jajodia, Sushil, and Mayes, Keith, editors, *Computer Security – ESORICS 2013*, pages 517–535, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-40203-6_29.

[13] Marinescu, Paul, Hosek, Petr, and Cadar, Cristian. 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*, ISSTA 2014, pages 93–104, New York, NY, USA, 2014. ACM. DOI: 10.1145/2610384.2610419.

[14] Ray, Baishakhi, Hellendoorn, Vincent, Godhane, Saheel, Tu, Zhaopeng, Bacchelli, Alberto, and Devanbu, Premkumar. On the "naturalness" of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 428–439, New York, NY, USA, 2016. ACM. DOI: 10.1145/2884781.2884848.

[15] Regehr, John, Chen, Yang, Cuoq, Pascal, Eide, Eric, Ellison, Chucky, and Yang, Xuejun. Test-case reduction for c compiler bugs. In *ACM SIGPLAN Notices*, Volume 47, pages 335–346. ACM, 2012. DOI: 10.1145/2345156.2254104.

[16] Rigby, Peter C and Storey, Margaret-Anne. Understanding broadcast based peer review on open source software projects. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 541–550. IEEE, 2011. DOI: 10.1145/1985793.1985867.