A survey on systematic construction of benchmarks for Static Analysis tools

Kasra Qasempour Soleimany
Matriculation number: 2777218
k.qasempour@gmail.com

Vivin
Matriculation number: Vivin
Vivin

Abstract
best abstract ever

1. Introduction

Static analysis tools (SAT) has been become popular recently. The BCS SIGIST defines static analysis of source code as the ”analysis of a program carried out without executing the program” (Glo). Thus the input for a static analysis tools is source code or other artifact builds from source code, like binary. However, there is not a clear list of metrics or definition in order to compare and benchmark these tools. Generally the main issue with static analysis programs is completeness and soundness. A static analysis program is either unable to reliably detect all targeted problems or is prone to false positives, i.e., it reports findings which turn out to be wrong on closer examination (Johns and Jodeit 2011). In this paper we provide a list of recent studies on how to systematically create and build benchmarks for static analysis tools and we try to bold commons and differences. we study these papers ...

2. Foundations

Before we dive in to approaches, we take look over some criterion of SATs introduced by Chess and West (Chess and West 2007):

- **C1 - Quality of the analysis** which indicated the precision of a given tool. Like the language feature coverage, or rate of false negative/positive.
- **C2 - Trade-off between precision and scalability** Like trade-off between analysis duration and unwanted results, or memory usage vs depth of data flow to capture.
- **C3 - Vulnerability types coverage** How many vulnerabilities is the tool aware of.
- **C4 - Usability of the tool** How easy the tool can be used.

We mentioned that the input of a SAT is the source code, and correspondingly the benchmarking approaches are categorized based on the input. Johns and Jodeit define 3 different benchmarking approaches (Johns and Jodeit 2011):

- **Real-world, vulnerable software** Test on big open source projects which has some known vulnerabilities. The approach is perfect for indicating C2, and the overall tool performance. It has some limitation, for example if a tool fails to find a known issue, what is the reason behind that? Does it stem from C1 or C2? or maybe the tool does not cover the issue, means C3. Another issue is that roughly no real world project can be found which contains all the covered issues of a tool, thus the tool should be tested on different projects which needs too much effort.
- **Educational applications** There are many testing application exists mostly for educational purposes which contains a wide range of known issues, for example for security vulnerabilities OWASPs WebGoat (owa) is a good choice. The source of these types of applications can be used as a test target (input to SAT). One of the benefits of these this type is that they cover a comprehensive range of issues for in context. Moreover they are well documented and with regard to size they are small but reach in issues, so the test time is less than real projects.
- **Micro benchmarking suites** contains test suits which each contains one or more issues. The test suits might not even be executable. The approach allows us to easily indicates C1. Also by defining test cases having C3 is not a problem. One of the shortages of this type of benchmarking is how to declare C2, because the size of the tests are way smaller than a real project.

In order to compare the upcoming systematic benchmark studies we tried to find a list of characteristics. These characteristics is well defined by Johns and Jodeit which are vital for any systematic benchmark approach (Johns and Jodeit 2011):

- **CH1 - Testing tool capabilities individually** a systematic benchmark should be able to test each capability of the tool individually.
- **CH2 - Easy, correct, and iterative testcase creation** The system should be able to create easy, correct and iterative testcases. Testcases should be short and human readable and the system should allow the manual verification of a certain issue among defined tests.
- **CH3 - Automatic test execution and result evaluation** The test-code development is not a one-time perfect task. The test cases might not be perfect intially and the test designers will iteratively improve them to fit their needs. Therefore ease the creations and also the benchmark process, the system should be able to automatically and repeatedly run tests and evaluate the results.

3. Scanstud

The paper propose a micro benchmarking approach for systematic, fine-grained evaluation of Security Static Analysis Tools. They have provide a system in which test could be defined, run automatically and repeatedly and evaluate the results.

In the rest of this section we look over some of the challenges they have and how they overcome them. The man challenge is how to map the tools findings with created vulnerabilities in tests. Using line number to address the issues is not enough, because different tools propose different line numbers for a certain issue. They have overcome this issue by hosting each *testcode* on a dedicated application. The application only contains a single test case, maybe a real issue or a false positive. They differntiane the testcode and *host program*. *Host program* is the test case added by some other codes to make it parse-able, compile-able and execute-able.

The host program might produce some false positives. To address this issue they create 2 version of host program, one with testcode and the other without testcode and they run the preprocessing step on both of them. The results that occur in both of the processes are the false positivies caused by host program and will be removed from result report.

In order to automatically creating the testcodes, they have a list of template issues saved in files, for example opening a socket. Then there is an script that could insert each template in code, and automatically create different version of testcodes.

Scanstud also provides a way to manually verify the issue. This is normally done by running the host program which is a complete application to run.

4. Delta-Bench

The aim of this study is to address the limitations of existing Static Analysis Security Testing tool benchmarks such as lack of vulnerability realism, uncertain ground truth, and large amount of findings not related to analyzed vulnerability(Pashchenko et al. 2017). The paper provides a *synthetic benchmark* in order to mitigate the current problem of other bench-marking approaches, like bug injection and historical vulnerabilities. The first one is not easy to verify (Dahse and Holz 2014) and the historical approach only declares a subset of the *ground truth*. On the other hand in a synthetic benchmark atomic tests could be defined which is specifically related to just one vulnerability or a false positive. Thus bugs, or false positives could be clearly verified and unlike the historical approach it is easy to extend the system. The paper tries to investigate how synthetic benchmark scales in real software projects.

One of the main problems of using a big software project for bench-marking is the amount of unrelated alamars produced which is not related to vulnerability that one would like to benchmark, which is known as *Background Noise*. The paper addresses the issue by mixing the historical and synthetic approach. They used big well documented software projects and extracted the ground truth from their source code repository. The ground truth is what they expect a tool could find.

In order to find the vulnerable source code they extract 2 revisions of the code, C_{vuln} and C_{fixed}. C_{vuln} is the revision prior to C_{fixed} which has the targeted vulnerability and C_{fixed} is the fixed version. In the ideal case the tool output for C_{vuln} only contains the the related alerts. However the tool might produce some false positive alarms which is not related to the type of targeted vulnerability. The paper says that, the change in code from vuln to fixed version is just related to the targeted vulnerability, hence C_{fixed} should not contains alarms related to fixed vulnerabilities, but still has the false alarms. Thus if we subtract the outputs we reach the set of true alarms.

The main 5 steps of approach:

1. Find a proper project with enough documentation to find the fixed and vulnerable revisions.
2. For each vulnerability find C_{vuln}, C_{fixed} pairs
3. Use *diff* tool to find the modified code
4. Run the tool on both C_{vuln} and C_{fixed} versions.
5. Signals are the alerts differing between the tool outputs on C_{vuln} and C_{fixed}. To find the *background noise* they use the line number of reported alarms.

Unfortunately there is not any detail about the automation, reproduction and extension of the process. The best advantage of the approach is that it could remove the background noise, but it still has some limitations. First of all, the bug fix commit should only contains the codes reltaed to the fix, and if there be any other unrelated codes like refactoring, could lead to some differences in the alarms. The second limitation is the initial phase where a set of vulnerable, fixed bugs must be found in a real world project, thus it could be just applied to project with well documented source issues and fixes.

5. MuBenchPipe

The paper tries to evaluate different Java API-misuse detectors, static, dynamic and hybrid detectors. and therefore developed a MuBenchPipe (Amann et al. 2018) the first automated pipeline to benchmark API-misuse detectors. They have a data set called MuBench, which contains 90 Java API misuses, 73 investigated from real world projects, and 17 from a conducted survey. They have set up different experiments in order to evaluate different aspects of the tools.

MuBenchPipe automates many parts of the experiments and provides a command line interface to control them, such as retrieval and compilation of target projects, running detectors, and collecting their findings. There is a review website that reviewers can see all the findings. Some of them are labeled as potential and MuBenchPipe will not compute the experiment statistics unless they receive a review by at least 2 reviewers.

With regard to reproduction of experiments, there is a Docker image for both MuBenchPipe and the review website which make it easier to run experiments on different platforms. The review website allows independent reviews, while reviewers working on different workstations, and ensures the integrity using authentication. It is easy to extend MuBenchPipe facilitates the extension in 2 ways. First it has a simple data schema for misuse examples and new plugins (for detection or evaluation) could be added via a clear java interface with Maven.

6. Conclusion

References

Owasp webgoat project. URL `https://www.owasp.org/index.php`.

S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering*, pages 1–1, 2018. ISSN 0098-5589. .

B. Chess and J. West. *Secure Programming with Static Analysis*. Addison-Wesley Professional, first edition, 2007. ISBN 9780321424778.

J. Dahse and T. Holz. Static detection of second-order vulnerabilities in web applications. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 989–1003, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-15-7. URL `http://dl.acm.org/citation.cfm?id=2671225.2671288`.

M. Johns and M. Jodeit. Scanstud: A methodology for systematic, fine-grained evaluation of static analysis tools. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 523–530, March 2011. .

I. Pashchenko, S. Dashevskyi, and F. Massacci. Delta-bench: Differential benchmark for static analysis security testing tools. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 163–168, Nov 2017. .